# A Meta-Pattern for Building QoS-Optimal Mobile Services out of Equivalent Microservices

Zheng Song[1*], Zhengquan Li[1] and Eli Tilevich[2]

[1*]Department of Computer and Information Science, University of Michigan at Dearborn, Dearborn, 48128, Michigan, USA.
[2]Department of Computer Science, Virginia Tech, Blacksburg, 24061, Virginia, USA.

*Corresponding author(s). E-mail(s): zhesong@umich.edu;
Contributing authors: zqli@umich.edu; tilevich@cs.vt.edu;

**Abstract**

A QoS-optimal service balances reliability, execution cost, and latency to satisfy application requirements. In emerging distributed environments, with their unreliable and resource-scarce mobile/IoT devices, it is hard but essential to optimize the QoS of mobile services. Fortunately, these environments are characterized by ever-growing equivalent functionalities that satisfy the same requirements by different means. The combined execution of equivalent microservices has been used to improve QoS (e.g., majority voting for accuracy, speculative parallelism for latency, and failover for reliability). These executions are commonly described as workflow patterns, crude-grained recurring interactions across microservices within a service. However, as the number of equivalent microservices grows, applying a crude-grained pattern may cause severely unbalanced QoS, while nesting these patterns is convoluted to implement and expensive to maintain. In this article, we introduce a novel workflow meta-pattern for defining fine-grained workflow patterns that describe QoS-optimal combined executions of equivalent microservices. The meta-pattern employs a domain-specific algebraic expression to specify the invocation sequences of equivalent microservices, and a Boolean function to determine whether to terminate the execution. To evaluate the applicability of our meta-pattern, we build a Scala functional programming library, by which we further develop edge computing and cognitive service applications. Our experiments show that applying our meta-pattern to define such workflow patterns saves programmer effort, while the resulting patterns effectively improve the QoS of distributed applications.

**Keywords:** Edge Computing, Equivalent Service, Service Orchestration

# 1 Introduction

The number of IoT devices has grown from 15 billion in 2015 to 26 billion in 2020, and is predicted to reach 75 billion in 2025 [1]. Instead of uploading raw sensor data to the cloud for processing, the emerging paradigm of edge computing [2] processes these data at the edge of the network to produce information that drives the execution of co-located applications. To facilitate the implementation of applications in this domain, the service/microservice oriented architecture (SOA) exposes the sensing and computing capabilities of edge-based mobile and IoT devices as services or microservices [3–6].

One of the major challenges in provisioning mobile services is achieving *QoS-optimality* [7, 8]. Because edge and IoT environments are less dependable than traditional cloud environments, mobile services are often unreliable and untrustworthy. When it is mobile and energy-harvesting devices that provide edge resources, the resulting services become vulnerable to partial failure and low reliability [7, 8]. Besides, operated in physically unprotected environments, devices can be compromised to report false information, so the services they provide become untrustworthy [9]. Moreover, the resource constraints of these devices render their services more sensitive to execution cost and latency.

The microservice architecture isolates business functionalities into fine-grained building blocks [10], and applies workflow patterns [11] to assemble the resulting microservices into services. *Microservices are considered equivalent if they satisfy the same requirements by different means* (e.g., authenticating a user via a password, biometrics, SMS, or touchscreen patterns). The reliability and trustworthiness of mobile services can be improved by exploiting *the combined execution of equivalent microservices* [12, 13].

Workflow patterns describe common execution strategies that solve recurrent problems in process-oriented applications. Workflow patterns that describe combined executions of equivalent microservices include failover, speculative parallel, and majority voting. These workflow patterns provide the same functionality, while enhancing certain QoS characteristics (e.g., speculative parallelism for performance, failover for reliability, and majority voting for trustworthiness). However, intended for a small number of equivalent microservices, these workflow patterns' crude-grained execution strategies cannot achieve optimal QoS for larger microservice numbers. For example, consider improving accuracy: with up to several equivalent microservices, majority voting improves accuracy without incurring unreasonable microservice usage fees. However, with a larger number of equivalent microservices, their combined usage fees can become prohibitive.

As compared to coarse-grained patterns, fine-grained patterns can balance QoS characteristics better. For example, to better balance cost and accuracy as compared to majority voting, some equivalent microservices can be executed first, with their results' coherence examined to determine whether to execute the remaining ones [13]; to improve reliability while controlling for costs, approximation algorithms are applied to discover optimal execution strategies [14].

However, fine-grained workflow patterns are hard and error-prone to express, implement, and maintain. Customizing workflow patterns in general-purpose programming languages is tedious, as it requires synchronizing multi-threaded execution and data exchanges. A fine-grained pattern can also be formed by nesting crude-grained patterns, but this approach suffers from several drawbacks: 1) some fine-grained patterns need to access the execution results of all constituent microservices, while some crude-grained patterns may not output their intermediate results; 2) nesting workflow patterns is hard to express and understand (i.e., to elucidate a nested workflow pattern, workflow expressions are accompanied by flow charts [15]).

This article introduces a workflow meta-pattern that declaratively specifies fine-grained workflow patterns for the combined execution of equivalent microservices. In particular, our meta-pattern describes a fine-grained workflow pattern as 1) an algebraic expression that denotes the invocation sequences of equivalent microservices and 2) a Boolean function that determines whether to terminate the execution. To demonstrate how our meta-pattern can concisely and flexibly express the combined execution of equivalent microservices on different programming platforms, we implemented it as a Scala library, and further integrated the resulting workflow patterns into realistic mobile services. We evaluated these integrations to determine how effectively our approach optimizes the QoS of mobile services and how much programmer effort it requires as compared to the state of the art.

The contribution of this article is three-fold:
**(1)** We introduce a meta-pattern for declaratively specifying fine-grained patterns that describe the combined execution of equivalent microservices to improve QoS.
**(2)** We concretely implement our meta-pattern in Scala to provide programming support for composing QoS-optimal mobile services.
**(3)** We apply our reference implementation to compose practical mobile services and empirically evaluate their QoS characteristics in dissimilar execution environments. Our evaluation shows that our design can be effectively reified to address some of the most pertinent problems in emerging applications, both enhancing various distributed execution properties and saving programmer effort.

