A Programming Model for Reliable and Efficient Edge-Based Execution Under Resource Variability

Zheng "Jason" Song and Eli Tilevich
Software Innovations Lab, Virginia Tech
{songz,tilevich}@cs.vt.edu

***Abstract*—Edge computing applications use the computational, sensor, and networking resources of nearby mobile and stationary computing devices. Because dissimilar devices can provide these resources, one cannot predict which exact combinations of resources will be available at runtime. The resulting variability hinders the development of edge computing applications. To address this problem, we present a new programming model that employs a domain-specific language (DSL), through which the developer declaratively specifies a collection of microservices and how they invoke each other. Given a concise declarative service suite specification, the DSL compiler automatically generates an execution plan, carried out by our distributed runtime. The resulting programming model is both reliable and efficient. The reliability is achieved by enabling the developer to provide equivalent microservices as switch-over recovery strategies. The efficiency is achieved by the DSL compiler orchestrating the speculatively parallel execution of certain equivalent microservices. Our evaluation demonstrates the reliability, efficiency, and expressiveness of the programming model, which can help developers who need to cope with variable resources at the edge.**

I. Introduction

In contrast to cloud computing, edge computing processes data locally near its source (i.e., at the "edge" of the network), thereby reducing the network transmission load and communication latency. In addition, by leveraging the edge environment's sensor and networking resources, edge computing applications can take advantage of the local context and accelerate data transfer [3], [11], [4], [5], [9], [15], [2], [10].

The need to access nearby sensors and to reduce communication latencies requires that edge resources be orchestrated for a reliable and efficient execution. Nevertheless, software developers lack adequate programming support to be able to engineer such edge computing applications [18], [17], [16]. In recent years, microservices [1] have been embraced as an architecture that structures distributed systems modularly to clearly separate concerns. Microservices fit naturally the domain of edge computing, which coordinates the execution of multiple dissimilar computing devices. However, extant microservice frameworks are inherently cloud-based, and cannot be directly applied to edge-based environments.

Two primary factors hinder the use of microservices at the edge: (1) Cloud-based microservice architectures require that all executable resources be pre-deployed on the participating devices, which can be accessed by querying an Internet-based registry service. However, connected via local-area networks, edge-based resources can only be accessed within a limited physical area. (2) As a result, edge environments differ in their setups, making it impossible to rely on any standard set of edge-based resources. Hence, robust and efficient edge computing applications should be able to adapt to the available sets of resources in dissimilar runtime environments.

Consider obtaining environmental sensor data, such as temperature, humidity, or CO_2. A mobile application may need to keep track of up-to-date environmental data, specific to the device's current geo location. However, edge environments often possess dissimilar resources that can provide the necessary data. For instance, temperature can be read from a local sensor or be obtained by passing the location parameter to a web-based weather service. To fulfill these application requirements, developers need to either implement complex logic that covers all possible combinations of available edge resources, or hardcode the implementation for a particular edge environment with pre-deployed resources.

In this paper, we present a novel programming model for orchestrating reliable and efficient execution in edge environments with variable resources. Our model features a declarative domain-specific language (DSL) for orchestrating the execution of microservices at the edge. Our language is called MOLE (**M**icroservice **O**rchestration **L**anguag**E**). The MOLE compiler takes as input the declarative specification of microservices and produces a platform-independent execution plan. The MOLE runtime takes the generated execution plan as input, and adaptively steers the execution of the expressed functionality on the set of available devices.

The contribution of this paper is four-fold:

1) We present MOLE—a declarative DSL that enables programmers to express edge-based application as an ensemble of microservice executions; MOLE naturally supports redundant execution to adapt to opportunistically available resources.
2) We describe the MOLE compiler that generates platform-independent execution plans; the compiler automatically parallelizes microservice execution.
3) We design a novel microservice-based runtime architecture that supports MOLE programs to execute microservices on the available edge devices.
4) We evaluate MOLE, its compiler, and runtime system on a set of benchmarks and case studies.

The rest of this paper is organized as follows: Section II analyzes the problem of variable resources at the edge; Section III gives an overview of MOLE and its infrastructure; Section IV and V details the design of MOLE language and

978-1-7281-2708-8/19/$31.00 ©2019 IEEE
DOI 10.1109/EDGE.2019.00026

its runtime. Section VI discusses our implementation and empirical evaluation. Section VII compares MOLE to the related state of the art. Section VIII concludes this paper.

II. Problem Analysis

In this section, we demonstrate the difficulties of programming edge computing applications.

A. High Resource Variability at the Edge

Developing software for edge computing environments differs from that for the cloud. Developers can reasonably assume the high availability and reliability of cloud-based resources. Cloud providers are bound by the terms of Service Level Agreements (SLAs) to ensure their services remain up and running. Hence, because most failures in cloud-based systems are recovered from quickly, a simple retry to contact a temporarily inaccessible cloud service is a reasonable fault handling strategy [8]. However, in edge-based environments, the resource availability is likely to cause execution failures, triggered by the differences in the resource setups of edge environments.

