Advanced Function Composition in Serverless Platforms

TIAGO LUÍS LIMA DA SILVA

(Licenciado)

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Doutor Filipe Bastos de Freitas
Doutor José Manuel de Campos Lages Garcia Simão

Júri:

Presidente: Doutor Nuno Miguel Machado Cruz
Vogais: Doutor Filipe Bastos de Freitas
Doutor Manuel Martins Barata

Outubro 2025

Advanced Function Composition in Serverless Platforms

TIAGO LUÍS LIMA DA SILVA

(Licenciado)

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Doutor Filipe Bastos de Freitas, ISEL
Doutor José Manuel de Campos Lages Garcia Simão, ISEL

Júri:

Presidente:	Doutor Nuno Miguel Machado Cruz, ISEL
Vogais:	Doutor Filipe Bastos de Freitas, ISEL
	Doutor Manuel Martins Barata, ISEL

Outubro 2025

Statement of integrity

I declare that this dissertation is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

Tiago Luís Lima da Silva

Lisbon, October, 2025

Advanced Function Composition in Serverless Platforms

Copyright© TIAGO LUÍS LIMA DA SILVA, Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa.
The Instituto Superior de Engenharia de Lisboa and the Instituto Politécnico de Lisboa have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

Abstract

Serverless computing, particularly Function-as-a-Service (FaaS) platforms, allows developers to focus on the software engineering aspects of their services without managing the underlying infrastructure. These platforms rely on stateless functions that are triggered by events, making them a common choice for workflows and function composition. However, despite their advantages, serverless workflows often require developers to meet provider-specific requirements, leading to portability challenges and vendor lock-in.

Previous work has attempted to address these limitations. The QuickFaaS project demonstrated the importance of standardizing function definitions across platforms to create a uniform programming model. Building on this, the OmniFlow project introduced a Domain-Specific Language (DSL) that enables developers to define serverless workflows in a provider-agnostic manner, allowing them to be reused across different cloud environments without modification.

This work extends OmniFlow by introducing additional capabilities that enhance serverless workflow execution and function composition. The proposed enhancements include control flow-based workflow execution for repetitive tasks, enabling the definition of iterations within their workflows without relying on provider-specific construct. In addition, it also introduces support for parallel execution, allowing workflows to scale efficiently. By leveraging parallel processing, serverless applications can execute independent tasks concurrently, improving performance and reducing execution time. Additionally, this research explores cross-cloud function composition, ensuring that workflows can seamlessly integrate functions across multiple cloud providers, to mitigate vendor lock-in and allow developers to optimize performance by leveraging the strengths of different platforms while maintaining a unified workflow definition.

The proposed enhancements provide a more flexible, scalable, and portable approach to serverless workflow orchestration, enabling developers to build complex workflows that are not constrained by the limitations of individual cloud providers.

Keywords: FaaS, Function Composition, Cloud-agnostic Architecture

Resumo

A computação *serverless*, em particular as plataformas *Function-as-a-Service* (FaaS), permite que programadores se concentrem nos aspetos de engenharia de software, sem a necessidade de gerir a infraestrutura subjacente. Estas plataformas baseiam-se em funções sem estado desencadeadas por eventos, tornando-as uma escolha comum para fluxos de trabalhos e composição de funções. No entanto, apesar das suas vantagens, fluxos de trabalhos *serverless* exigem o cumprimento de requisitos específicos de cada fornecedor, levando a desafios de portabilidade e dependências no fornecedor.

Trabalhos anteriores tentaram abordar estas limitações, projeto QuickFaaS demonstrou a importância da uniformização da definição de funções entre diferentes plataformas para criar um modelo de programação uniforme. Com base nesse trabalho, o projeto OmniFlow introduziu uma Linguagem Específica de Domínio que permite aos programadores definirem fluxos de trabalhos de forma agnóstica ao fornecedor, possibilitando a sua reutilização em diferentes ambiente cloud sem necessidade de modificação.