The rest of this article is organized as follows: Section 2 introduces the background of this article and Section 3 demonstrates usage examples of fine-grained combined execution of equivalent microservices. Section 4 introduces the design of our meta-pattern. Section 5 details how we implement our design and apply it to engineer novel mobile services, as well as our empirical evaluation. Section 6 compares our approach with existing works. Section 8 concludes.

# 2 Background

We introduce workflow patterns as well as mobile and IoT services, background required to understand our contribution.

## 2.1 Workflow Patterns

In SOA, workflow patterns serve as basic building blocks. Based on their application targets, workflow patterns divide into multiple categories: control flow, resource, data,

and error handling patterns. We use the following control flow [16] to introduce a meta-pattern for generating fine-grained workflows for the combined execution of equivalent microservices:

- XOR Split: connects to multiple microservices that can be invoked. Only one branch executes given a condition.
- AND Split: connects to multiple microservices that can be invoked, with all branches executing in parallel.
- AND Join: connects from multiple microservices; the following process continues only upon receiving all results.
- Cancelling Discriminator: connects from multiple microservices; the following process continues upon receiving any result, with the remaining branches terminated.
- Cancelling Partial Join (a.k.a, M-out-of-N join): connects from N microservices; the following process continues upon receiving M results, with the remaining branches terminated.

## 2.2 Equivalent Microservices and their Combined Execution

Equivalent microservices satisfy the same requirements by different means. In this work, we assume that equivalent microservices share the same invocation interface (i.e., signature) and input/output parameters.

Several well-known workflow patterns describe the combined execution of equivalent microservices. As shown in Fig. 1, *failover* [17] improves reliability by switching to equivalent microservices upon failure; *speculative parallel execution* [12, 18] executes multiple equivalent microservices simultaneously and uses the first result to improve both reliability and latency; *majority voting* [19, 20] compares the execution results of multiple equivalent microservices and outputs the most likely result to improve trustworthiness.

## 2.3 Mobile and IoT services

As edge computing technologies evolve, mobile and IoT services become possible [21, 22]: mobile and IoT devices at the edge expose their sensing and computing capabilities as services, accessed by nearby client devices.

Equivalent microservices are common in mobile environments: 1) *recognizing facial images using services provided by different vendors* [13]; 2) *authenticating users* by means of a password, biometrics (fingerprint, iris or facial image), SMS, or touchscreen patterns [23–26]; 3) *detecting atmospheric particulate matter value (PM2.5)* by reading from a portable PM2.5 sensor, estimating from images [27, 28], or invoking the web service of the nearest environmental station; *detecting crowds* by reading a weight sensor, recognizing persons from the area's camera image[29], using an entrance-exit counting device, or counting WiFi beacons[30], or using modern smartphones[31].

When provisioning mobile services, *QoS-optimality* is hard to achieve [7, 8]. The mobility and diversified ownership of these devices lead to low reliability and trustworthiness. The combined execution of equivalent microservices can improve these QoS characteristics. However, compared with web service composition, mobile services pose

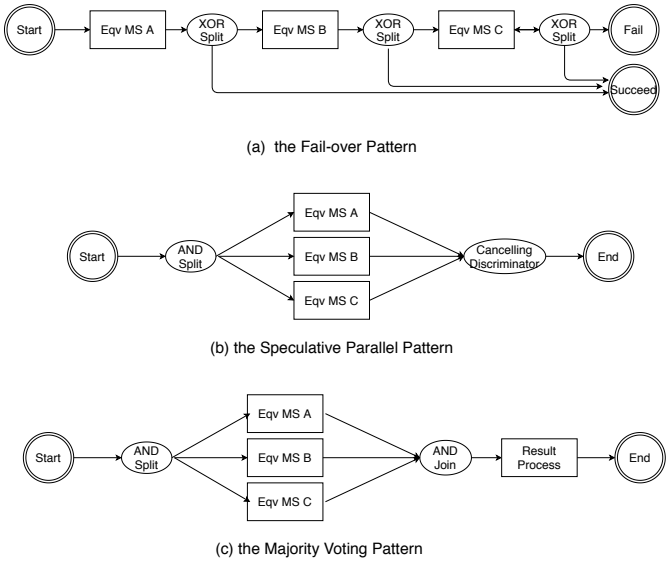


**Fig. 1** Combined Execution of Equivalent Microservices A, B, and C

two unique challenges: 1) the variety and number of equivalent mobile services are significantly larger as compared with web services, as data-rich mobile environments feature multiple ways to satisfy the same requirement while microservices provided by different mobile devices are also considered equivalent [32]; 2) mobile services are provided in resource-constrained environments, rendering them more in need of QoS optimality. In the next section, we demonstrate by example how a larger number of equivalent microservices requires a fine-grained workflow pattern to optimize service QoS.

# 3 Motivating Scenario

Authentication, expression analysis, and emotion recognition rely on detecting and locating faces in images and videos. Facial detection is provided as web services by different vendors and as deployable mobile services [13]. However, none of these equivalent microservices is 100% accurate, as the quality of input images and videos affect the accuracy of these functionalities [33]. To improve accuracy, the majority voting pattern has been applied [34], which executes all alternatives simultaneously, and waits till receiving all results to determine the final output. We use facial detection as an example to demonstrate how fine-grained patterns improve overall QoS as compared with majority voting and the problems in constructing such patterns.

## 3.1 Fine-Grained Patterns for Optimizing QoS

Alas, majority voting improves accuracy at the expense of increasing execution latency and cost: the additional latency is incurred by the necessity to wait for those microservices that takes longer to execute, while the additional cost is incurred by the necessity

to invoke all equivalent microservices. In the presence of equivalent microservices whose execution latency or cost is unusually high, a more fine-grained workflow pattern can better optimize the QoS of the combined execution of equivalent microservices.

To demonstrate how fine-grained workflow patterns work, we denote three equivalent microservices as "A", "B", and "C". To increase accuracy while reducing the overall execution latency, **example pattern 1** executes "A", "B", and "C" simultaneously, and terminates upon receiving two coincident results. If the first two results are the same, the execution can terminate without waiting for the third result, which could incur unusually high latency. To increase accuracy while reducing the overall cost, **example pattern 2** first executes "A" and "B" simultaneously, and waits for both of their results. If "A" and "B" return the same result, output it as final; otherwise, execute "C" and output the results agreed upon by any two microservices.