Nevertheless, edge programming models [8], [12] continue to follow the fault handling strategies, originally introduced for cloud-based microservices — handling faults by retries and adjusting minor configuration setups (e.g., switching network connectivity methods, switching to devices capable of providing the same functionalities).

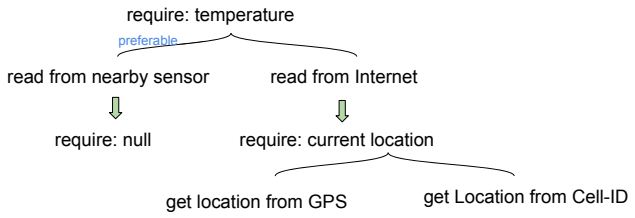

Fig. 1. Increasing Dependability by Increasing Redundancy.

We observe that edge environments can provide the same application functionality in a variety of ways. In the motivating example, the developer can either read a local temperature sensor or parameterize a web-based weather service with the user's geo location. As another example, consider detecting the breakout of fire in a building, different sensors (e.g., temperature, dust level, CO_2 level, etc.) can be combined to ascertain whether there is fire.

Hence, given the high variability at the edge, our programming model centers around the concept of *resource redundancy and makes it natural for the developer to specify alternative ways to provide the same functionality*. Continuing with the temperature example above, Fig. 1 shows a possible design flow, in which the developer first considers obtaining temperature by reading a local sensor, but then realizing that such sensors may be unavailable or disabled, would specify a back-up alternative of obtaining the required information from a web-service. Both alternatives provide equivalent functionalities with minor caveats. Local sensors are likely to provide higher accuracy, while weather web services are highly reliable, even when given a coarse-grained geo location. To obtain the location, multiple localization methods (e.g., GPS based, cell-id based, WiFi based) are equally suitable.

B. Complexity of Orchestrating Edge Microservices

To implement the redundancies, developers typically need to engineer high-complexity code, particularly if the resulting execution has efficiency requirements.

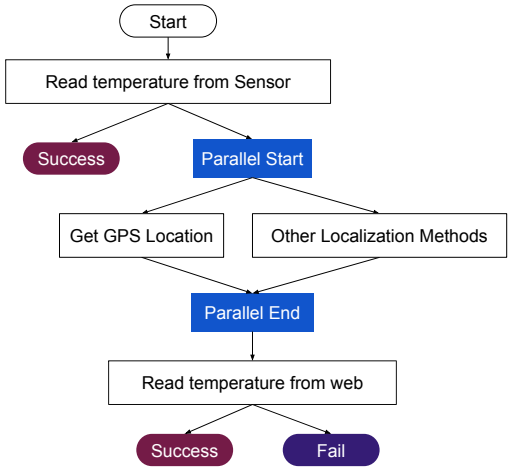

Fig. 2. Execution Sequence of Example Edge Application.

Let us revisit the temperature example above. Some localization methods can experience unexpectedly high latencies. To accelerate the overall execution, the developer may want to take advantage of speculative parallelism: spawn multiple localization methods at once, and proceed once any one of them returns successfully. Fig. 2 shows how speculative parallelism can be integrated into the execution flow.

There is an impedance mismatch between the simplicity of how developers can divide a required functionality into distinct functions (Fig. 1), and the complexity of orchestrating these functions to execute correctly and efficiently (Fig. 2). Existing programming models require that functions be explicitly arranged into an execution flow, thus unnecessarily burdening the developers.

III. MOLE Overview

Next, we give a brief overview of how developers can use MOLE to provision for edge applications. First we briefly introduce the system architecture, and then explain the basic system execution flow of MOLE applications.

A. System Architecture

Fig. 3 shows the MOLE system architecture, which comprises four major components: 1) a client device that requires distributed resources to accomplish an application functionality; 2) a local device that serves as a gateway by maintaining an up-to-date mapping between the available nearby devices and their resource capacities; 3) a microservice market, a cloud-based repository of executable code of all available

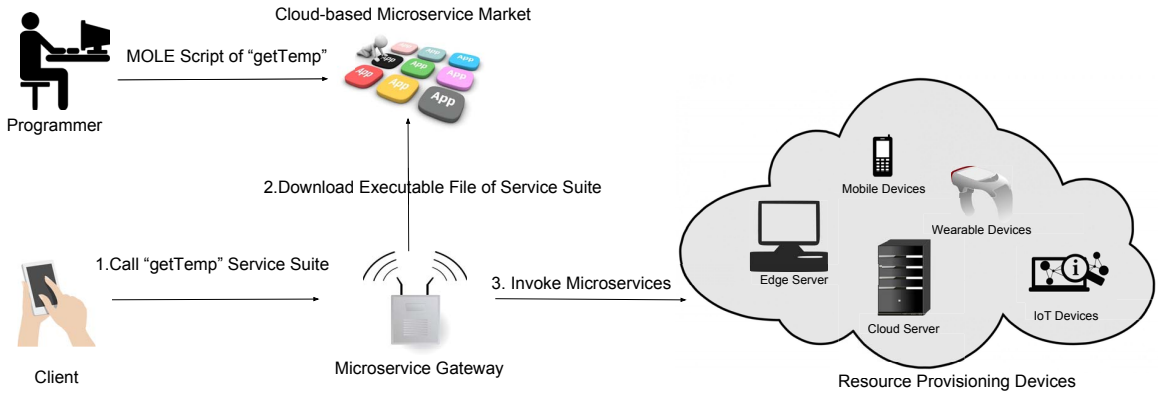

Fig. 3. Solution Overview.

microservices; 4) a set of local devices that provide their resources to applications.

