Adaptive Management of Volatile Edge Systems at Runtime With Satisfiability

COSMIN AVASALCAI, CHRISTOS TSIGKANOS, and SCHAHRAM DUSTDAR,
Distributed Systems Group, TU Wien

Edge computing offers the possibility of deploying applications at the edge of the network. To take advantage of available devices' distributed resources, applications often are structured as microservices, often having stringent requirements of low latency and high availability. However, a decentralized edge system that the application may be intended for is characterized by high volatility, due to devices making up the system being unreliable or leaving the network unexpectedly. This makes application deployment and assurance that it will continue to operate under volatility challenging. We propose an adaptive framework capable of deploying and efficiently maintaining a microservice-based application at runtime, by tackling two intertwined problems: (i) finding a microservice placement across device hosts and (ii) deriving invocation paths that serve it. Our objective is to maintain correct functionality by satisfying given requirements in terms of end-to-end latency and availability, in a volatile edge environment. We evaluate our solution quantitatively by considering performance and failure recovery.

CCS Concepts: • **Computer systems organization → Distributed architectures;**

Additional Key Words and Phrases: Resource management, edge computing, adaptive systems, distributed systems

ACM Reference format:
Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. 2021. Adaptive Management of Volatile Edge Systems at Runtime With Satisfiability. *ACM Trans. Internet Technol.* 22, 1, Article 26 (September 2021), 21 pages.
https://doi.org/10.1145/3470658

1 INTRODUCTION

One of the primary premises of edge computing is the utilization of available resources close to end devices, at the edge of the network. This paradigm extends the typical cloud viewpoint, aiming to satisfy stringent requirements such as low latency and high availability desired in emergent microservice-based applications. However, edge computing is characterized by a high degree of distribution that introduces volatility—computational nodes that participate on edge systems are often heterogeneous, mobile, and spatially distributed, and may fail or leave the system often [27]. Recent developments in DevOps and software design advocate dividing applications' functionality

This research was partially supported by EU'H2020 Marie Skłodowska-Curie grant agreement No. 764785 FORA, FWF Austria project M 2778-N "EDENSPACE," and the Austrian Federal Ministry of Science in the CPS/IoT project.

Authors' address: C. Avasalcai, C. Tsigkanos, and S. Dustdar, Distributed Systems Group, TU Wien, Vienna, Austria; emails: {c.avasalcai, christos.tsigkanos, dustdar}@dsg.tuwien.ac.at.

1533-5399/2021/09-ART26 $15.00
https://doi.org/10.1145/3470658

into small, modular, and easily deployable microservices. Those may be independently deployed, updated, scaled, or migrated according to various criteria. The overall application's functionality is then defined in terms of an invocation sequence among those microservices. Individual microservices may have resource requirements—for instance, some dependency on local data or dedicated hardware, to support, for example, machine learning functionalities. The overall application, as a composition of those microservices, may need to adhere to certain requirements. Fundamental ones, the importance of which is exacerbated on edge computing settings, are exhibiting a certain availability, and enjoying low end-to-end (e2e) latency. During the application's lifespan, deployment should take such requirements into account, something that is non-trivial and requires novel resource management techniques to deploy the application and maintain its correct functionality at runtime.

Resource management in edge computing has considered various aspects—resource placement, migration, and discovery among others [24]. We focus on combining *resource placement* with *resource migration*, to deploy and host a microservice-based application on an edge system satisfying its latency and availability requirements. We tackled *latency* in previous work, where we proposed seamlessly deploying applications such that their latency requirements are satisfied [2, 3]. Existing approaches to tackling availability have been based on scaling up resources in pools/clusters of nodes hosting microservices, such as OpenFaaS, Kubernetes, and Docker Swarm.[1] In the case considered in this article, however, we are not concerned with scaling or elasticity, but with decentralized systems where devices hosting microservices may fail or leave the network. Thus, the systems we target for deployment are highly volatile, raising challenges on dependable execution of microservice-based applications spanning multiple hosting nodes.

Specifically, we target decentralized edge-intensive systems where unforeseen behaviors at runtime might hinder system stability as far as deployment is concerned. Computational nodes where microservices are deployed may fail or leave at any point, or network circumstances may change—as such, deployment decisions previously made may be rendered obsolete. Therefore, we cast our proposal within self-adaptive systems and tackle the problems of volatility and distribution where microservices, using resources of hosting nodes, typically need to be executed in some specific sequence. Nodes may fail, and a specific invocation path may need to change, but it should still satisfy the application's latency and availability requirements. We interpret the preceding problem as two separate goals in the systems we target: (i) placing appropriately microservices to nodes (ii) and reacting to instabilities caused by failure of nodes.

During the *microservice placement stage*, we ensure that applications' resource requirements are met by allocating microservices appropriately over edge nodes. This stage takes into account the desired application availability along with known failure probabilities of individual hosting nodes. Availability is pursued by replication; this is a costly step, as it involves migrating and moving around container images—as such, it should be performed as infrequently as possible. During the *invocation path stage*, we ensure functionality by finding an invocation sequence among nodes hosting microservices satisfying latency constraints. As the system is assumed to be unstable, with nodes failing or leaving the system at any point, an invocation path may be disrupted and a new one must be derived, which must adhere again to the application's availability and latency requirements.

The proposed framework builds on the premise that an application model consists of microservices that should be invoked in some specific sequence to fulfill the desired functionality. The application model is accompanied by a set of requirements—that is, the microservices' resource requirements, the maximum end-to-end (e2e) latency that an invocation path should exhibit, and

[1]https://www.openfaas.com;https://kubernetes.io;https://www.docker.com.

the desired application's availability. Based on these, the framework deploys and maintains the application over the edge system. We consider that the target system consists of spatially distributed edge devices. However, we are not in a static setting similar to mobile edge computing where edge servers reside at certain locations—instead, we assume volatile systems where service operators have no control over nodes joining/leaving the network, which are often operated by end users.

The problem setting treated in this article can be summarized as follows. The edge system is defined as (i) microservices hosted on nodes (ranging from single-board computers to server-class data center hosts) forming a network, (ii) every node is network reachable from any other node, (iii) microservice instances are replicated to ensure availability, (iv) nodes may have high failure likelihoods, and (v) several invocation paths among microservices hosted are possible, each characterized by an e2e latency. Our framework encodes the two problems encountered—application deployment and its runtime management—within Satisfiability Modulo Theories (SMT) [5], where placement is captured as constraints in first-order logic while latency requirements are encoded with integer linear arithmetic. Thus, we provide guarantees—if a solution exists, it is always found at runtime by a solver situated in some edge node and is always correct. Those two problems correspond to two Monitoring-Analysis-Planning-Execution (MAPE) loops [15]. Finally, we evaluate performance by measuring the execution time required to deploy and maintain an application as well as investigate its recovery from emergence of volatility.

The rest of the article is structured as follows. Section 2 summarizes the related work on both service placement and adaptive techniques. In Section 3, we present the overview of our proposed solution and introduce a motivational example. Section 4 defines the application and network considered in this article. In Section 6, we describe the implementation details of our resource provisioning technique, whereas in Section 7, we describe our stabilization technique. Section 8 presents the methodology and results of our evaluation regarding both provision and stabilization. Finally, Section 9 concludes the article and provides an outlook on future work.

2 RELATED WORK

In this section, we discuss existing techniques found in the literature for deploying and maintaining applications on edge systems. We divide related work into two categories: service placement and adaptive techniques. In the former, we present resource placement efforts for deploying applications. In the latter, we focus on solutions that utilize adaptive techniques.

2.1 Service Placement Techniques

Multiple works tackle the resource placement problem in edge and fog settings. Typically, two main objectives control the service placement—that is, decreasing the latency by moving some services from the cloud closer to the end user and offload services from constrained devices to preserve energy and increase performance. Mahmud et al. [18] describe a latency-aware policy aiming to satisfy the latency requirements and optimize the utilization of fog node's available resources. In the target fog system, the authors consider two types of applications to be deployed: latency-sensitive and latency-tolerant applications. The proposed decentralized placement algorithm aims to place the applications between fog node clusters and cloud considering the applications' latency requirements. Furthermore, to optimize the fog node's resource utilization, an optimization technique using linear programming is proposed that minimizes the number of computational active fog nodes found in a cluster. Skarlat et al. [23] propose an optimization service placement algorithm, using the genetic algorithm to find a services placement strategy on fog nodes. The authors first organize computational nodes into a hierarchy, where fog nodes are grouped into colonies. The main idea is to distribute any application submitted to a fog colony to the controlled fog nodes found closer to the end user. In case there are no available resources in the current fog colony, the technique always tries to map the application to its neighbor colony. Ultimately, the application is sent to the cloud if there are insufficient available resources on the fog layer. Wobker et al. [28] introduce a fog computing platform to deploy and manage fog applications. The service placement technique is based on Kubernetes and uses a labeling system that enables the mapping of application components on fog nodes based on the application's requirements like memory, computational power, and storage. Another approach [6] proposes a placement technique that supports application deployment in a fog computing architecture considering quality of service requirements like latency and bandwidth. The proposed solution consists of two procedures: preprocessing and resource placement. The former filters the available fog nodes and cloud data centers according to the application's resource requirements, reducing the search space. The latter finds a single deployment strategy using a backtracking and heuristic approach.