## 3.2 Problems with Expressing Fine-Grained Patterns

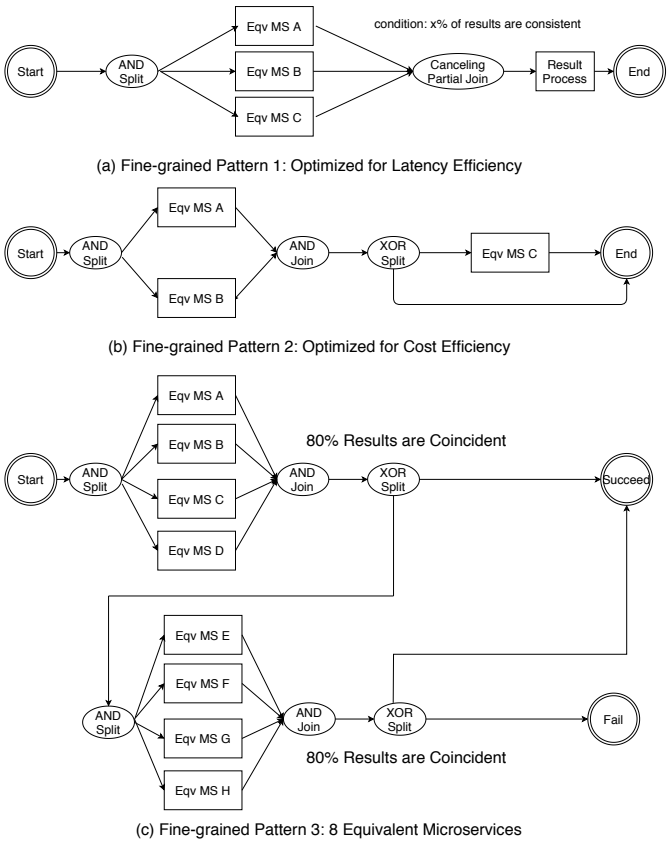


**Fig. 2** Fine-grained Combined Execution of Microservices A, B, and C

Fig. 2 demonstrates how workflow constructs can express the aforementioned fine-grained patterns. For the pattern in Fig. 2.a, we change the semantics of the standard construct "M-out-of-N join" from "terminating upon receiving M results from N branches" to "terminating upon a certain condition,", i.e., "two received results coincide" in our case.

Although standard pattern constructs can fully support the pattern in Fig. 2.b, the required number and complexity of workflow constructs would be much higher than in the standard majority voting pattern. As the number of equivalent microservices grows, expressing such patterns would become unwieldy. For example, **example pattern 3** can be: for 8 equivalent microservices, execute every four in a row, and continue to execute the next row of four microservices only if less than "80%" of previous results coincide. It takes 6 workflow constructs to express this pattern, while the condition of "80% of all results coincide" needs to be repeated twice (Fig. 2.c). To make things worse, the "XOR split" needs the execution results of all equivalent microservices to determine the next step, while these results serve as intermediate information and are not exposed to these external "XOR split" constructs.

The necessity to change the semantics and data access of basic workflow constructs makes nesting workflow constructs tedious and error-prone, while unwieldy nested workflow patterns are hard to understand and maintain. These shortcomings motivate the need for dedicated programming support for the fine-grained combined execution of equivalent microservices.

# 4 Meta-pattern Design and Implementation

To express and manage the combined execution of equivalent microservices, we design a meta-pattern that generates workflow patterns with the following properties:

1. Applicable to microservices that are equivalent;
2. The generated workflow shares the same input and output with its constituent microservices, thus providing the same functionality;
3. Compared with its constituent microservices, the generated workflow improves at least one QoS characteristic.

In the rest of this section, we introduce the syntax, semantics, design rationale, and visualization of our meta-pattern, as well as its applicability and runtime support.

## 4.1 Meta-Pattern Syntax and Semantics

Formally, the meta-pattern expresses a pattern as a triple $m =< \theta, \zeta, \omega >$:

- $\theta$: a set of equivalent microservices;
- $\zeta$: an invocation sequence, an expression that denotes the complete execution order of the equivalent microservices;
- $\omega$: a terminating condition, a Boolean function that takes as input the receive results of microservices' executions and outputs whether to terminate the workflow execution.

For example, the workflow pattern in Fig. 2.b can be expressed by Fig. 3. It describes a sequence of "executing A and B in parallel first (i.e., $A * B$), and then

C (i.e., $-C$)", which can be short-circuited upon reaching the condition: the mostly agreed upon result should reach at least 60% of all received votes.

```
1 m_1=< θ = (A, B, C),
2       ζ = A*B-C,
3       ω = mostVotedResult.votes/totalVotes>=0.6 >
```

**Fig. 3** Meta-Pattern for Expressing Fine-Grained Pattern 2

An invocation sequence is expressed by a set of equivalent microservices and the operators connecting them into an expression. The binary operators $-$ and $*$ denote a sequential and a parallel execution, respectively. For example, given two equivalent microservices $a$ and $b$, $a - b$ expresses that the microservices are to be executed in sequence from left to right, while $a * b$ expresses that the microservices are to be executed in parallel. Notice that because the $-$ and $*$ operators take equivalent microservices as their operands, the traditional built-in operator precedence is slightly altered. For example, for the invocation sequence $a - b * c$, $a$ is executed first; then $b$ and $c$ are executed in parallel. The parentheses operators denote that the invocation sequence inside a pair of parentheses is considered as one equivalent microservice. For example, $a * b - c$ means to execute $a$ and $b$ in parallel first and then $c$, while $a * (b - c)$ means to treat $b - c$ as an equivalent microservice, and execute $a$ and $b - c$ in parallel. Fig. 4 gives the EBNF grammar of an invocation sequence.

```
1 invokeSeq(ζ) ::= f|(f)|f-f|f*f, ∀f ∈ θ
```

**Fig. 4** EBNF Definition for Invocation Sequence Specification

An invocation sequence can terminate at different points between runs. In the example in Fig. 3, "C" would be executed only of "A" and "B" return different results. Terminating conditions control such variability across runs.