Gateways: A typical microservice architecture features a centralized service registry, a collection of registered microservice-to-device mappings, with a remote interface through which clients can bind themselves to the microservices they want to invoke. Notice that if the registry is not replicated, it becomes vulnerable to the single point of failure. Besides, edge-based applications need to invoke microservices on the devices reachable via short-range communication methods (e.g., WiFi, Bluetooth), rendering cloud-based registries inapplicable. To meet these requirements, MOLE features a novel system component: an edge gateway, thus replacing the standard cloud-based service registry. Similarly to its cloud-based counterparts, the edge gateway maintains a registry of all the microservices provided by edge devices. At runtime, clients interact with the reachable gateway in its vicinity to execute microservices; the gateway interacts with the available devices on its clients' behalf. Since gateways form a network, in case a device hosting a gateway fails, clients proceed contacting the remaining gateways until reaching one of them.

Cloud-based Microservice Market: Our design leverages the **M**icro**S**ervice **M**arket (MSM for short), a cloud-based network component that combines features of application markets and service repositories [14]. By following the application market model, MSMs enable devices to automatically download the needed microservices for execution. By following the service repositories model, MSMs enable edge application developers to implement the needed functionalities as microservices, to be executed by the available devices in a given edge computing environment.

B. Service Suite Execution Model

To understand the general MOLE system flow, recall the "getting the temperature" example. An application running on a client device sends the request to execute `getTemp` service suite to a nearby gateway (Step 1). The gateway downloads the `getTemp` service suite from a cloud-based MSM (Step 2), and executes it by orchestrating the microservice invocations on the available devices at the edge (Step 3). The gateway continuously collects the microservice execution results, which drive the orchestration of the microservice invocations involved. Upon completing the service's execution, the gateway returns the final results to the client.

In the example above, `getTemp` comprises a collection of microservice invocations, which can be initiated by edge applications to obtain the functionality at hand. In the rest of the manuscript, we refer to such collections as **a service suite**.

Definition 1. Service Suite: implements an application functionality by orchestrating a collection of microservices.

The MOLE programming model enables service suite developers to declaratively specify how to orchestrate the execution of microservices. The MOLE compiler then translates these specifications into an execution graph, while optimizing the resulting edge based execution via speculative parallelism. The MOLE distributed runtime finally discovers the available devices to execute the specified microservices on them, as directed by the compiled MOLE specifications.

IV. MOLE DSL Design

Fig. 4 defines the syntax of MOLE in EBNF. Some of the key features are as follows:

- Each service suite is identified by a unique id, `Service Identity`. Service suites may take `Service Parameter`, which must be passed when the suite is invoked.
- A service suite comprises one or more `Microservice Invocation`'s, identified by unique IDs, and containing additional attributes explained next.
- A microservice invocation comprises the following attributes: 1) the `Device Selection` rules that guide how to select a device to run on; 2) the `Input Params` that specify the microservice's invocation parameters, some of which are hardcoded (indicated by `set`) while others are passed at runtime (indicated by `req` – short of "require"); 3) the `After Execution Rules` that specify what results should be returned (`ret`), and what the next suite execution step should be (could be either `exit`ing service suite , or invoke another `MS`).
- The execution procedure of a microservice can be controlled by execution parameters (`ep`), which is

```
<Service_Suite> ::= <Service_Identity> <Service_Description>
<Service_Identity> ::= "Service "String
<Service_Description> ::= "{"[<Service_Parameter>] <Microservice_Invocations>"}"
<Service_Parameter> ::= "global_input: "[<Input_Parameter_Name> ","]
<Microservice_Invocations> ::= [Microservice_Invocation]+

<Microservice_Invocation> ::= "MS:" <MS_Identity> "{" [<MS_Detail>]+ "}"
<MS_Identity> ::= String

<MS_Detail>::= <Device_Selection>|<Input_Params>|<After_Execution_Rules>
<Device_Selection> ::= "device:" [<Select_Rule> "."]+
<Select_Rule> ::= "select"|"sort" "(" String ")"
<Input_Params> ::= ("req": [<Param Name> ","]+)|("set:" [<Param_Name> "to" <Param_Value> ","]+)
<After_Execution_Rules> := "on." <Condition> ":" [ <return> ";"] [<redirection>]
  <Condition> ::= "success"|"fail"|"res."<Param_Name><Operation><Value> |"ep."<Param_Name><Operation><Value>
  <return> ::= "ret" [String ["as" String] "," ]+
  <redirection> :== <MS_Identity>|"exit"
```