Until now, all service placement techniques focused on migrating an application from the cloud closer to the edge of the network. However, such techniques may be used for offloading service computation from a resource-constrained device. Avgeris et al. [4] propose a service offloading technique focusing on the efficient utilization of edge servers. For this purpose, the authors introduce a two-level resource allocation mechanism that allows users found in the area of an edge server to offload computational tasks. However, the mechanism considers a cluster of edge servers without taking into account other edge clusters or the mobility of users. De Maio and Brandic [19] tackle the same problem of service offloading from mobile devices. In this case, they propose a multi-objective heuristic approach to allocate computational services to nearby edge nodes considering users' satisfaction as well as providers' profit. Chen et al. [7] propose a resource allocation technique based on deep reinforcement learning to offload tasks from resource-constrained devices to MEC servers. The objective is to find an optimal resource allocation strategy that considers latency, energy consumption, and radio transmission bandwidth. Multiple other works undertake the problem of service placement in different scenarios [10, 13, 17]. As can be observed, those efforts targeting placement have in common that the main objective is to ensure latency fulfilment upon placement of services.

Zhu and Huang [30] present EdgePlace, a heuristic technique capable of finding a placement strategy that can minimize the resource unit cost and increases availability. Depending on the application's demands, the approach makes a trade-off between resource utilization, bandwidth utilization, and availability. The technique receives as input a service graph consisting of multiple service chains that must be deployed on MEC servers; a service corresponds to a VM. In this case, to achieve service availability during the service placement phase, constraints like affinity and anti-affinity are used when deploying a service graph. However, in the case of host failure, functionality is recovered by migrating the service chain to another host. Sangolli et al. [22] propose an edge platform aiming to guarantee high service availability and minimize latency. For this purpose, the authors describe a real-time service migration technique to migrate services among nearby edge nodes when an edge node becomes unresponsive or has a high resource utilization. Daneshfar et al. [8] propose a service allocation technique with a central controller capable of mapping users' requests to fog nodes where services reside. In this case, service availability is inherited from nodes' availability. Furthermore, the authors do not consider the possible dependencies between services, each service representing a stand-alone entity. Lera et al. [16] present an availability-aware service placement to ensure objectives as the application's e2e latency and availability. The technique consists of two different stages: (i) break the fog network into smaller and better-connected communities and choose a community where the entire application is placed and (ii) map the application's services between the nodes found in the chosen community. Availability is achieved by deploying an entire application in one fog community since the fog nodes have high connectivity. Furthermore, the authors consider that in a fog system, only the communication link may fail. As a result, the application can continue to operate by reaching the fog node using a different communication path. However, if a node fails and becomes unreachable, the system cannot ensure the correct application functionality.

2.2 Adaptive Techniques

Generally, resource placement techniques do not account for the volatility of edge systems, where nodes have a high failure probability. As a solution to this challenge, research has strived to propose adaptive mechanisms based on migrating services between nodes. Govindaraj et al. [12] present an approach to perform smart resource allocation, allowing them to achieve live migrations on demand. Since performing a migration is an expensive task, the solution tries to minimize the migrations required while maintaining the round trip time for each device under a certain threshold. Gonzalez et al. [11] introduce a VM migration and placement technique for fog environments. The technique consists of two parts, a proactive VM migration approach that uses user's mobility predictions and a VM placement approach based on Integer Linear Programming (ILP). The latter aims at improving the VM placement on selected fog nodes. Kassir et al. [14] propose an adaptive distributed service placement technique based on Least Ration Routing capable of migrating VMs to other locations when there are changes in the network, such as a change in node's location, a change in network size, or network congestion appears. Mseddi et al. [20] describe an online intelligent resource allocation solution capable of determining the optimal service mapping on fog nodes and find a migration strategy considering the node's available resources. The main objective of this work is to maximize the number of satisfied user requests considering latency as a quality of service requirement. Rossi et al. [21] introduce a self-adaptive technique for deploying microservice-based applications in the cloud. The solution is using reinforcement learning techniques and has a two-layered hierarchical approach, based on MAPE cycles, capable of self-adapting based on a learned microservices scaling threshold. In this case, the authors choose a decentralized approach, where the first layer provides application-level feedback to the second layer, which takes decisions at the microservice level.

From the preceding related works, we can observe that different approaches are applicable for different problem settings. Heuristic approaches are applicable in situations where the target system does not exhibit high uncertainty, allowing for the necessary execution time to find near-optimal solutions. Typically, heuristic approaches are used when high scalability and maximization/minimization of available resources or application requirements are required. In contrast, in situations where for specific types of application and systems training data can be obtained, learning approaches appear promising. Such learning approaches are also suitable for finding optimal resource offloading in a distributed manner or adapting to changes found in the application or the target system. Similar to SMT, in an ILP approach, all objectives and requirements are expressed as constraints. However, the difference is that ILP techniques aim for an optimal solution. Thus, we argue that SMT fits well the problem setting where qualities like fast reaction and providing guarantees that a found solution satisfies the application's requirements are desirable. Overall, we distinguish ourselves from the works previously mentioned by (i) proposing an adaptive framework capable of ensuring availability on edge systems, (ii) avoiding container migration and instead changing invocation paths, and (iii) when an application is placed on edge nodes, the required number of replicas is adjusted considering nodes' failure probabilities and the application's availability requirement.

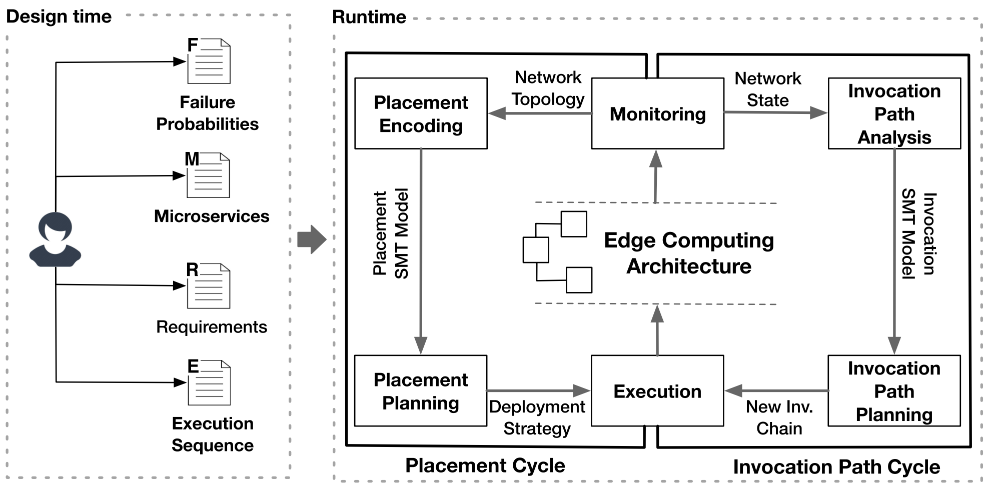

Fig. 1. Adaptive framework overview featuring two adaptive cycles at runtime, in charge of placing the application and managing the invocation path across microservices, respectively.

3 ADAPTATION FOR VOLATILE EDGE SYSTEMS

Edge-intensive systems typically heavily take into account runtime aspects, as unforeseen as well as emergent system behaviors might hinder system stability; computational nodes may fail or leave at any point, network circumstances may change, and assumptions taken into account for a deployment may be rendered obsolete. As such, we cast our framework within self-adaptive systems—microservices need to be executed in some way depending on resources that are located in nodes that may fail. Microservices require specific resources, and "some way" refers to an invocation path among them that needs to satisfy some desired property. We interpret the preceding problem as two separate goals in the systems we target: (i) placing appropriately microservices to nodes and (ii) reacting to instability caused by failure of nodes. Within our approach, a MAPE [1] cycle addresses each system goal, where both take place at system runtime (Figure 1).