## 4.2 Design Considerations

Our meta-pattern for the combined execution of equivalent microservices describes a workflow pattern as an invocation sequence of microservices that is short-circuited upon reaching a specified condition. Two observations inform our design:

1. For the combined execution of equivalent microservices, different terminating conditions determine which QoS characteristic to enhance, while different invocation sequences determine how to balance the remaining QoS characteristics. Common terminating conditions include: any results is received (to enhance reliability) and the received results coincide (to enhance trustworthiness). The equivalent microservices are executed either in parallel or in sequence; the parallel execution incurs additional cost but shortens latency, and the sequential execution is vice versa. To optimize service QoS, one can vary the terminating conditions and the invocation sequences of equivalent microservices.
2. An invocation sequence cannot be altered, only continued or discontinued. The generated patterns only apply to equivalent microservices, so a microservice's

result cannot serve as a control flow condition that determines which microservice to execute next. Hence, an invocation sequence is expressed with no control flow constructs, with "-" and "*" denoting sequential and parallel invocations, respectively.

It has become common to use the "|" and "→" operators to express parallel and sequential execution, respectively [35]. In our design, we intentionally avoid using the same operators to express the parallel or sequential execution of equivalent microservices that can be short-circuited. Besides, based on our definition of equivalent microservices (i.e., all microservices take the same input and generate the same output), workflow patterns specified by our meta-pattern coordinate the execution of equivalent microservices that share the same input and output parameters. Hence, the algebraic expressions exclude invocation parameters.

### 4.3 Runtime Support for Executing Patterns

Executing the workflow patterns specified by our meta-pattern requires dedicated runtime support. The flowchart in Fig.5 explains the execution logic for this runtime support:

- $S_0$: to start executing, a workflow pattern receives input parameters and then transitions to state $S_1$, "execute next equivalent microservices".
- $S_1$: based on $\zeta$, determine which microservices to execute next and initialize them, transitioning to state $S_2$, "waiting for any results to be returned".
- $S_2$: wait to receive any microservice execution result or for the overall execution to timeout. Upon timeout, transition to the"failure" state. Upon receiving a result, persist it with the other microservice execution results, transition to state $S_3$, "applying the terminating condition."
- $S_3$: apply the terminating condition $\omega$, output a Boolean value indicating whether to terminate the execution. If true, transition to the final state,"success and pruning", which terminates all unfinished microservices and outputs the final result; Otherwise, transition to state $S_4$, "checking whether there is any microservice running."
- $S_4$: If no microservices are still running, transition to state $S_5$, "checking if the invocation sequence has reached the end;" otherwise, transition to state $S_1$.
- $S_5$: check if there are still microservices in $\zeta$ waiting to be executed. If true, transition to state $S_0$; otherwise, transition to the "failure" state.

## 5 Reference Implementation and Evaluation

We implement our meta-pattern design as a Scala library for functional programming. We demonstrate that the meta-pattern is expressive enough to generate fine-grained workflows that enhance the performance of mobile/IoT services. Our evaluation shows that compared with crude-grained patterns, the generated fine-grained patterns improve the overall QoS.

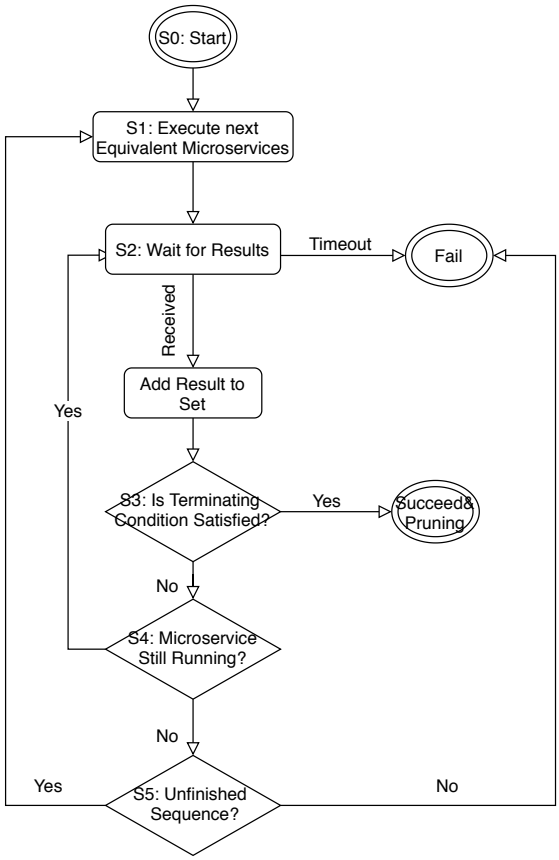


**Fig. 5** Runtime Support for Executing Generated Patterns

## 5.1 Reference Implementations

The Scala-based reference implementation (Scala SDK 2.12.8) comprises approximately 870 lines of code (ULOC). To allow any set of functions with the same signature to represent equivalent microservices, the library features a generic function container and an invocation sequence class. Each constituent function is wrapped into a container object, whose operators `-` and `*` are overloaded to generate an invocation sequence. The Scala compiler checks if all functions forming an invocation sequence share the same signature. An invocation sequence sets its terminating condition by calling the `terminate` method, and executes the equivalent functions by calling the overridden `apply` method, returning a mapping of `(functionName, executeResult)`.

We observe that an invocation sequence naturally maps into a tree structure that can serve as its runtime representation. The tree structure's nodes have three types: `leaf`, `sequential`, and `parallel`. A `leaf` is an equivalent functionality. A `sequential` node has its `left` and `right` children, and a `parallel` node has two or more `child` nodes.