Fig. 4. DSL EBNF Definition.

```
Service getTemp {
   MS: getTempSensorReading {
      device: select("Sensor.Temperature")
      on.success: ret temp
      on.fail: getTempbyLocation
   }
   MS: getTempByLocation {
      device: select("Internet")
      req: location
      set: ep.max_retry to 3
      on.success: ret temp
   }
   MS: getLocationByGPS {
      device: select("Location.GPS_PROVIDER")
      on.success: ret loc as location
   }
   MS: getLocationByCellID {
      device: select("Location.NETWORK_PROVIDER")
      on.success: ret loc as location
   }
}
```

Fig. 5. Source File of `getTemp` Service Suite

a special kind of `Input Params`. `ep` contains a fixed set of directives: `maxExecutionTime`, `maxRetry`, `retryOnOtherDevices`, and `counter`.

As a concrete example, consider the MOLE script in Fig. 5, which describes the `getTemp` service suite. `Service Identity` is `getTemp`, a service suite that takes no parameters. It comprises four `Microservice Invocation`: `getTempSensorReading` (m_1), `getTempbyLocation` (m_2), `getLocationbyGPS` (m_3), and `getLocationbyCellID` (m_4). Each microservice has `Device Selection` rules and `After Execution Rules`, while only m_2 needs `Input`.

A pair of microservices (m_1, m_2) can relate to each other in two ways: 1) *forward relationship*: m_1 invokes m_2 based on the suite's business logic; 2) *backward relationship*: m_1 has an input parameter, whose value must first be computed by invoking m_2. In a given suite, developers orchestrate the execution of microservices based on the concepts of forward and backward relationships.

To provide an intuitive programming model, MOLE requires that **only the forward relationship be explicitly defined** (e.g.,invoke m_2 iff m_1 fails). Backward relationships are automatically inferred based on the naming correspondences between the input and output parameters of the microservices in a suite (e.g., m_2 requires input parameter 'a', m_3 produces 'a' as its execution result, so the compiler orchestrates the correct execution sequence of $\{m_1, m_3, m_2\}$).

V. MOLE Compiler and Runtime

Fig. 6 shows how a MOLE script file is compiled, optimized, and executed. Upon completing a microservice suite, developers upload them to the mobile service market (MSM) containing the referenced microservices. Recall that MSMs are network components that combine features of service repositories and app markets. An MSM has the facilities for error checking, compiling, and optimizing MOLE suite specifications. The end result of processing a specification is an executable containing the service suite's **Execution Graph**, a self-contained repository for all the information required to efficiently execute the suite. Once a client invokes the edge application, the edge's gateway component downloads the compiled Execution Graph from the MSM and starts executing it. The execution also involves downloading the referenced microservices to the devices selected to execute them. If the gateway fails for any reason, the edge app's client can always start interacting with an alternate gateway component, thus providing a fail-over fault handling strategy. Upon successfully completing its execution, the suite returns the results back to the client, or an error if the execution failed for any reasons.

We first describe execution graphs, and then explain how MOLE scripts are compiled into execution graphs. Finally, we discuss how the MOLE distributed runtime executes execution graphs using a distributed microservice gateway.

A. Execution Graph Definition

An execution graph $G = (N, E, P)$ comprises a set $N = \{n|n = (t, m, d, p)\}$ of nodes, a set $E = \{e|e = (n_s, n_t, c, a)\}$ of edges, and a set of global parameters P that must be bound before an execution can start.

A **node** $n = (t, m, d, p)$ comprises the type t of the node, the microservice m related to the node, the device selection

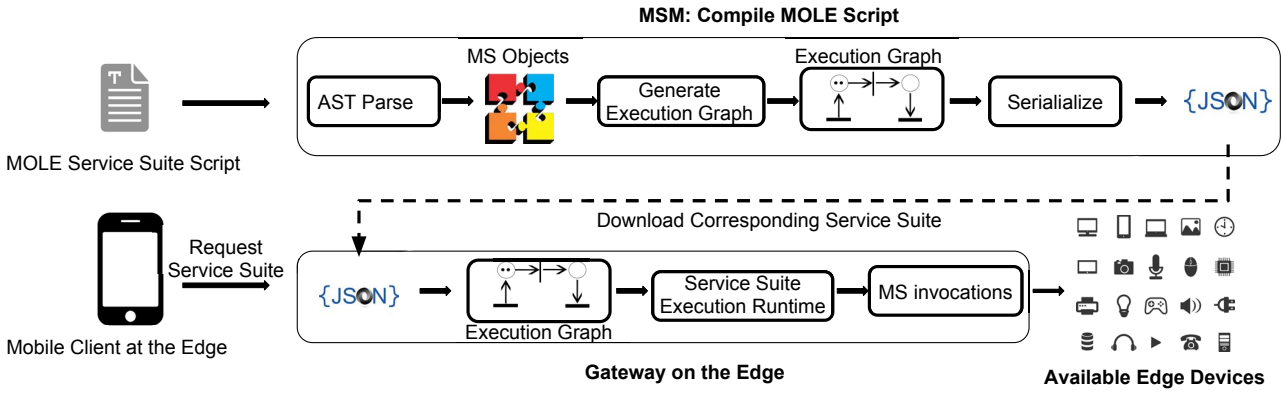

Fig. 6. DSL Parsing and Execution.