The *placement cycle* ensures that applications' resource requirements are met, by allocating microservices appropriately over participating edge nodes. To do so, it monitors the state of the system, namely which nodes are available and which resources they currently have, through the monitoring activity. Subsequently, the placement encoding activity takes into account some desired availability factor of the application along with known failure probabilities of nodes (assumed to be provided at design time[2]) and models the problem in a representation amenable to analysis. The model is then supplied to the placement planning activity, which produces a *plan* by replicating microservices accordingly throughout the system and ensures that the overall application availability is satisfied. Finally, the plan is executed upon the system, by actual allocation of microservices upon nodes.

The *invocation path cycle* is responsible for devising a sequence of nodes where microservices reside among the ones currently deployed such that applications' e2e latency and availability are satisfied. As the system is assumed to be unstable, with nodes failing or leaving the system at any point, the cycle is triggered when system changes are detected by the monitoring activity. Recall that nodes host replicas of various microservices, but nodes themselves have communication links

[2]Failure probabilities may be also learned at runtime, something which we identify as an interesting avenue of future work.

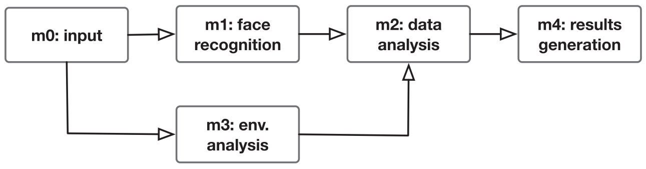

Fig. 2. Public-safety application model and its execution sequence.

of various qualities between them, which are also monitored. When changes occur and an existing invocation path is disrupted, a new one must be devised by the invocation path planning activity.

Notice that the two cycles are triggered independently and upon different premises. The *placement cycle* is triggered initially and then sporadically, when radical changes in the network appear. Conversely, the *invocation path cycle* is triggered with every change in the system state, to produce a satisfiable path, and as such targets non-radical system state changes, when the system merely needs to be *stabilized.* If the invocation path cycle fails to succeed, then placement needs to be attempted again. The difference among the two adaptive behaviors—targeting infrequent and radical change versus minor instability in invocation paths—is exploited in the design of the framework's runtime behavior, as described in the following sections.

Running example. Consider a public-safety application deployed in a smart city scenario and within the circumstances of a special event, such as a festival taking place within the city. Usually, large crowds of people attend such events, leading to an increase in different types of user-facing applications deployed in the area. A sudden request for hosting multiple applications on the current infrastructure renders it inadequate to meet each applications' requirements. However, there are many unused resources owned by various participants, such as within user's smart devices found in the vicinity, which can be used for the benefit of the collective. Note how an application operator does not control the users' devices—devices may leave the system at any time for any reason.

In our example, the public-safety application must be deployed in the system for the duration of the entire event. The application aims at citizen safety by analyzing the surroundings to find suspicious cases such as an unattended package or a dangerous activity in the crowd. Thus, once data is analyzed, the application helps authorities for prevention. As one can imagine, response time is critical in such situations, as well as maintaining correct functionality throughout the entire event. The application may not reside entirely in a remote, centralized cloud facility due to low latency and high availability—instead, a placement closer to the event location is desired. The application consists of different microservices such as (i) *face recognition*, (ii) *environment analysis*, and (iii) *data analysis*, among others. Furthermore, the application has a given execution sequence reflecting business logic, illustrated in Figure 2. A maximum e2e latency of *12 units* and a desired availability of *0.5* is assumed. The target system consists of multiple nodes that may range from powerful server-grade edge nodes (e.g., in the event premises) to user-operated mobile nodes. Each node has an associated failure probability and each communication link has a latency, assumed to be known. The objective is to satisfy all the application's requirements during its life span—at runtime.

We distinguish three characteristic stages that an edge system may be found in during the lifespan of an application. Those represent the initial placement and start of the application's execution, the emergence of some disruption due to node failure, and then the stable system state after the corrective action employed by our approach. Those are illustrated in Figure 3; specifically:

- *Initial placement*: Microservices are mapped; this includes replication of certain microservices across nodes, to account for possible future node failures. During this process, both the *placement cycle* and the *invocation path cycle* are utilized in sequence. The former places the required microservices and their replicas, whereas the latter aims at finding a satisfi-

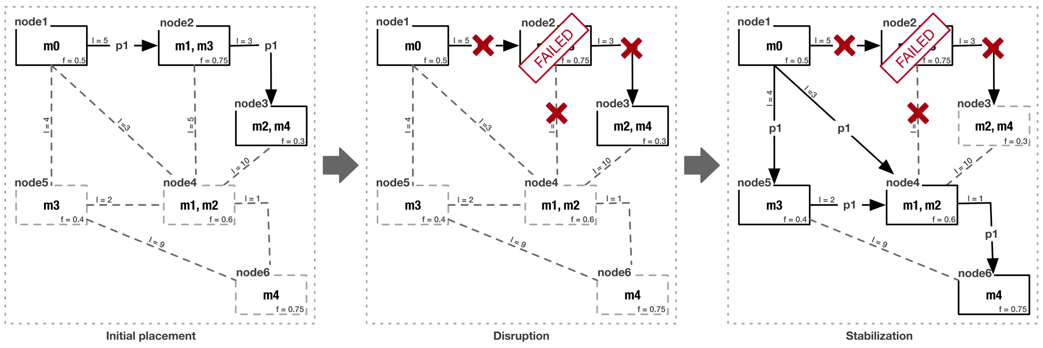

Fig. 3. Three distinct stages in which the edge system may be found: (i) *initial placement*, the application becomes operational; (ii) *disruption*, a node failure renders the invocation path non-viable; and (iii) *stabilization*, the system adapts by finding a new path.

able invocation path among the already allocated microservices such that the latency requirements are met. The *invocation path cycle* chooses one path from the many possible paths. This stage yields a *stable system*, where all nodes and placed microservices are operational and an invocation path is derived. In Figure 3, the dashed line represents the communication path between two nodes, whereas f shows the failure probability of a node, and l is the latency associated to a certain communication path. Finally, *p1* represents the invocation path chosen by the *invocation path cycle*.

- *Disruption*: Given a stable and running system, a disruption occurs due to the failure of one or more nodes resulting in an unresponsive application since microservices nodes previously hosted cannot be reached. For example, assume that during execution of the public-safety application, *node 2* where m_1 and m_3 reside has failed. The previous invocation path is no longer viable and a new one must be devised.
- *Stabilization*: After a new path is calculated, the application's operation is restored. Note how this new stable state may be fragile; if more nodes fail, the invocation path planning activity may fail. If a path cannot be calculated (e.g., because multiple nodes hosting critical microservices disappeared), the system is unable to return to a stable state and a placement must be triggered again.

4 APPLICATION AND SYSTEM MODELS

In this section, we outline the application and system models upon which the proposed framework is based. We start by defining the target edge system model; next, we present the application model along with its defining components, such as microservices and execution sequence. Finally, we introduce the framework objectives—latency and availability.

4.1 Edge System Model

The target edge system consists of interconnected nodes distributed at different locations. It is assumed that there is a central node (i.e., a *coordinator node*) that governs over all other nodes. Let E_N represent the total number of devices found under the supervision of a coordinator node (i.e., $E_N = \{E_1, E_2, \ldots\}$). The coordinator is meant to reside in a powerful device like an edge server or gateway. Edge nodes within the scope of the coordinator may have different capabilities, ranging from resource-constrained single-board computers to server-class data center hosts. Each edge node has a *failure probability* representing its likelihood to leave the network or fail during the application's life span. The failure probability can be estimated at runtime using historic data of each type of edge device to predict the failure probability of a new node [29]. Besides the failure probability, a set of available resources $\mathsf{E}_{res} = \{r_1, r_2, \ldots\}$ denotes the node's current capabilities. Those may range from typical computation, memory or storage to specialized hardware (e.g., GPUs/TPUs for machine learning workloads) and sensing/actuation features.

4.2 Application Model

In our conception, the (developer-provided) application model follows a microservice-based architecture where its functionality is divided into multiple linked microservices, each being characterized by a set of requirements. As such, a model is defined as (i) a set of microservices $\mathsf{M}_N = \{m_1, m_2, \ldots\}$, (ii) the execution sequence, and (iii) a set of functional and resource requirements. The functional requirements are in terms of latency and availability, whereas the resource requirements reflect what each microservice needs to function properly on a hosting node.