Algorithm 1 explains how to implement the runtime using the tree structure. To create and manage concurrency, our implementation uses the `Future`, `Promise`, and `concurrentMap` APIs. A concurrent access protected key-value data structure

**Algorithm 1** Execute a Specified Meta-pattern

**Input:** $p$: execution parameter; $< \theta, \zeta, \omega >$
**Output:** $r$: result

```
1: execute(ζ.root)
2: function EXECUTE(t:TREE)(Boolean)
3:     switch t.Type do
4:         case Leaf(v)
5:             resultMap ← resultMap + (v.funcName, v.func(p))
6:             return terminator.check(resultMap)
7:         case SequentialNode(left, right)
8:             if execute(left) then
9:                 return true
10:            else
11:                return execute(right)
12:            end if
13:        case ParallelNode(chidren)
14:            fSet ← ∅
15:            for each c ∈ children do
16:                fSet ← fSet + Future(execute(c))
17:            end for
18:            Wait any f ∈ fSet.Complete:
19:            if f==True then
20:                return True                          ▷ Early Termination
21:            else if fSet.all.isCompleted then
22:                return False
23:            else
24:                continue Wait
25:            end if
26: end function
```

(`resultMap`) maps the completed equivalent functions and their results. A recursive procedure starts from the tree structure's `root` node, and returns `true`, as soon as the terminating condition is fulfilled. Upon reaching a `leaf` node, its equivalent function is executed, with the result stored in the key-value structure. All the stored results are checked after each completed function if the pattern's terminating condition has been fulfilled (line 6). For a `sequential` node, its `left` node is executed first, followed by executing its `right` node if the recursive procedure of its `left` node returns `false`. For a `parallel` node, all `child` nodes are executed in parallel, and the parallel node waits for the results of these recursive procedures. If any of the `child` nodes fulfills the terminating condition and returns `true`, the `parallel` node returns `true` (line 20) and the parallel execution is terminated without waiting for the other branches to complete. Otherwise, it continues to wait, until all `child` nodes' executions fail to fulfill the terminating condition and return `false`. After the recursive procedure completes, it returns the stored final results to the caller.

## 5.2 Applying Meta-Pattern to Mobile Service

We adopt the mobile services provisioning system model introduced in [36]. In particular, the system features a local gateway that collects the available microservices, provided by mobile and IoT devices. For a given mobile service request with reliability, trustworthiness, and QoS-optimality requirements [37], the gateway orchestrates the

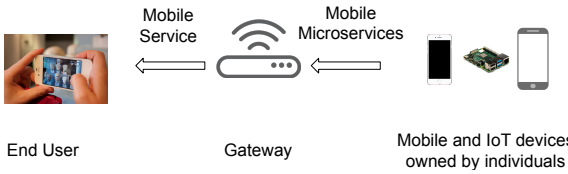


**Fig. 6** System Components for Provisioning Mobile Services

combined execution of equivalent microservices, provided by mobile and IoT devices, which can be unreliable and untrustworthy [36].

### 5.2.1 Enhancing Service's Accuracy

```
1  // invoke a web service:
2  def ibm(image:String):Boolean = {...}
3  def ms(image:String):Boolean = {...}
4  def face(image:String):Boolean = {...}
5  // invoke an edge service:
6  def dl(image:String):Boolean = {
7      val reg = new EdgeReg() //connect to an edge gateway
8      val edgeService = reg.query("deepLearningFaceDetection")
9      result = edgeService.execute(image)
10 }
11 def opencv(image:String):Boolean = {...}
12 // Specify equivalent microservices
13 val (e1, e2, e3, e4, e5) = (eqv(ibm), eqv(ms), eqv(face), eqv(dl),
       eqv(opencv))
14 // Specify an invocation sequence
15 val seq = e4*e5 - e1*e2*e3
16 // Specify a terminating condition
17 seq.terminate(majorityVoting())
18 // Execute and process result
19 val result = seq('img.jpg').groupBy(_._2).maxBy(_._2.size)._1
```

**Fig. 7** Specifying Mobile Service in Scala Library

To detect faces, developers can choose proprietary cloud services (IBM[1], Microsoft[2], and Face++[3]) or deploy open-source libraries as edge services (deep learning based[4] and openCV Cascade classifier based[5]). Fig. 7 shows how with our Scala library, a fine-grained workflow pattern that enhances service accuracy can be implemented in 4 lines of code. Lines 1-11 implement microservices `ibm, ms, face, dl, opencv`, taking a String (i.e., image file) and returning a Boolean (i.e., face detected). Line 13 wraps them up in equivalent microservice containers. Line 15 uses the overloaded operators to declare an invocation sequence, and Line 17 sets the terminating condition for the invocation sequence. Line 19 executes the specified pattern

[1] https://www.ibm.com/watson/services/visual-recognition/
[2] https://azure.microsoft.com/en-us/services/cognitive-services/face/
[3] https://www.faceplusplus.com/face-detection/
[4] https://github.com/ageitgey/face_recognition
[5] https://docs.opencv.org/3.4/d1/de5/classcv_1_1CascadeClassifier.html

**Table 1** Facial Detection Microservices

| id | func | cost | latency (ms) | accuracy |
|---|---|---|---|---|
| e1 | IBM | 400 | 95 | 0.918 |
| e2 | MS | 100 | 74 | 0.737 |
| e3 | Face++ | 50 | 96 | 0.898 |
| e4 | DL-based | 2 | 56 | 0.642 |
| e5 | openCV-based | 2 | 66 | 0.676 |

**Table 2** QoS of Facial Detection Services

| invocation sequence | accuracy | cost | latency |
|---|---|---|---|
| Crude-grained Majority Voting | 0.859 | 554 | 97 |
| $e_4 * e_5 - e_2$ | 0.739 | 23 | 81 |
| $e_4 * e_5 - e_3$ | 0.812 | 13 | 85 |
| $e_1 * e_3 * e_5$ | 0.908 | 452 | 95 |
| $e_3 * e_5 - e_1$ | 0.908 | 110 | 110 |
| $e_2 * e_3 - e_1$ | 0.939 | 311 | 134 |

with the input of "`img.jpg`" and obtains the execution result that is agreed by most microservices.

### 5.2.2 Enhancing Service's Reliability

Edge environments typically feature multiple sensors, whose input can be used to satisfy the data requirements of an edge service. Consider a service that obtains ambient temperature. Ambient temperature can be obtained by: 1) directly reading a local temperature sensor (`readTempSensor`); 2) estimating based on the CPU temperature of an edge computer [38] (`estTemp`); 3) reading from a web service based on the current location (`readLocationTemp`). In most edge environments, the `readTempSensor` microservice is first executed to provide a low-latency, low-cost, and accurate temperature reading. However, if some microservices are unavailable in a given environment, `estTemp` and `readLocationTemp` are executed next in parallel as fail-over to guarantee an acceptable latency. Hence, we express the `getTemp` as `readTempSensor-estTemp*getLocationTemp`.