rules d of the node, and a set of required execution parameters p. An **edge** $e = (n_s, n_t, c, a)$, also written as:

$$e = n_s \xrightarrow[a]{c} n_t$$

indicates that when the execution results of a microservice node s fits a condition c ($c = null$ if the type of n_s is not a microservice), the next microservice node to visit is n_t, with a set of arguments a passed to it. n_s is called the *source* of e, n_t is called the *target* of e, e is an *outgoing* edge of n_s, and e is an *incoming* edge of n_t.

The possible **type** of nodes $t \in [M, E, ES, EF, PS, PE]$, where M stands for a microservice node, E for the entry node of the service suite, ES for the successful exit node, EF for the failure exit node, PS for a parallel start node, PE for a parallel end node. There can be only one entry node (n_E), one successful exit node (n_{ES}) and one failure exit node (n_{EF}).

The PS and PE nodes designate the start and the end of a **speculative parallel execution** block, respectively. Upon entering a PS node, all parallel branches start executing their first nodes (linked by the outgoing edge of the PS node) in parallel. A parallel branch may have multiple MS nodes to execute, and all parallel branches aggregate at the peer PE node. When all required parameters of the PE node have been provided by any combination of branches, the PE node starts to execute its next MS node, disregarding the completion statuses of the remaining parallel branches.

B. Generating Execution Graphs

The MOLE compiler transforms an input script file into an execution graph. The key idea of the transformation is to run a two-phase analysis: (1) control-flow analysis adds edges between pairs of microservices on the control path (e.g., if n_s fails, invoke n_t) and (2) data-flow analysis adds edges between microservices with data dependencies, (e.g., n_t takes as input the n_s's execution result). The required parallel blocks are added into the execution graph during the second phase.

Algorithm 1 controls the transformation in four basic steps:

(1) Initialize Nodes (Line 3 - 5): convert each microservice declared in the source script into "MS" nodes. The node structures encapsulate the microservice invocations, device selection rules, and required input parameters. Each graph also includes four special nodes: N_E (entry node), N_{ES} (execution success) , N_{EF} (execution failure), and $Null$ Node N_N, a temporary placeholder used at graph construction time.

(2) Initialize Edges via Control Flow Analysis (Line 6 - 17): parse the MOLE script to extract the conditional statement for each "MS" node. Recall that only the forward relationships must be defined explicitly. If a node's conditional statement is linked to another node (could be either "MS", N_{ES}, or N_{EF} node, which define the forward relationships), add an edge to the execution graph connecting the two nodes; otherwise, if it only generates data as output (e.g., 'on.success: ret

Algorithm 1 Generate Execution Graph.

```
1:  function GENERATEEXECUTIONGRAPH()
2:      ExecutionGraph eg ← ExecutionGraph()
3:      // Step 1: init nodes
4:      eg.nodes ← ParseMicroserviceNodes()
5:      eg.nodes ← eg.nodes + N_E, N_ES, N_EF N_N
6:      // Step 2: init edges by control flow analysis
7:      for all node ∈ eg.nodes do
8:          for all c ∈ node.conditions do
9:              if c.type == "invoke microservice" then
10:                 e ← Edge(node, c.target, c.condition, c.params)
11:                 eg.edges ← eg.edges + e
12:             else
13:                 e ← Edge(node, N_N, c.condition, c.params)
14:                 eg.edges ← eg.edges + e              ▷ link to node Null
15:             end if
16:         end for
17:     end for
18:     // Step 3: add edges by data flow analysis
19:     dataEdges ← N_N.getIncomingEdges()               ▷ GIE() for short
20:     loopNodes ← eg.nodes - specialNodes - dataEdges.s
21:     loopNodes.BFS()          ▷ Breadth-First-Search for adding edges
22:     // Step 4: find entry node
23:     entryNodeSet ← EmptySet
24:     for all n ∈ eg.nodes do
25:         if n.type=="ms" AND n ∉ dataEdges.s AND n.GIE()=null then
26:             entryNodeSet ← entryNodeSet + n
27:         end if
28:     end for
29:     if entrySet.size!=1 then
30:         Raise CompileError("Cannot Find Entry Node")
31:     else
32:         eg.edges ← eg.edges + (N_E, entrySet[0], null, null)
33:     end if
34:     return eg
35: end function
```

loc as location", which can be used to infer the backward relationships), add an edge to the graph, connecting the node with the special N_N node.