A microservice m_i performs one part of the application's functionality. Each is defined at design time, by a set of resource requirements $\mathsf{M}_{res} = \{res_1, res_2, res_3\}$. To obtain a successful deployment, at least one node must be able to fulfill the requirements for each microservice. Each m_i can have multiple replicas $Rp = \{rp_1, rp_2, \ldots\}$, intended to guarantee fulfillment of availability. The number of replicas for each microservice is dependent on the application's availability requirements and nodes' failure probabilities. Replication increases the application's resilience to disruption, enabling the coordinator to stabilize the application in the event of node failure. Recall that the business logic entails an order in which microservices are invoked. This execution sequence starts with a source and ends with a sink. The total e2e latency of an invocation path is key—this is the time required for the network to traverse the path starting from the source microservice and ending with the sink microservice (barring any computation, which is application dependent). A communication link between two nodes E_i and E_j thus has an associated latency l_{E_i,E_j}. This is assumed to be monitored at runtime (i.e., through the monitoring activities of the MAPE loop of Figure 1). Latency of a node with another is inherited by all the microservices m_i and m_j they host. Finally, similar to latency, each microservice has an availability inherited from their host.

5 MONITORING AND EXECUTION ACTIVITIES

The *monitoring and execution* activities guarantee that any devised strategy considers the up-to-date information retrieved from the target edge system and is capable of enacting the required changes. Consequently, both the placement and the invocation path cycles share the two activities.

5.1 Monitoring Activity

Considering the uncertainty found in the target edge system—where nodes may fail or leave the system—the monitoring activity is crucial for the two MAPE cycles. The overall objective is to provide a global view by providing information about (i) overall system changes, (ii) edge nodes, and (iii) communication links:

- *System and node monitoring*: To be effective in determining when a change occurs, the activity detects system changes by keeping as a reference the last known stable state—the state when the application was successfully operating within desired parameters. A change appears when a microservice used in the current invocation path is no longer active due to node failure or the arrival of new nodes. The former has a disruptive effect on the system's stability that requires the triggering of the invocation path cycle or, in the worst case, the placement cycle before it. The latter is not a adverse change; new nodes increase the resources available. At the node level, the coordinator monitors nodes' available resources every time the placement cycle is to be invoked.

- *Communication links monitoring*: Since a primary objective is to satisfy latency, monitoring the communication links between nodes is crucial. Not only do these links provide the communication latency between dependent microservices, but they are part of the invocation path alongside nodes. However, links may fail, and when this occurs, the coordinator must invoke the invocation path cycle; we assume that a link fails if either the source or the destination node is unresponsive.

Consider the running example; first, monitoring recognizes the appearance of new nodes when participants arrive at the event, obtains nodes' available resources, and measures their communication links. After the public-safety application is operational, the activity monitors in particular the nodes that participate in the current path, triggering the corresponding cycle when one or more become unresponsive.

5.2 Execution Activity

The last conceptual step in the MAPE cycle is the execution activity. When either of the two cycles finds a successful plan, the execution activity is responsible for enacting it upon the edge system—after its conclusion, the application becomes operational.

Initially, based on the placement plan conceived during *placement planning*, the execution activity distributes microservices on nodes. In this case, execution works above the node level, by sending microservice container images to nodes, where their execution is assumed to be managed by some container framework such as Kubernetes or OpenFaaS. Once all microservices are available, execution enacts the newly generated path by informing each node about each of its microservices' destination (i.e., the node where dependent microservices reside). As such, there is no need for costly migration; instead, execution locates the new nodes that participate in the path and informs them about addresses used for communication. For an example of this, consider executing a path found for the public-safety application (Figure 3). First, during *initial placement*, the cycle finds p_1 as the satisfying path. In this case, to execute the plan, execution finds the addresses of nodes E_1, E_2, and E_3. Next, it informs E_1 that m_0 communicates with its dependent m_1, using the address of E_2. Notice that if two dependents share the same node as in the case of m_1 and m_3, the communication is internal. Finally, the activity informs E_2 about E_3.

6 PLACEMENT CYCLE

The *placement cycle* ensures that the edge system satisfies the application's requirements in terms of availability and microservices' resource requirements. The cycle is bootstrapped with a given application-wide availability requirement, an application model, and the target edge system—those are specified per application. Note how the cycle is not concerned with invocation path calculation and its associated e2e latency, but merely aims to derive the needed number of replicas to satisfy availability, and subsequently map them. We distill the following objectives:

- What is the minimum number of microservice replicas required to satisfy the application's availability requirement?
- Given the application model and the number of replicas, what placement strategy satisfies the microservices' resource requirements?

6.1 Placement Encoding Activity

The purpose of the *placement encoding activity* is to find a satisfiable mapping considering the information received from the monitoring activity: the edge system, the nodes' failure probabilities, and the intended availability. Computing an optimal placement is not necessary (due to the increased computation this would require), since the network configuration may change frequently and the procedure may need to be invoked frequently as well. Instead, we seek a placement that satisfies the given requirements with respect to the target edge system. To this end, we adopt SMT [5]; we model placement as first-order formulas consisting of constraints and an objective. Moreover, computational demands are kept low—more on this will be discussed in Section 8. Specifically, we model the problem with the linear integer arithmetic and Boolean theories, having three distinct targets: (i) placing microservices on nodes, (ii) placing microservice replicas on nodes, and (iii) keeping track of availabilities of microservice on their hosting nodes.

Microservice placement. The first constraint ensures correct placement. As a rule, upon deployment, a microservice m_i can be placed on a node E_i if and only if (i) E_i is capable of satisfying the resource requirements of m_i and (ii) m_i does not exist in the system yet. Based on these rules, we can deduct that *microservice placement* does not consider replicas of m_i, replicas being handled by another constraint. Consider an edge system of two nodes E_1 and E_2 on which mapping of microservice m_1 (*face recognition*) from the public-safety application is desired. There are two possible mappings of m_1—behavior where the m_1 is placed on both nodes must be avoided. This is captured with Formula 1, where n_m represents the total number of microservices found in the application model, whereas the *map()* function provides a mapping between m_i and a node E. For number of nodes n and microservices m, the formula construction exhibits complexity $O(nm)$.

$$\mathsf{mrcFacts}: \bigwedge_{i=1}^{n_m} (\exists!\, E : \mathsf{map}(m_i = E)),\ \forall E \in E_N \tag{1}$$

Microservice replication. Replication of a microservice m_i is considered separately. First, only one replica rp_i can be placed on a node E_i; placing replicas on different nodes aims at higher availability. Second, as in the case of the placement constraint, E_i must have the required resources to execute rp_i. For example, consider one more replica of the *face recognition* m_1 is desired. In this case, its replica rp_1 must be deployed on a system consisting of three nodes E_1, E_2, and E_3. Following Formula 1, m_1 is mapped on E_2. rp_1 cannot be mapped on the same node as m_1, and only two possible nodes are available when deploying $rp_1 - E_1$ and E_3. More concretely, Formula 2 ensures that if a replica is mapped on a certain node, then all other remaining replicas cannot be placed on that node, where n_r represents the total number of replicas for a certain microservice and n_{rl} the total number of replicas without considering the current rp_i. Construction of Formula 2 exhibits complexity of $O(nn_r)$, where n is the total number of possible locations where replicas of m_i can reside.

$$\mathsf{replDomain}: \bigwedge_{i=1}^{n_r} ((\mathsf{map}(rp_i = E)) \implies \bigvee_{j=i}^{n_{rl}} (!\mathsf{map}(rp_j = E)),\ \forall E \in E_N \tag{2}$$

Microservice availability. A microservice m_i inherits availability from its host node, which has a known failure probability. Knowing the availability of each microservice aids in determination of the maximum number of replicas required and the decision of if a certain placement configuration fulfills the requirements. For example, if m_1 is mapped on E_1 where E_1 has a failure probability of *40%*, then the availability factor of m_1 is $m_{avail_i} = 1 - E_{failure}$. Formula 3 encodes this based on the deployment destination node; its construction complexity is analogous to the previous formula.

$$\mathsf{availDomain}: \bigwedge_{i=1}^{n_r} (\mathsf{map}(m_i = E) \implies m_{avail_i} = 1 - E_{failure}),\ \forall E \in E_N \tag{3}$$

Placement objective. The number of replicas is dependent on the application's availability requirement. As a result, we define an *objective constraint* to ensure that placement satisfies the application's availability. Microservices' availability are dependent on the number of replicas and their inherited availability. Keeping their availability equal to the application's availability aims at having enough resources to make the application more resistant to node failures; the availability of a microservice helps to distribute multiple replicas on different edge nodes. This is captured in Formula 4, where $\mathsf{t}_{\mathsf{maxAvail}}$ is the desired microservice availability ; its construction amounts to complexity of $O(n_r)$.

$$\mathsf{objConstraint} : 1 - \prod_{\mathsf{i}=1}^{\mathsf{n_r}} \mathsf{t}_{\mathsf{avail_i}} \geq t_{maxAvail} \tag{4}$$