Este trabalho acrescenta ao OmniFlow capacidades que melhoram a execução de determinados fluxos de trabalhos e composição de funções. As melhorias propostas incluem a execução de fluxos de trabalhos baseados em tarefas repetitivas, permitindo a definição de iterações nos mesmos sem dependerem de construções específicas de cada fornecedor. Além disso, introduz suporte para execução paralela, permitindo os mesmos possam escalar de forma eficiente. Ao aproveitar o processamento paralelo, as aplicações podem executar tarefas independentes em simultâneo, melhorando o desempenho e reduzindo o tempo de execução. Adicionalmente, este trabalho explora composição de funções entre múltiplos fornecedores, garantindo que os fluxos de trabalhos possam ser integrados em diferentes plataformas cloud sem modificação. Esta abordagem contribui para mitigar a dependência de fornecedor e otimização no desempenho aproveitando os pontos fortes de diferentes plataformas, enquanto mantêm uma definição unificada do fluxo de trabalho.

As melhorias propostas proporcionam uma abordagem mais flexível, escalável e portátil para a orquestração de fluxos de trabalhos *serverless*, permitindo a construção de fluxos de trabalhos complexos sem estarem limitados pelas restrições de fornecedores individuais.

Palavras-chave: FaaS, Composição de funções, Arquitetura agnóstica na nuvem

Contents

List of Figures xiii

Listings xv

1 Introduction 1

1.1 Contributions 2

1.2 Document outline 2

2 Background and Related Work 5

2.1 OmniFlow 5

2.2 Cloud Providers 7

2.2.1 Google Cloud Platform 7

2.2.2 Amazon Web Services 10

2.3 Related Work 13

2.3.1 FaaSFlow 13

2.3.2 Triggerflow 13

2.3.3 Serverless Workflow 14

2.3.4 Temporal Platform 14

2.3.5 FaaSR 14

2.3.6 Code 14

3 Proposed Solution 17

3.1 Introduction 17

3.2 Extending OmniFlow 19

3.2.1 Domain Specific Language 19

3.2.2 Renderer 22

3.3 Implementation Details 24

3.3.1 Google Renderers 29

3.3.2 Amazon Renderers 31

3.3.3 Node Renderer Strategy 34

4 Evaluation 37

4.1 Overview 37

4.2 Rendering Process 38

4.2.1 Independent 38
4.2.2 Variables 38
4.2.3 Binary Conditions 38
4.2.4 Multiple Decisions 38
4.2.5 Iteration 39
4.2.6 Parallel 39
4.2.7 Parallel with Iteration 39
4.3 Results 39
4.3.1 Independent 40
4.3.2 Variables 40
4.3.3 Binary Conditions 40
4.3.4 Multiple Decisions 42
4.3.5 Iteration 42
4.3.6 Parallel 42
4.3.7 Parallel with Iteration 44
4.3.8 Improvements 46
4.4 Case Study Sentiment Classification Workflow 46

5 Conclusion **49**
5.1 Future work 49

Bibliography **51**

Appendices

A Example of a calculation workflow using OmniFlow DSL **53**

B Amazon Web Services map workflow **55**

C Amazon Web Services iteration workflow **57**

D Amazon Web Services parallel workflow **59**

E Amazon Web Services parallel iteration workflow **61**

F Sentiment classification workflow **63**

List of Figures

3.1 Original abstract workflow definition 18
3.2 Main OmniFlow components 19
3.3 Added steps to the original abstract workflow definition 20
3.4 OmniFlow step model class diagram 25
3.5 OmniFlow node renderer class diagram 26

4.1 Independent workflow benchmark original vs new implementation 40
4.2 Variables workflow benchmark original vs new implementation 41
4.3 Binary condition workflow benchmark original vs new implementation . . . 41
4.4 Multiple decisions workflow benchmark original vs new implementation . . 42
4.5 Iteration forEach workflow variation benchmark 43
4.6 Iteration range workflow variation benchmark 43
4.7 Parallel with one branch workflow benchmark 44
4.8 Parallel with multiple branch's workflow benchmark 44
4.9 Parallel iteration forEach workflow variation benchmark 45
4.10 Parallel iteration range workflow variation benchmark 45
4.11 Sentiment Classification Process represented with OmniFlow DSL 47