(3) Add Edges via Data Flow Analysis (Line 18 - 21): generate a set, initialized with the "MS" nodes, except those connected to the N_N node. All edges leading to the N_N node become "dataEdges" to provide missing arguments for other microservices. For each node's incoming edges, calculate whether the incoming edge's bound arguments can serve as the microservice's required parameters. If not, check if the missing parameters can be provided by "dataEdges". If only one "dataEdge" can provide the missing parameter(s), add the source of the "dataEdge" to the graph, between the current node's source node and the current node. If more than one "dataEdge" can provide the missing parameter(s), add a pair PS, PE of parallel blocks between the source node of the current node and the current node, and then add all "dataEdges" into the parallel blocks. When new edges are added, the edges' source nodes are added to the set, so they can be also properly processed. This step is actually applying the breadth-first search algorithm.

(4) Find Entry Node (Line 22 - 34): for all "MS" nodes, find those without any incoming edges or connected by "dataEdges". If only one such node is found, add an edge between the entry node and the found node. Otherwise, throw a compile error.

C. MOLE Runtime

MOLE features a distributed runtime system that efficiently and reliably executes compiled scripts. The runtime's pivotal component is an *edge gateway*, responsible for collecting the real-time status of surrounding edge devices, accepting service suite execution requests, downloading the corresponding compiled MOLE scripts from MSMs, and invoking the constituent microservices. Each microservice-executing device runs a light-weight HTTP server, which dispatches the referenced microservices by invoking their execution packages, provided on demand by MSMs.

To execute a compiled MOLE script, the edge gateway's runtime starts the execution at the entry node, moving through the connected nodes to the end node. The execution goes from node to node as follows. When visiting a microservice node, the runtime invokes the microservice, and determines what the next node should be based on the invocation results. For a parallel start node, the runtime spawns concurrent branches, with each branch proceeding along its own path and finally aggregating at the following parallel end node. For a parallel end node, the runtime waits until either the concurrent branches provide the required parameters, or all of them experience faults or timeouts.

VI. Evaluation

In this section, we evaluate the MOLE programming model and performance in a realistic use cases. Our evaluation seeks answers to the following questions:

- Can MOLE programs adapt to resource variability?
- Does MOLE offer acceptable execution efficiency?
- How hard is it to develop a MOLE program?

A. Setups

The following discussion first describes the experimental setup, then introduces the executed service suites, and finally reports on the performance characteristics.

The evaluation hardware setup comprises: 1) a wireless router, running the OpenWrt OS; 2) a Chromebook; 3) two Android smartphones; 4) a Raspberry PI; 5) a Dell desktop serving as the edge server, and 6) an AWS cloud-based server (not shown in the Figure). Devices 2)-5) are connected to the wireless router, thus forming a wireless local area network.

A DS18B20 temperature sensor is connected to the Raspberry Pi via general purpose input/output (GPIO). The Raspberry PI hosts a web server that handles POST requests by invoking the corresponding microservice executables. "getSensoryTemperature" is pre-deployed on this device.

The NanoHttpd servers on Android devices invoke the corresponding microservices via reflection in response to incoming HTTP POST requests. One of these devices is configured to provide a fine-grained location, while the other one a coarse-grained one.

The Dell desktop plays two roles: the edge gateway and the edge server. It runs an HTTP server, and a MySQL database. Each edge device communicates with the edge gateway via HTTP to register their microservices; the gateway then persists this information in its database. As an edge server, it runs microservices, such as querying a web service to get the temperature in a given location.

B. Service Suite Execution

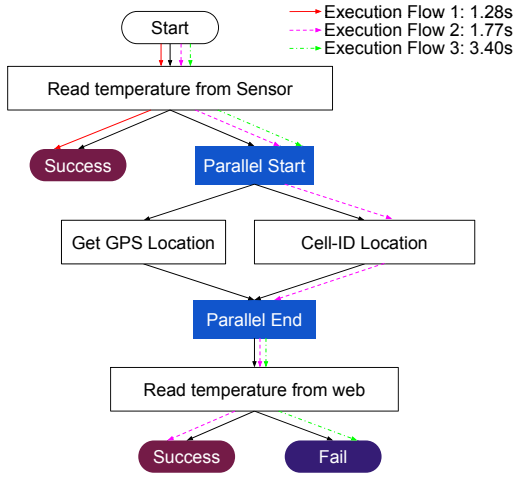

Fig. 7. Execution Time of Different Availability of Microservices.

End users access a dynamic web page from the Chromebook, which contains a JavaScript function that retrieves the service suite's name, connects to the local gateway (by querying the wireless router), sends the service suite execution request to the gateway, and blocks until receiving the results.

To evaluate how MOLE programs adapt to resource variability, we run our experiments in three dissimilar execution environments. The execution results fit the generated execution graph, as shown in Fig. 7:

A Make the Raspberry PI and the temperature sensor available. In this execution environment, the overall execution result obtained by the Chromebook is the temperature measured by the temperature sensor. The average execution time of 5 runs is 1.28 seconds.