6.2 Placement Planning Activity

This activity employs the encoding of the previous step to generate a satisfiable placement plan. By virtue of the design choice of employing SMT, we guarantee that the generated plan provably satisfies all the defined constraints and fulfills the placement objective. At the core of the *placement planning activity* is solving a formula $\mathcal{F}$ capable of deciding if a strategy satisfies the cycle's objectives or not. Formula 5 is a conjunction of Formulas 1 through 4, received from the previous activity. Once the activity knows $\mathcal{F}$, a plan is devised by employing an SMT solver—solutions obtained are used to deduce the plan.

$$\mathcal{F} : \mathsf{mrcFacts} \ \wedge \mathsf{replDomain} \ \wedge \ \mathsf{availDomain} \wedge \mathsf{objConstraint} \tag{5}$$

In essence, the activity calculates the minimum number of replicas required by each microservice to meet the availability requirements. To do this and find a placement strategy, the *placement planning activity* uses $\mathcal{F}$ to place one microservice $\mathsf{m_i}$ at a time in an iterative process, where at each step a new replica of $\mathsf{m_i}$ is placed in the system until the availability of $\mathsf{m_i}$ is satisfied. Once $\mathsf{m_i}$ is placed successfully in the system, the process continues with the next microservice $\mathsf{m_j}$. The process terminates upon two conditions: (i) all microservices are placed in the system or (ii) one or more microservices cannot be placed. In the former case, since there are enough resources in the system to accommodate all applications' microservices, the cycle is able to find a plan. In the latter, the target system lacks the required resources to host all microservices; a plan does not exist.

For an example of the entire process, consider the activity placing the running example application on an edge system consisting of six nodes. First, the activity deduces $\mathcal{F}$; the process to find the minimum number of replicas and the placement strategy can commence. We start by mapping one microservice at a time, placing its required replicas as well. For example, consider the process of placing $\mathsf{m_1}$. At this stage, the activity already found a suitable placement for $\mathsf{m_0}$. Hence, on the system, there are fewer available resources when placing $\mathsf{m_1}$. Based on this information, a replica of $\mathsf{m_1}$ is placed in the system and an evaluation occurs, to check if the current placement of $\mathsf{m_1}$ is enough to satisfy Formula 4. If placing only one replica of $\mathsf{m_1}$ is not enough, then replicas for $\mathsf{m_1}$ are increased and a solution of $\mathcal{F}$ is attempted again. We continue to increase the number of replicas until either Formula 4 is satisfied or there are not enough available resources to accommodate all replicas. Finally, if we manage to find and place the minimum number of replicas for $\mathsf{m_1}$, the planning process continues with $\mathsf{m_2}$. However, if no solution is found for $\mathsf{m_1}$, then the process stops without producing any placement strategy. Figure 4 shows a placement strategy found for the public-safety application.

7 INVOCATION PATH CYCLE

The objective of the placement cycle was to deploy the application's microservices across available nodes comprising the system. After the cycle has been completed, they may be invoked according to the application's business logic. Due to replication, however, multiple invocation paths are possible. The invocation path cycle is responsible for establishing (and maintaining) the call se-

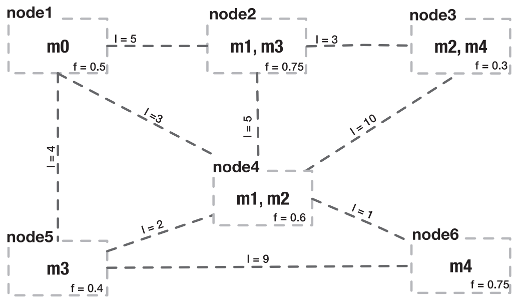

Fig. 4. Placement of the example, showing locations of microservices and their replicas across nodes.

quence across those replicated microservices such that the application's latency requirements are met. Disruptions such as network changes or node failures may render a path no longer usable; thereupon, the cycle is triggered.

7.1 Invocation Path Encoding Activity

Considering the invocation cycle's objectives and information received from monitoring, three different formulas can be constructed—two that reflect the cycle's constraints and one that considers its objectives; they capture (i) the location of microservices and (ii) the communication latency between nodes.

Microservice location. The location of microservices plays an important role in finding a satisfiable path; the domain for each needs to be defined, offering a set of nodes where it resides. This domain is an asset to the cycle when searching for a satisfiable path: first, by knowing the location of each $\mathsf{m_i}$, the search space is constrained by considering a subgroup of nodes—a group formed with nodes from the combined microservices' domains. For example, consider that there are nodes $\mathsf{E_1}$, $\mathsf{E_2}$, and $\mathsf{E_3}$, where the *face recognition* microservice $\mathsf{m_1}$ exists. Under these conditions, the invocation path cycle considers one of the three when building the path. This is illustrated in Formula 6, where $\mathsf{n_M}$ represents the total number of microservices and $\mathsf{n_E}$ is the total number of nodes where $\mathsf{m_i}$ is mapped; its construction exhibits complexity of $O(n_E n_M)$.

$$\mathsf{microDomain}: \bigwedge_{i=1}^{n_M} \left(\bigvee_{j=1}^{n_E} (m_i = E_j) \right) \tag{6}$$

Microservice latency. Recall that microservices inherit latency from their hosting nodes; a pair $\mathsf{m_i, m_j}$ may have a different communication latency on different combinations of nodes. Given Formula 6, one can observe the latency between two microservices $\mathsf{l_{m_i,m_j}}$, considering their possible locations. For example, for the public-safety application (Figure 2), $\mathsf{m_1}$ has a dependency with $\mathsf{m_2}$; $\mathsf{m_1}$ and its replicas are mapped on $\mathsf{E_1}$, $\mathsf{E_2}$, and $\mathsf{E_3}$, whereas $\mathsf{m_2}$ is on $\mathsf{E_2}$. Under these conditions, the encoding consists of all possible combinations between the two microservices' domains. In the end, the constraint yields a connection between the location of a microservice and its associated node latency (i.e., $\mathsf{l_{m_i,m_j} = l_{E_i,E_j}}$)—captured in Formula 7. Its construction exhibits complexity of $O(n_{MD} n_{ES} n_{ED})$, where n_{MD} is the total number of dependent groups consisting of two microservices (a source and a destination), whereas n_{ES} and n_{ED} represent the total number of nodes where microservices part of a dependent group exist.

$$\begin{aligned} \mathsf{latencyDomain}: \bigwedge_{D}^{n_m} (m_i = E_i,\ m_j = E_j) \Rightarrow (l_{m_i,m_j} = l_{E_i,E_j}) \text{for } D = \{i, j\}, \\ \text{where } i \in [0,\ n_m] \text{ and } j \in [0,\ n_{EN}]. \end{aligned} \tag{7}$$

Invocation path cycle objectives. There are two objectives the cycle must ensure: e2e latency and availability. The latter can be obtained by Formula 3, where $\mathsf{n_r}$ represents all available microservices and their replicas. The former represents the sum of all communication latencies between each of two dependent microservices. Recall that we assume that the maximum e2e latency and the desired availability are given at design time; to capture the e2e latency of a path, the invocation path cycle must perform three steps: (i) choose an invocation path, (ii) obtain the latency between dependent microservices, and (iii) verify that the total e2e latency is fulfilled. The cycle can perform the first two steps using the placement and the latency constraints. For the third step, Formula 8 is used to check if the chosen path fulfills the objective. The formula ensures that the path's e2e latency is smaller than or equal to the maximum e2e latency allowed $\mathsf{e2e_{max}}$. Construction of Formula 8 exhibits complexity $O(mn_{md})$, where m represents the number of available microservices and n_{md} the number of dependencies a microservice has.

$$\mathsf{objConstraint}: \sum_{\mathsf{i=1}}^{\mathsf{m}} \mathsf{l_i} \leq \mathsf{e2e_{max}} \tag{8}$$

7.2 Invocation Path Planning Activity

Similar to placement, this activity creates a plan for a new path. Formula $\mathcal{F_I}$ builds upon the formulas received from the invocation path encoding activity. By casting this activity within SMT, we guarantee that every path found fulfills the cycle's objectives. Finally, to obtain the path plan, $\mathcal{F_I}$ is solved by employing an SMT solver.

$$\mathcal{F_I}: \mathsf{microDomain} \wedge \mathsf{latencyDomain} \wedge \mathsf{objConstraint} \tag{9}$$

In contrast to the iterative process employed by placement planning, this activity considers all microservices in one step. After obtaining $\mathcal{F_I}$, planning takes the application's execution sequence and attempts to find a series of nodes (ones that host microservices invoked in the execution sequence) such that the cycle's objectives are fulfilled. In comparison to the formula employed by the placement cycle, observe that $\mathcal{F_I}$ has (i) a smaller formula size since it considers only a subgroup of nodes, (ii) awareness of microservice placement, and (iii) existing information about latency. In essence, $\mathcal{F_I}$ must find a combination of nodes that fulfill Formula 8. As a result, $\mathcal{F_I}$ is expected to scale well with an increase in network or application size—more on this topic will be discussed in Section 8.