## 5.3 Performance Evaluation

### 5.3.1 Enhancing Accuracy

To verify how the generated workflow patterns work for the aforementioned application scenario, we use the image dataset collected from WiKi [33] as an alternative for human labeling, in which each image contains a face. We deploy the edge services on a Dell desktop with a i7-4790@3.6GHz processor and 16GB RAM. Table 1 shows the average latency, accuracy, and cost of each equivalent microservice. The cost of invoking the web services provided by IBM, Microsoft, and Face++ are \$0.004, \$0.001, and \$0.0005 per request, respectively. Assuming the average electricity rate of \$0.12 per kWh, and the power supply of the experimental desktop of 0.65kW, the costs of microservices become 400, 100, 50, 2, and 2, respectively.

Table 2 compares the QoS of crude-grained majority voting and fine-grained patterns generated by our meta-pattern. We observe that: 1) the generated plan can be extremely cost/latency efficient by first executing the two open source implementations

deployed at the edge. Compared with invoking the IBM web service, the generated pattern saves as much as 97% of the execution cost, while reducing the accuracy by 13%; 2) the workflow pattern $e_3 * e_5 - e_1$ strikes a good balance between accuracy and cost. By invoking a low-cost web service and an open-source implementation first, the execution gains more accuracy than when using the two cost-efficient open-source implementations. Compared with invoking the IBM service, the generated pattern saves 72.5% of execution cost, while the accuracy only decreases by 3.2%; 3) compared with the crude-grained majority voting pattern, the generated patterns on average reduce the cost by 67.6%, with less than 3% differences in accuracy and latency. This observation confirms our motivation: fine-grained workflow patterns do optimize performance.

**Table 3** QoS of Service "getTemp"

| Pattern | Invocation Sequence | Reliability | Latency | Cost |
|---|---|---|---|---|
| Speculative Parallel | $e_1 * e_2 * e_3$ | 100% | 56 ms | 148 |
| Fine-Grained | $e_1 - e_2 * e_3$ | 99% | 69 ms | 74.5 |

### 5.3.2 Enhancing Reliability

The execution environment features a mobile device (Moto G6) that queries the "getLocationTemp" microservice, a temperature sensor (Raspberry Pi 3 and DS18B20) for "readTempSensor", and an edge server (ThinkCentre M900 Tiny) for "estTemp". The sensor is only available for 60% of all requests. We further set the cost for executing each microservice to 50 points. Table 3 compares the reliability, latency, and cost of the speculative parallel execution and the fine-grained workflow pattern. The fine-grained pattern reduces the average latency by 49.7%, at the expense of 23.2% additional latency, as compared with the speculative parallel workflow pattern.

## 6 Related Work

Most existing approaches related to systemically leveraging equivalence focus on programming and system support for pre-defined execution strategies. For example, Orc [35] provides abstractions for declaratively specifying the fail-over and speculative parallel execution of two function units; NVP [39] introduces a programming model to implement equivalent functions for parallel execution and processing their execution results with majority voting; Eureka [40] introduces a programming model that supports speculative parallel execution; references [41] and [42] introduce several predefined execution strategies that combine sequential and parallel execution strategies, expressed as BPEL [43] or WSDL workflow constructs. However, these predefined strategies only cover a limited portion of all possible execution strategies. Compared with these approaches, the flexibility of our meta-pattern makes it possible to express and execute any strategy.

Other approaches enable developers to implement the control logic for equivalent execution but are often non-trivial and error-prone. For example, Baresi et al. [44] introduce self-healing BPEL, for manually writing logic to control the execution of equivalent services. Dino [45] provides semantic and runtime support for atomic tasks

by creating and recovering from checkpoints, which ensures the stateless execution of equivalent functionalities. Compared with these approaches, our meta-pattern enables developers to concisely express equivalent execution.

Several studies of web service composition have observed that different execution strategies of a set of equivalent services may lead to dissimilar execution characteristics [14, 15, 20, 46]. Some of them provide system or algorithmic support for approximating the sub-optimal execution strategies for a given set of QoS requirements. As the number of fine-grained patterns can get exponentially large, they cannot be named explicitly, so these works express them by nesting basic workflow constructs, which could be error-prone to express and hard to understand. For example, to express an execution strategy for equivalent services, a tree graph was used [15].

Meta-patterns have been previously applied to scientific computing and exception handling [47–49], as a flexible approach to express complicated workflow patterns [50]. However, existing meta-patterns include full control flow semantics for generality. To the best of our knowledge, this article is the first to provide a meta-pattern tailored for the combined execution of equivalent microservices, so the generated patterns can provision QoS-optimal services in unreliable, untrustworthy, and cost/latency-sensitive mobile/IoT environments.

# 7 Discussion

We envision that the applicability of our meta-pattern framework should extend to large-scale application environments. Unfortunately, at the time of this writing, we are unable to support this claim empirically, as we lack the computing and personnel resources required to obtain evaluation results from the field. Instead, we next list several realistic domains that could benefit from our approach and discuss the applicability of our design and its potential issues.

**Smart Homes:** encompass a wide array of devices and services designed to automate and enhance various aspects of home living. For example,

- *Home security systems*, including smart locks, surveillance cameras, and alarm systems, heavily rely on trustworthiness. Trustworthiness is crucial as homeowners need to be able to trust that these systems effectively protect their homes while maintaining privacy. In this case, the developer can develop multiple equivalent intrusion detection microservices (e.g., camera-based, motion-based, audio-based), and specify the termination condition of the meta-pattern as "give a warning if any microservice returns true." The termination condition also improves the responsiveness of the system, as it sends alarms upon receiving the first positive detection result. However, it also increases the chance of sending false alarms, thus requiring that each microservice improve its accuracy.
- *Voice Assistants*: Devices like Amazon Echo or Google Home must incur minimal latency to provide a seamless interaction experience. Reliability is crucial as users depend on assistants for various tasks, ranging from setting reminders to controlling other smart home devices. To ensure the user's voice command is correctly captured, even in noisy environments or for users with speech differences, developers could invoke multiple voice translation web services and specify

the termination condition as "when a majority of these approaches reach a consensus." However, the approach may incur additional costs. Hence, we plan to extend this work by validating the results of equivalent microservices for a given user, and dynamically selecting the microservice that best fits their context.