B Shut down the Raspberry PI, to make the temperature sensor unavailable. In addition, enable the fine/coarse-grained localizations for the two Android devices. In this run, the overall execution result is the temperature of a geo-area, which differs from the first result. The overall execution takes 1.77 seconds. Please note that in our implementation, the GPS localization takes 2 seconds and the cell-network localization takes less than 1 seconds. This result indicates that even though two localization methods are all initialized, the execution continues when the cell-network location is returned.

C Disable both the temperature sensor and two localization methods. In this run, the parallel start node initializes two threads for two localization methods, but none of them goes to the parallel end node. The timeout for the latch count down is set to 3 seconds, with the parallel end node being reached after the timeout. The parallel end node fails to collect all necessary parameters for its connected node (`getTempByLocation`), so it triggers the execution fail condition of its connected node, thus causing the "Execution Failure" of the service suite. The average execution time of 5 runs is 3.4 seconds.

C. Programming Effort

Fig. 5 lists the source code of the `getTemp` service suite. It takes only 21 lines of code to specify the parameterization of and control-flow between 4 realistic microservices. Under any programming model, programmers have to implement the application functionalities, but representing them as microservices eases reuse. Each microservice is likely usable in multiple scenarios.

A particular strength of the MOLE programming model is how it accommodates change. Consider adding an alternate localization method for MS `getTemperatureByLocation`. Unlike the current two localization methods, the new method operates in two steps: 1) obtain the current IP address; 2) get the location from the IP address using a web service. This change requires only 6 additional lines of code.

D. Reliability Evaluation

To assess how reliable MOLE programs are, we simulate the execution of a suite under different failure conditions. This simulation varies the failure rate of each microservice execution between 10%, 20%, 30%, and 90%. Two types of failures apply: (1) no device is available to execute a given microservice; (2) the selected device fails to successfully execute a given microservice. We compare the resulting

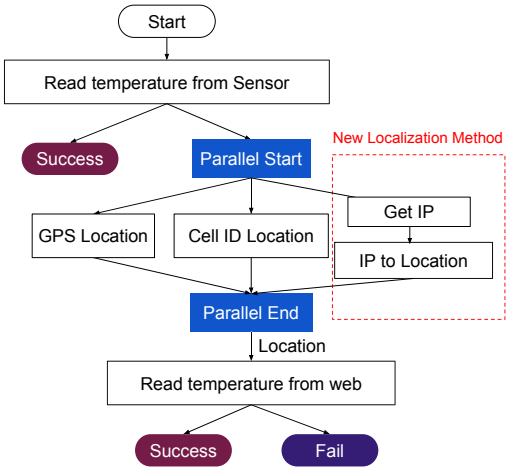

Fig. 8. New Execution Graph.

```
1  Service GetTemp {
2    ...
3
4      MS: getIP{
5          device:has("INTERNET")
6          on.success:ret ip;
7      }
8
9      MS: IP2Location{
10         req:ip
11         on.success:ret location;
12     }
13 }
```

Fig. 9. Adding a new localization method to the service suite

reliability levels of three scenarios: 1) obtain temperature from a temperature sensor; 2) execute service suite `GetTemp` with two localization methods; 3) execute this service suite with one additional IP-based localization method. Fig. 10 shows that compared with scenario 1, the service suite improves its reliability, especially when the failure rate is around 50% (it improves the reliability of scenario 1 by 37.5%). Besides, by comparing scenarios 2 and 3, we see how introducing an alternative localization method increases the overall reliability of the service suite execution. When two existing localization methods fail, the suite can still successfully complete its execution. However, the increase may not seem as striking, as the two alternative localization methods already exhibit considerable reliability.

E. Efficiency Evaluation

Next, we measure how efficient a MOLE program is. We set the execution failure rate of each microservice between 10%, 20%, 30%, and 90%. We compare the total execution time of three scenarios: 1) sequential execution, which runs one microservice at a time; 2) the `GetTemp` service suite executing its two localization methods in parallel; 3) the improved `GetTemp` service suite executing three localization methods in parallel. For each failure level, we repeat each execution scenario 100 times, and record the average execution time. Fig. 11 shows that MOLE can increase the base line of the

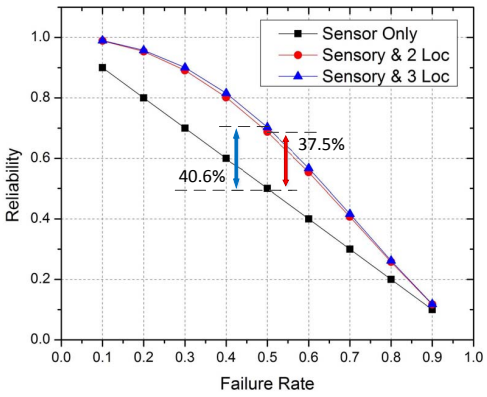

Fig. 10. Reliability W/ W/O MOLE.