Figure 5 illustrates a produced plan for the public-safety application. Path planning starts from the placement cycle; the activity obtains the application's execution sequence and attempts to find a set of nodes where microservices reside to build a path. There may be multiple invocation paths in the system, but not all of them satisfy Formula 8. Finally, the activity returns a satisfiable path (e.g., p_1).

8 EVALUATION

To concretely support evaluation and investigate feasibility of the proposed framework, we realized a proof-of-concept implementation that is available as open source software reflecting the cycles of Sections 6 and 7, along with auxiliary functionalities supporting monitoring and execution. Thereupon, we assess performance and scalability of the critical placement and invocation path cycles. We additionally investigate the framework's capability to reach stabilization when nodes fail and conclude with a discussion.

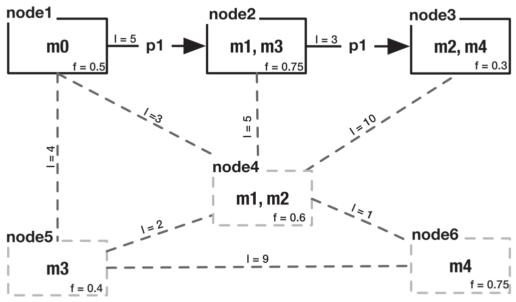

Fig. 5. Satisfiable path p_1 found for the public-safety example.

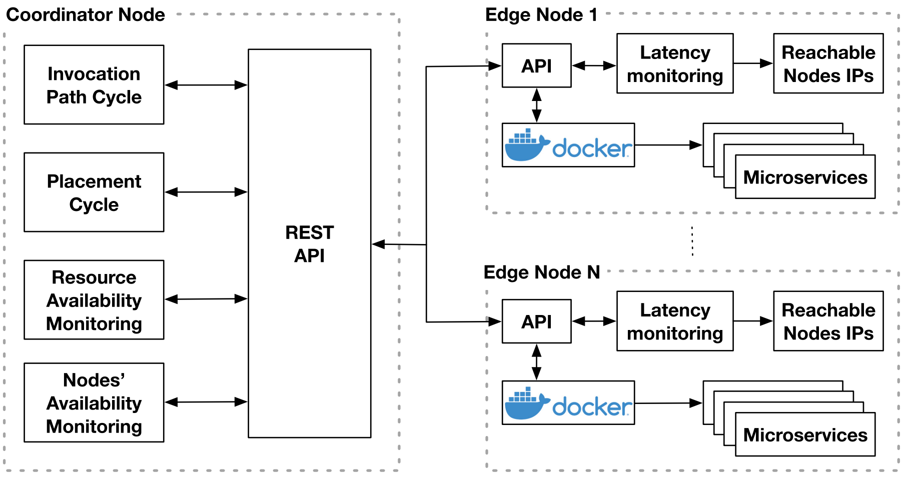

Fig. 6. Combined deployment and dataflow diagram for a framework realization.

8.1 Framework Realization

To investigate feasibility, we realized the proposed framework as a prototype implementation; its main architectural components are illustrated in Figure 6. The coordinator node hosts the placement and invocation cycles, which construct the required formulas and through interaction with a solver build the corresponding plans. Monitoring functionality is responsible for detecting the status of nodes in the network, their available resources as well as obtaining latencies between them. Execution entails deployment of containers on nodes and setting up communication between deployed microservices. Edge nodes host Docker (or alternatively Kubernetes/OpenFaaS) and report their latencies to others to the coordinator. All interactions are performed through REST APIs and use Ansible for infrastructure-as-code. The prototype built in Python using Z3 [9] as the underlying SMT solver and an example microservice application are available.[3]

8.2 Experiments Setup

In the following, we evaluate the performance of our proposed solution considering different scenarios. For each, we first assess the performance of the placement cycle; then we employ the invocation path cycle to find valid paths. Each scenario has different characteristics to evaluate the performance in terms of the time required to find a valid solution. Therefore, we gradually

[3]https://github.com/cavasalcai/Adaptive-Volatile-Edge-Systems.

Table 1. Characteristics of the Synthesized Scenarios Adopted for Performance Evaluation

Scenario	Micro–services	e2e Latency	Availability	Failure Probability	Nodes
1	10	350	0.75	[0.1, 0.5]	10–500
2	20				
3	30				
4	40				
5	50				

increase the number of nodes of the target system—Table 1 shows the five scenarios along with their specifications. Each has a different number of microservices. This intends to examine the impact of the system size as the total number of nodes on execution time. The e2e latency and availability are kept unchanged since (i) we are interested in understanding how the number of nodes influences the placement execution time, and (ii) choosing more stringent constraints (i.e., having a lower e2e latency or a higher availability) will have an impact on the execution time; however, this does not alter how the execution time rises with an increase in network size. Finally, the resource requirements of each microservice are a set of resources as a tuple *(RAM, CPU, HDD)*; for each, we randomly choose a value between [5, 18] units. Experiments are performed on a machine with an Intel i5 2.3-GHz processor and 16 GB of RAM.

Recall that the system is characterized by (i) the number of nodes, (ii) their failure probabilities, and (iii) the communication latency associated with each link. We set the failure probability of each node and latency by choosing a random value between [0.1, 0.5] and [1, 10] ms, respectively. Regarding resources, we assign a random value chosen from [15, 30] units. Finally, for each scenario, we increase the number of nodes by 10 up to a maximum of 500. Furthermore, when we increase the system size, we add new nodes on top of the current network. Thus, a progressively larger system always contains the previous nodes, simulating a case where new nodes appear.

Finally, for each application size, we choose an execution sequence that starts with a source and ends with a sink. For this reason, we create a sequence that involves all microservices and leads to a different sequence for each application size. For example, consider building the execution sequence for an application with 10 microservices, $M_N = \{m_1, m_2, m_3, \ldots, m_{10}\}$. The procedure starts by choosing m_1 as the initial source. Next, we remove m_1 from M_N and randomly choose a destination microservice from the remaining elements in M_N. Assume that m_3 is chosen as the destination of m_1—there is a dependency between m_1 and m_3. Notice that we do not remove m_3 from M_N; we remove m_3 only when it becomes the source. After a destination for m_1 is found, the procedure is restarted by taking the next microservice from M_N (i.e., m_2). Finally, the procedure stops when M_N is empty.

8.3 Placement and Invocation Path Performance

We consider that the placement cycle consists of two phases: (i) finding a placement strategy and (ii) starting the microservice on the host node. To evaluate the placement cycle's performance, we measure the time required to find a satisfiable placement strategy according to the application's objectives. We perform 50 placements per scenario, one placement for each new network size. In this case, once we find a placement strategy for the current network, we increase the network size by 10 and attempt to find a new strategy. Figure 7 illustrates the execution time required to find a placement strategy for all scenarios. The *x-axis* shows the number of nodes, whereas the *y-axis* presents the execution time in seconds. Subsequently, the invocation path cycle requires as input

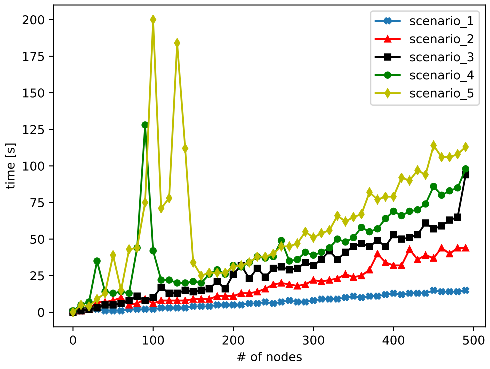

Fig. 7. Execution time of the placement cycle for different scenarios across different network sizes.

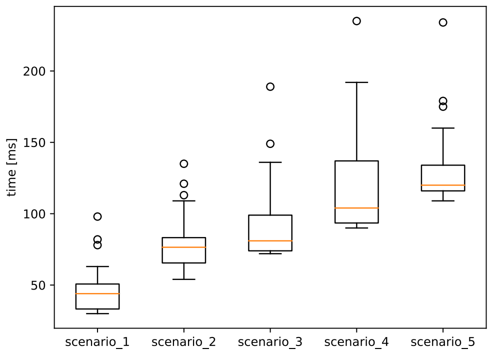

Fig. 8. Average execution time required by the invocation path cycle across the five scenarios.