**Autonomous Driving** is a field in which the accuracy and latency of systems are of paramount importance, particularly in applications such as detecting pedestrians. In autonomous vehicles, the three primary technologies used for this purpose are Radar, Lidar, and cameras. Each of these technologies has unique characteristics that affect their performance in terms of accuracy and latency. In autonomous vehicles, these technologies are often used in conjunction to compensate for each other's weaknesses. For instance, radar can detect objects in challenging weather conditions where cameras and Lidar might struggle, while Lidar and cameras provide the detailed object recognition necessary for complex urban environments. Developers can rely on our meta-pattern to intuitively specify their combination in a fine-grained fashion, like "using Camera and Lidar first, and switch to Radar if they both fail."

To summarize, our meta-pattern can improve QoS for various applications while saving programming effort. The framework should be feasible to deploy in realistic settings, as it can orchestrate the execution of microservices built as standard Docker images and as Android apps. Specifically, for Android classes, we incorporated a Java reflection-based execution method, a concept we introduced and detailed in our earlier work [51].

# 8 Conclusion

We have presented a meta-pattern that declaratively expresses fine-grained workflow patterns for the combined execution of equivalent microservices to improve QoS. The meta-pattern declaratively specifies a fine-grained pattern as a set of equivalent microservices, an invocation sequence, and a terminating condition. Our evaluation demonstrates that our approach is expressive and effective, presenting a viable solution that helps conquer the complexity of reliable, accurate, and efficient execution in distributed execution environments with scarce and unreliable resources.

# Acknowledgment

This research is supported by NSF through grants #2104337, and #2232565.

# Compliance with Ethical Standards

The authors have no relevant financial or non-financial interests to disclose.

# References

[1] Statista Research Department: Internet of Things - Number of Connected Devices Worldwide 2015-2025. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[2] Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE Internet of Things Journal **3**(5), 637–646 (2016)

[3] Wu, H., Deng, S., Li, W., Yin, J., Li, X., Feng, Z., Zomaya, A.Y.: Mobility-aware service selection in mobile edge computing systems. In: 2019 IEEE International Conference on Web Services (ICWS), pp. 201–208 (2019). IEEE

[4] Sun, M., Zhou, Z., Zhang, W., Hung, P.C.: Iot service composition for concurrent timed applications. In: 2019 IEEE International Conference on Web Services (ICWS), pp. 50–54 (2019). IEEE

[5] Moeini, H., Yen, I.-L., Bastani, F.: Service specification and discovery in iot networks. In: 2019 IEEE International Conference on Web Services (ICWS), pp. 55–59 (2019). IEEE

[6] Achir, M., Abdelli, A., Mokdad, L., Benothman, J.: Service discovery and selection in iot: A survey and a taxonomy. Journal of Network and Computer Applications **200**, 103331 (2022)

[7] El-Sayed, H., Sankar, S., Prasad, M., Puthal, D., Gupta, A., Mohanty, M., Lin, C.-T.: Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment. IEEE Access **6**, 1706–1717 (2017)

[8] Varghese, B., Wang, N., Barbhuiya, S., Kilpatrick, P., Nikolopoulos, D.S.: Challenges and opportunities in edge computing. In: Smart Cloud (SmartCloud), IEEE International Conf. On, pp. 20–26 (2016). IEEE

[9] Chiang, M., Zhang, T.: Fog and iot: An overview of research opportunities. IEEE Internet of Things Journal **3**(6), 854–864 (2016)

[10] Hassan, S., Bahsoon, R.: Microservices and their design trade-offs: A self-adaptive roadmap. In: 2016 IEEE International Conference on Services Computing (SCC), pp. 813–818 (2016). IEEE

[11] Der Aalst, W.M., Ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and parallel databases **14**(1), 5–51 (2003)

[12] Song, Z., Tilevich, E.: A programming model for reliable and efficient edge-based execution under resource variability. In: 2019 IEEE International Conf. on Edge Computing (EDGE), pp. 64–71 (2019)

[13] Bhatia, A., Li, S., Song, Z., Tilevich, E.: Exploiting equivalence to efficiently enhance the accuracy of cognitive services. In: 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 143–150 (2019). IEEE

[14] Hiratsuka, N., Ishikawa, F., Honiden, S.: Service selection with combinational use

of functionally-equivalent services. In: Web Services (ICWS), IEEE International Conference On, pp. 97–104 (2011). IEEE

[15] Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Presti, F.L., Mirandola, R.: Moses: A framework for qos driven runtime adaptation of service-oriented systems. IEEE Transactions on Software Engineering **38**(5), 1138–1159 (2011)

[16] Workflow Patterns. Workflow Patterns Initiative (2017)

[17] Aldwyan, Y., Sinnott, R.O.: Latency-aware failover strategies for containerized web applications in distributed clouds. Future Generation Computer Systems **101**, 1081–1095 (2019)

[18] Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: 2012 Proceedings IEEE Infocom, pp. 945–953 (2012). IEEE

[19] Yen, I.-L., Bastani, F., Solanki, N., Huang, Y.: Trustworthy computing in the dynamic iot cloud. In: 2018 IEEE International Conference on Information Reuse and Integration (IRI), pp. 411–418 (2018). IEEE

[20] Song, Z., Rowader, O., Li, Z., Tello, M., Tilevich, E.: Quality of information matters: Recommending web services for performance and utility. In: 2022 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 41–48 (2022). IEEE

[21] Qiao, Y., Nolani, R., Gill, S., Fang, G., Lee, B.: Thingnet: A micro-service based iot macro-programming platform over edges and cloud. In: 2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), pp. 1–4 (2018). IEEE

[22] Cheng, B., Solmaz, G., Cirillo, F., Kovacs, E., Terasawa, K., Kitazawa, A.: Fogflow: Easy programming of iot services over cloud and edges for smart cities. IEEE IoT Journal **5**(2), 696–707 (2018)