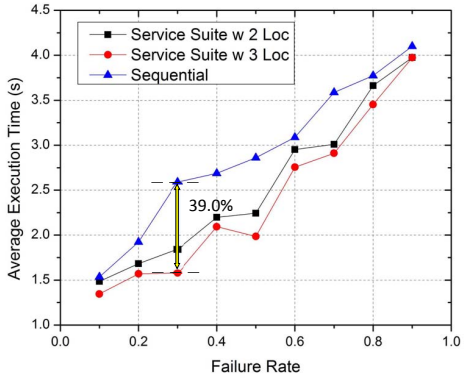

Fig. 11. Efficiency W/ W/O MOLE.

sequential execution by 12-17% by leveraging speculative parallelism. At most, MOLE saves 39% of execution time when the failure rate is 30%. Besides, because IP-based localization is known to be more efficient than other location methods, MS `getTemperatureByLocation` can proceed without waiting for the other two slower localization methods to complete, thus improving the overall efficiency.

VII. Related Work

Recent survey papers treat the issue of programming edge applications as both a serious technical challenge and a research opportunity [17], [16]. [7] introduces a P2P message exchange based programming model, by which programmers develop functionalities for each distributed component and handle their communication. However, such programming models can only be applied to execution environments with fixed resources. [13] considers the resource dynamicity of edge computing environments, and models edge service provision as a QoS-constrained resource selection problem. [6] provides a data-flow based programming model, also applicable to edge environments with dynamic resources. However, these approaches neglect failure handling, an essential provision given the high failure ratio of edge-based execution.

VIII. Conclusion

This paper has presented MOLE, a declarative DSL for developing reliable and efficient edge computing applications. MOLE adopts the microservice architecture, with edge functionalities provided as microservices, downloaded and executed by available devices at runtime. MOLE enables developers to concisely express how to parameterize microservices, and automatically orchestrates their execution flow. MOLE exploits the presence of equivalent microservices to orchestrate both fail-over and speculatively parallel execution workflows. Our evaluation has demonstrated the expressiveness, reliability, and efficiency of the MOLE programming model.

Acknowledgement

This research is supported by the National Science Foundation through the Grant CCF-1717065.

References

[1] What are microservices. http://microservices.io/.
[2] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: a system solution for sharing i/o between mobile systems. In *MobiSys'14*, pages 259–272. ACM, 2014.
[3] K. Bhardwaj, S. Sreepathy, A. Gavrilovska, and K. Schwan. Ecc: Edge cloud composites. In *MobileCloud'14*, pages 38–47. IEEE, 2014.
[4] X. Chen. Decentralized computation offloading game for mobile cloud computing. *IEEE TPDS*, 26(4):974–983, 2015.
[5] X. Chen, L. Jiao, W. Li, and X. Fu. Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM TON*, 2015.
[6] N. K. Giang, R. Lea, M. Blackstock, and V. C. Leung. Fog at the edge: Experiences building an edge computing platform. In *IEEE EDGE'18*, pages 9–16. IEEE, 2018.
[7] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.
[8] Y.-L. Hu, Y.-Y. Cho, W.-B. Su, D. S. Wei, Y. Huang, J.-L. Chen, Y. Chen, and S.-Y. Kuo. A programming framework for implementing fault-tolerant mechanism in iot applications. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 771–784. Springer, 2015.
[9] B. Jones, K. Dillman, R. Tang, A. Tang, E. Sharlin, L. Oehlberg, C. Neustaedter, and S. Bateman. Elevating communication, collaboration, and shared experiences in mobile video through drones. In *DIS'16*, pages 1123–1135. ACM, 2016.
[10] H. Liang, H. S. Kim, H.-P. Tan, and W.-L. Yeow. Where am i? characterizing and improving the localization performance of off-the-shelf mobile devices through cooperation. In *IEEE NOMS'16*.
[11] R. Loomba, R. de Frein, and B. Jennings. Selecting energy efficient cluster-head trajectories for collaborative mobile sensing. In *GLOBECOM'15*, pages 1–7. IEEE, 2015.
[12] S. Qanbari, S. Pezeshki, R. Raisi, S. Mahdizadeh, R. Rahimzadeh, N. Behinaein, F. Mahmoudi, S. Ayoubzadeh, P. Fazlali, K. Roshani, et al. Iot design patterns: Computational constructs to design, build and engineer edge applications. In *IoTDI'16*, pages 277–282. IEEE, 2016.
[13] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar. Towards qos-aware fog service placement. In *Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on*, pages 89–96. IEEE, 2017.
[14] Z. Song, M. Le, Y.-W. Kwon, and E. Tilevich. Extemporaneous mobile service execution without code sharing. In *HotPOST'17*, pages 181–186. IEEE, 2017.
[15] S. Sur, T. Wei, and X. Zhang. Autodirective audio capturing through a synchronized smartphone array. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 28–41. ACM, 2014.
[16] B. Varghese and R. Buyya. Next generation cloud computing: New trends and research directions. *arXiv preprint arXiv:1707.07452*, 2017.
[17] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. Challenges and opportunities in edge computing. In *IEEE SmartCloud*, pages 20–26. IEEE, 2016.
[18] S. Yi, C. Li, and Q. Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, pages 37–42. ACM, 2015.