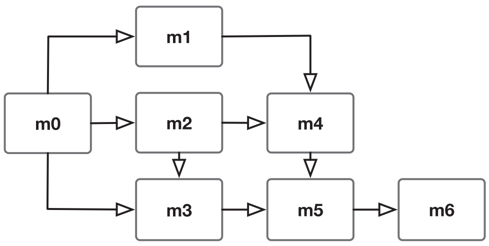

Fig. 9. Application model and its execution sequence.

Table 2. Initial Placement and Invocation Paths

Micro-services	**Nodes**	**Initial Path**
m0	E_6, E_{16}, E_{49}, E_{50}, E_1, E_{43}	E_6
m1	E_6, E_{16}, E_{49}, E_{50}, E_1, E_{43}	E_6
m2	E_6, E_{16}, E_{49}, E_{50}, E_1, E_{43}	E_6
m3	E_{39}, E_{17}, E_{33}, E_{48}, E_{32}, E_4	E_{33}
m4	E_4, E_{32}, E_{44}, E_6, E_2, E_{39}	E_4
m5	E_{25}, E_{45}, E_2, E_4, E_{33}, E_{50}	E_{25}
m6	E_{17}, E_2, E_4, E_{32}, E_{33}, E_{43}	E_{17}

the location of microservices and their replicas on the system, which is the result of placement. Similar to placement, we measure the invocation path cycle's performance as the time required to derive a feasible path considering the application's latency requirement. We evaluate the cycle's performance on obtaining satisfiable invocation paths on 250 different network topologies, 50 for each scenario, when there was no failure in the system. Figure 8 illustrates the time required to find a satisfiable invocation path among all replicas.

8.4 Stabilization Performance

In the following, we investigate the framework's capabilities to reach stabilization when one or more nodes have failed. This occurs when the system recovers from node failure, by employing the *invocation path cycle*; we focus on evaluating the performance of the cycle after the system has lost nodes. We consider an application of 50 edge nodes. The application model is a slightly modified version of the one presented in the work of Rossi et al. [21] and illustrated in Figure 9. It consists of seven microservices—we further add a new dependency between m_4 and m_5; it has one source (m_0) and one sink (m_6). We adopt a maximum e2e latency of 100 time units and a required availability of 0.96. All other characteristics, like resource requirements and latency, remain the same as described for the other scenarios. However, to evaluate stabilization, we adopt a different set of characteristics for edge nodes, aiming for a higher number of replicas in the system: we randomly select node failure probabilities between [0.1, 0.5] and set available resources in the range of [50, 80] units, respectively.

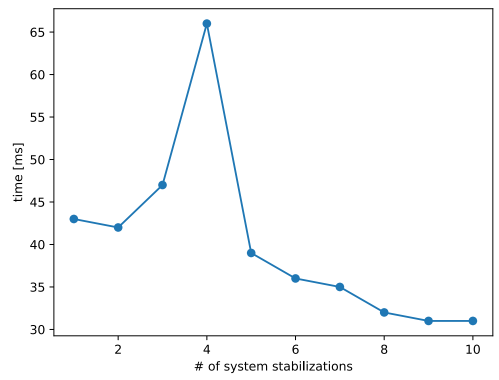

Fig. 10. Execution time required by the invocation path cycle to reach stabilization after node failures.

Table 3. Invocation Paths and Their e2e Latency Found When Incrementally Failing Nodes

#	Microservice Invocation Path	e2e Latency	Node Failure
1	$E_{17}, E_4, E_{25}, E_{33}, E_6$	32	E_4
2	$E_{32}, E_{17}, E_{25}, E_{33}, E_6$	41	E_{25}
3	$E_{32}, E_{17}, E_{45}, E_{33}, E_6$	39	E_6
4	$E_{32}, E_{17}, E_{45}, E_{16}, E_{33}$	31	E_{32}
5	$E_{17}, E_{44}, E_{45}, E_{16}, E_{33}$	42	E_{44}
6	$E_{45}, E_{17}, E_2, E_{16}, E_{33}$	42	E_{17}
7	$E_2, E_{45}, E_{39}, E_{16}, E_{33}$	46	E_{33}
8	$E_{45}, E_{39}, E_{16}, E_2$	45	E_{45}
9	$E_{49}, E_{39}, E_{16}, E_2$	46	E_{16}
10	$E_{49}, E_{39}, E_2, E_{50}$	41	E_{50}
11	E_{49}, E_2, E_{48}	30	E_2

Initially, microservices and the invocation path are missing from the target system: a placement and derivation of a path are first performed. Table 2 records the starting placement configuration and the chosen path. Each row then shows all nodes where a microservice resides and the node that participates in the respective path. Recall that after the initial deployment, nodes may fail, leading to the need to start the stabilization process. However, not all failed nodes from the system disrupt the application's functionality: only ones that are part of the path. As such, we incrementally fail a node from the current path and use the cycle to recover. Table 3 records each path found during stabilization as well as its associated e2e latency. For example, in the first row, the current path (from Table 2) has a total e2e latency of *32* units. However, at some point, the edge node E_4 fails, disrupting the path. As a result, a new path is obtained and presented on the next row. This process continues until there is no possibility of system stabilization with the remaining microservices across the system, resulting in the need of employing the placement cycle again to repopulate nodes. Performance results are shown in Figure 10.

8.5 Discussion

We believe that we have demonstrated that by virtue of the framework's two cycles, application placement and its recovery from an unstable state resulting from volatility can be performed timely. Results of Figure 7 illustrate that (i) the time required to find a placement strategy increases with the network size and that (ii) the application size impacts the execution time as well. Among the two, the application's size has a greater impact on the execution time, since one microservice and its replicas are deployed at a time, a step that requires to invoke the solver for each. The execution time is influenced by other factors as well, such as applications' requirements (e2e latency, microservice's resource requirements, and availability) and network characteristics (failure probabilities and available resources). However, these factors do not alter the increasing trend observed in Figure 7, as it only impacts the time required to find a solution for specific scenarios; it can increase or decrease the execution time depending on how stringent the requirements are. We can observe the impact of these factors at the spikes found at lower network sizes. In these particular cases, the network topology size, the node's failure probabilities, and its available resources play an important role: (i) either there is no feasible solution due to the lack of available resources in the target network or (ii) available resources are equal to the application's requirements. In both situations, the solver must traverse the entire search space to try and find a placement strategy. We can observe that with the increase in the network size, there are more available resources, offering the solver more possibilities to find a satisfiable placement, lowering the execution time.

Figure 8 shows the execution time required to find a path for different cases. Compared to placement, the invocation path cycle is capable of finding a solution faster. We can observe that there are no big spikes in execution times—this behavior is attributed to the extra knowledge that the solver has as the encoding is "aware" of the location of all microservices and replicas on the network. Hence, independent of the network size, the solver only attempts to find a path between nodes where the application is deployed. As a consequence, we can conclude that this cycle is influenced only by the total number of microservices and their placement.

Figure 10 shows the proposed framework's capabilities to restore functionality after node failures. In this scenario, we make use of specific requirements—availability and node's failure probabilities—to increase the number of microservice replicas found. As a consequence, the failure probabilities of all nodes are set under *0.5* and a high availability of *0.95* is chosen. This practice forces the placement cycle to place replicas on more nodes. Another factor that influences the number of replicas is the network size. In a network with many nodes (e.g., 500), there are more chances to find nodes with available resources and low failure probability to deploy a small-sized application. Thus, placement is capable of being satisfied with fewer replicas. However, when the size of the network is limited to 50 and stringent requirements are set, a solution would require increasing the number of replicas. Under these conditions, the placement cycle requires more time to find a satisfiable deployment, whereas the invocation path cycle remains unchanged. We can observe that in the experiments performed the invocation path cycle is capable of recovering from an unstable state in under 70 ms. Furthermore, since the cycle only considers nodes where microservices are placed (see Table 3), the required time to find a path decreases with every node failure.

Regarding the overall process, observe that adaptation entails changing the path between replicated microservices instead of migrating them to other hosts. This is because a migration typically requires higher communication overhead as a microservice must be moved from one node to another. For example, a 25-MB container would take 25 seconds to migrate from one node to another, assuming an 1-MB/s link. In comparison, our approach targets recovery with small overhead, since no migrations are involved; naturally, there is a trade-off between communication overhead and the increased redundancy due to replication. Depending on the application, the technique advocated may require significantly more available resources to host replicas. However, this can be mitigated by considering microservice boot/cold starts. On the one hand, if fast reactions or stabilization is desired, the replicas can be kept warm, a practice that requires more available resources. On the other hand, if conservation of resources is desired, replicas can be kept cold and latency introduced by cold starts can be considered in the e2e latency computation.