[23] Rogowski, M., Saeed, K., Rybnik, M., Tabedzki, M., Adamski, M.: User authentication for mobile devices. In: Computer Information Systems and Industrial Management: 12th IFIP TC8 International Conference, CISIM 2013, Krakow, Poland, September 25-27, 2013. Proceedings, pp. 47–58 (2013). Springer

[24] Mahfouz, A., Mahmoud, T.M., Eldin, A.S.: A survey on behavioral biometric authentication on smartphones. Journal of information security and applications **37**, 28–37 (2017)

[25] Wang, C., Wang, Y., Chen, Y., Liu, H., Liu, J.: User authentication on mobile devices: Approaches, threats and trends. Computer Networks **170**, 107118 (2020)

[26] Zaidi, A.Z., Chong, C.Y., Jin, Z., Parthiban, R., Sadiq, A.S.: Touch-based continuous mobile device authentication: State-of-the-art, challenges and opportunities. Journal of Network and Computer Applications **191**, 103162 (2021)

[27] Liu, X., Song, Z., Ngai, E., Ma, J., Wang, W.: Pm2: 5 monitoring using images from smartphones in participatory sensing. In: Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference On, pp. 630–635 (2015). IEEE

[28] Song, S., Li, V.O., Lam, J.C., Wang, Y.: Personalized ambient pollution estimation based on stationary camera-taken images under cross-camera information sharing in smart city. IEEE Internet of Things Journal (2023)

[29] Li, M., Zhang, Z., Huang, K., Tan, T.: Estimating the number of people in crowded scenes by mid based foreground segmentation and head-shoulder detection. In: Pattern Recognition, 2008. ICPR 2008. 19th International Conference On, pp. 1–4 (2008). IEEE

[30] Schauer, L., Werner, M., Marcus, P.: Estimating crowd densities and pedestrian flows using wi-fi and bluetooth. In: MobiQuitous 2014, pp. 171–177 (2014)

[31] Torkamandi, P., Pajevic Kärkkäinen, L., Ott, J.: Characterizing wi-fi probing behavior for privacy-preserving crowdsensing. In: Proceedings of the 25th International ACM Conference on Modeling Analysis and Simulation of Wireless and Mobile Systems, pp. 203–212 (2022)

[32] Song, Z., Tilevich, E.: Equivalence-enhanced microservice workflow orchestration to efficiently increase reliability. In: 2019 IEEE International Conference on Web Services (ICWS), pp. 426–433 (2019). IEEE

[33] Jung, S.-G., An, J., Kwak, H., Salminen, J., Jansen, B.J.: Assessing the accuracy of four popular face recognition tools for inferring gender, age, and race. In: Twelfth International AAAI Conference on Web and Social Media (2018)

[34] Vuurens, J., Vries, A.P., Eickhoff, C.: How much spam can you take? an analysis of crowdsourcing results to increase accuracy. In: Proc. ACM SIGIR Workshop on Crowdsourcing for Information Retrieval (CIR'11), pp. 21–26 (2011)

[35] Kitchin, D., Quark, A., Cook, W., Misra, J.: The orc programming language. In: Formal Techniques for Distributed Systems: Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings, pp. 1–25 (2009). Springer

[36] Song, Z., Tilevich, E.: Pmdc: Programmable mobile device clouds for convenient and efficient service provisioning. In: 2018 IEEE 11th International Conf. on Cloud Computing (CLOUD), pp. 202–209

[37] Le, M., Song, Z., Kwon, Y.-W., Tilevich, E.: Reliable and efficient mobile edge computing in highly dynamic and volatile environments. In: 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC), pp. 113–120 (2017). IEEE

[38] Krintz, C., Wolski, R., Golubovic, N., Bakir, F.: Estimating outdoor temperature from cpu temperature for iot applications in agriculture. In: International Conference on the Internet of Things (2018)

[39] Hu, T., Bertolott, I.C., Navet, N.: Towards seamless integration of n-version programming in model-based design. In: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–8 (2017). IEEE

[40] Imam, S., Sarkar, V.: The eureka programming model for speculative task parallelism. In: 29th European Conference on Object-Oriented Programming (ECOOP 2015) (2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik

[41] Zheng, Z., Lyu, M.R.: A distributed replication strategy evaluation and selection framework for fault tolerant web services. In: 2008 IEEE International Conference on Web Services, pp. 145–152 (2008). IEEE

[42] Russell, N., Ter Hofstede, A.H., Van Der Aalst, W.M., Mulyar, N.: Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPMcenter. org, 06–22 (2006)

[43] Louridas, P.: Orchestrating web services with bpel. IEEE software **25**(2), 85–87 (2008)

[44] Baresi, L., Guinea, S.: Self-supervising bpel processes. IEEE Transactions on Software Engineering **37**(2), 247–263 (2010)

[45] Lucia, B., Ransford, B.: A simpler, safer programming and execution model for intermittent systems. ACM SIGPLAN Notices **50**(6), 575–585 (2015)

[46] Song, Z., Tilevich, E.: Win with what you have: Qos-consistent edge services with unreliable and dynamic resources. In: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pp. 530–540 (2020). IEEE

[47] Abouelhoda, M., Alaa, S., Ghanem, M.: Meta-workflows: pattern-based interoperability between galaxy and taverna. In: Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science, pp. 1–8 (2010)

[48] Taghiyar, M.J., Rosner, J., Grewal, D., Grande, B.M., Aniba, R., Grewal, J., Boutros, P.C., Morin, R.D., Bashashati, A., Shah, S.P.: Kronos: a workflow assembler for genome analytics and informatics. Gigascience **6**(7), 042 (2017)

[49] Kumar, A., Wainer, J.: Meta workflows as a control and coordination mechanism

for exception handling in workflow systems. Decision Support Systems **40**(1), 89–105 (2005)

[50] Reichert, M., Rinderle-Ma, S., Dadam, P.: Flexibility in process-aware information systems. Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems, 115–135 (2009)

[51] Song, Z., Chadha, S., Byalik, A., Tilevich, E.: Programming support for sharing resources across heterogeneous mobile devices. In: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, pp. 105–116 (2018)