The maximum number of stabilizations employing singularly the invocation path cycle is dependent on multiple factors. First, the distribution of microservices on hosting nodes plays the most crucial role. Stabilization is not possible if a microservice does not have any active replicas in the network. The application's requirements represent a second factor that impacts stabilization success; in this case, available replicas are still found, but the cycle is unable to find a new path. Finally, the number of failed nodes in the current path lowers the number of possible stabilizations; losing more nodes leads to one of the two factors previously discussed. If there is at least one available replica for each microservice and both the e2e latency and availability are fulfilled, stabilization is, however, guaranteed. Finally, we note that the proposed framework is not bound to a specific edge network topology or density and assumes that all nodes are reachable. Density may hinder the *invocation path cycle's* ability to find a paths if edge nodes are scattered in a large area, increasing links' communication latency. Finally, we acknowledge the high computational demands of the placement cycle in some scenarios (referring to the spike of Figure 7). However, since placement is performed only at the beginning of the application's life span and sporadically during its execution, we argue that it does not impose significant delays in practical settings. Moreover, as mentioned earlier, placement performance depends on the target system and how stringent requirements are. Placement is not used for recovery from an unstable state, since for such cases the other cycle is employed for adapting to volatility; the invocation path cycle is capable of recovering an application rather timely (under 100 ms for sizes considered in Section 8).

9 CONCLUSION AND FUTURE WORK

In this article, we considered that applications execute on distributed edge systems where node failure is a prime concern; devices may leave the network or fail without prior notice. As such, the application's stable state needs to be maintained throughout its execution. We proposed an adaptive framework consisting of two MAPE cycles: the placement and invocation path cycles. The former aims at devising a placement to provide the required resources for a microservice-based application. Furthermore, the placement cycle facilitates availability by replicating microservices throughout the system. The latter cycle provides fast recovery from unstable states by building satisfying invocation paths across deployed microservices and their replicas.

Regarding future work, we intend to extend the adaptive framework presented to deploy applications with multiple invocation paths (i.e., more than one source and sink microservices). Furthermore, we aim to develop functionality capable of deducing nodes' failure probabilities at runtime, also since their availability may change over time; something that is closely related to monitoring in edge systems [26]. Finally, integration with management frameworks such as OpenFaas or Kubernetes is another aspect that should be further tackled, as well as capturing other requirements pertinent to the edge-based systems that we target, such as privacy [25].

REFERENCES

[1] Danilo Ardagna and Li Zhang. 2010. *Run-time Models for Self-Managing Systems and Applications*. Springer Science & Business Media.

[2] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. 2019. Decentralized resource auctioning for latency-sensitive edge computing. In *Proceedings of the IEEE International Conference on Edge Computing (EDGE'19)*.

[3] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. 2021. Resource management for latency-sensitive IoT applications with satisfiability. *IEEE Transactions on Services Computing*. Early access, April 20, 2021.

[4] Marios Avgeris, Dimitrios Dechouniotis, Nikolaos Athanasopoulos, and Symeon Papavassiliou. 2019. Adaptive resource allocation for computation offloading: A control-theoretic approach. *ACM Transactions on Internet Technology* 19, 2 (April 209), Article 23, 20 pages.

[5] Clark Barrett and Cesare Tinelli. 2018. *Satisfiability Modulo Theories*. Springer International Publishing, Cham, Switzerland, 305–343.

[6] Antonio Brogi and Stefano Forti. 2017. QoS-aware deployment of IoT applications through the fog. *IEEE Internet of Things Journal* 4, 5 (2017), 1185–1192.

[7] J. Chen, S. Chen, Q. Wang, B. Cao, G. Feng, and J. Hu. 2019. iRAF: A deep reinforcement learning approach for collaborative mobile edge computing IoT networks. *IEEE Internet of Things Journal* 6, 4 (2019), 7011–7024.

[8] Nader Daneshfar, Nikolaos Pappas, Valentin Polishchuk, and Vangelis Angelakis. 2018. Service allocation in a mobile fog infrastructure under availability and QoS constraints. In *Proceedings of the 2018 IEEE Global Communications Conference (GLOBECOM'18)*. 1–6.

[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

[10] Raphael Eidenbenz, Yvonne-Anne Pignolet, and Alain Ryser. 2020. Latency-aware industrial fog application orchestration with Kubernetes. In *Proceedings of the 2020 5th International Conference on Fog and Mobile Edge Computing (FMEC'20)*. 164–171.

[11] Diogo Goncalves, Karima Velasquez, Marilia Curado, Luiz Bittencourt, and Edmundo Madeira. 2018. Proactive virtual machine migration in fog environments. In *Proceedings of the 2018 IEEE Symposium on Computers and Communications (ISCC'18)*. 00742–00745.

[12] Keerthana Govindaraj, Jibin P. John, Alexander Artemenko, and Andreas Kirstaedter. 2019. Smart resource planning for live migration in edge computing for industrial scenario. In *Proceedings of the 2019 7th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud'19)*. 30–37.

[13] M. Huang, W. Liu, T. Wang, A. Liu, and S. Zhang. 2020. A cloud-MEC collaborative task offloading scheme with service orchestration. *IEEE Internet of Things Journal* 7, 7 (2020), 5792–5805.

[14] Saadallah Kassir, Gustavo de Veciana, Nannan Wang, Xi Wang, and Paparao Palacharla. 2020. Service placement for real-time applications: Rate-adaptation and load-balancing at the network edge. In *Proceedings of the 2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud'20) and the 2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom'20)*. 207–215.

[15] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[16] Isaac Lera, Carlos Guerrero, and Carlos Juiz. 2019. Availability-aware service placement policy in fog computing based on graph partitions. *IEEE Internet of Things Journal* 6, 2 (2019), 3641–3651.

[17] C. Liu, M. Bennis, M. Debbah, and H. V. Poor. 2019. Dynamic task offloading and resource allocation for ultra-reliable low-latency edge computing. *IEEE Transactions on Communications* 67, 6 (2019), 4132–4150.

[18] Redowan Mahmud, Kotagiri Ramamohanarao, and Rajkumar Buyya. 2018. Latency-aware application module management for fog computing environments. *ACM Transactions on Internet Technology* 19, 1 (Nov. 2018), Article 9, 21 pages.

[19] V. De Maio and I. Brandic. 2018. First hop mobile offloading of DAG computations. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 83–92.

[20] Amina Mseddi, Wael Jaafar, Halima Elbiaze, and Wessam Ajib. 2019. Intelligent resource allocation in dynamic fog computing environments. In *Proceedings of the 2019 IEEE 8th International Conference on Cloud Networking (CloudNet'19)*. 1–7.

[21] Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. 2020. Self-adaptive threshold-based policy for microservices elasticity. In *Proceedings of the 2020 IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'20)*.

[22] Deepa R. Sangolli, Nagthej M. Ravindrarao, Priyanka C. Patil, Thrishna Palissery, and Kaikai Liu. 2019. Enabling high availability edge computing platform. In *Proceedings of the 2019 7th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud'19)*. 85–92.

[23] Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. 2017. Optimized IoT service placement in the fog. *Service Oriented Computing and Applications* 11, 4 (Dec. 2017), 427–443.

[24] Klervie Toczé and Simin Nadjm-Tehrani. 2018. A taxonomy for management and optimization of multiple resources in edge computing. arXiv:1801.05610.

[25] Christos Tsigkanos, Cosmin Avasalcai, and Schahram Dustdar. 2019. Architectural considerations for privacy on the edge. *IEEE Internet Computing* 23, 4 (2019), 76–83.

[26] Christos Tsigkanos, Marcello Bersani, Pantelis A. Frangoudis, and Schahram Dustdar. 2021. Edge-based runtime verification for the Internet of Things. *IEEE Transactions on Services Computing* 1 (2021), 1.

[27] Christos Tsigkanos, Stefan Nastic, and Schahram Dustdar. 2019. Towards resilient Internet of Things: Vision, challenges, and research roadmap. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS'19)*.

[28] Cecil Wobker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. 2018. Fogernetes: Deployment and management of fog computing applications. In *Proceedings of the 2018 IEEE/IFIP Network Operations and Management Symposium (NOMS'18)*. 1–7.

[29] Kuang Yuejuan, Luo Zhuojun, and Ouyang Weihao. 2021. Task scheduling algorithm based on reliability perception in cloud computing. *Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering)* 14, 1 (2021), 52–58.

[30] He Zhu and Changcheng Huang. 2018. EdgePlace: Availability-aware placement for chained mobile edge applications. *Transactions on Emerging Telecommunications Technologies* 29, 11 (2018), e3504.

Received October 2020; revised March 2021; accepted June 2021