Listings

2.1 OmniFlow workflow definition 6
2.2 OmniFlow step definition 6
2.3 Google Cloud Platform workflow definition 8
2.4 Google Cloud Platform iteration workflow 8
2.5 Google Cloud Platform parallel workflow 9
2.6 Google Cloud Platform parallel iteration workflow 9
2.7 Amazon Web Services workflow definition 10
2.8 Amazon Web Services Map definition 11
2.9 Amazon Web Services Parallel definition 11
2.10 Amazon Web Services combined iteration and parallel definition 12
3.1 Range based iteration example 19
3.2 Collection based iteration example 20
3.3 Parallel execution example 21
3.4 Parallel iteration example 22
3.5 Node Interface 24
3.6 Workflow and Step model 24
3.7 Iteration step context 25
3.8 Parallel and Branch step context 25
3.9 Node Renderer 26
3.10 Indented Node Renderer 27
3.11 Google Rendering Context 27
3.12 Amazon Rendering Context 28
3.13 Google Renderer 28
3.14 Amazon Renderer 28
3.15 Google Branch Node Renderer 29
3.16 Google Parallel Node Renderer 29
3.17 Google Iteration Node Renderer 30
3.18 Amazon Branch Node Renderer 31
3.19 Amazon Parallel Node Renderer 32
3.20 Amazon Iteration Range Loop 32
3.21 Amazon Iteration Node Renderer 33
3.22 Node Renderer Strategy 34

3.23 Google Branch Strategy 35
4.1 Initialization of input variables 46
4.2 Parallel iteration and API call for sentiment classification 47
4.3 Handling of sentiment classification results 48
A.1 OmniFlow simple calculation workflow 53
B.1 Amazon Web Services map workflow 55
C.1 Amazon Web Services iteration workflow 57
D.1 Amazon Web Services parallel workflow 59
E.1 Amazon Web Services parallel iteration workflow 61
F.1 Sentiment classification workflow 63

1 Introduction

Serverless computing has transformed the way developers design and deploy cloud applications, offering scalability and cost efficiency by abstracting infrastructure management [4, 8]. However, despite its benefits, serverless platforms present challenges in workflow composition, especially when building complex applications that require advanced function orchestration [6]. Most cloud providers offer proprietary solutions for defining and executing serverless workflows, leading to vendor lock-in and reduced portability.

To address these challenges, various frameworks and platforms have been developed. These solutions mostly fall into two categories, either a runtime environment that work as replacements for the cloud providers, such as FaaSFlow and Triggerflow. Or a provider of an agnostic abstraction layer to simplify migration and deployment between different cloud providers, like FaaSR and Serverless Workflow.

OmniFlow [2] is a cloud-agnostic framework that fall in the category of a provider of an agnostic abstraction layer, by providing a unified Domain-Specific Language (DSL) that abstracts provider-specific workflow definitions. It allows developers to define serverless workflows in a generic way, which OmniFlow then converts into the corresponding definitions for different cloud platforms. However, the current implementation of OmniFlow is limited in that it primarily supports sequential execution and conditional branching, lacking key features such as iteration and parallel execution.

Such features are necessary in data processing pipelines, event drive architecture, and now also applied to AI workloads [9]. For example, an AI pipeline may need to iterate over thousands of images to run inference on a pretrained model, aggregate predictions and then further processing as model retraining. By adding these features to OmniFlow this works enables such scenarios to describe these workflows as portable and cloud-agnostic.

This dissertation proposes an extension to OmniFlow by introducing support for *iterations* and *parallel execution* within its DSL. These features are crucial for improving the efficiency of serverless workflows, enabling dynamic task orchestration, reducing execution time, and handling large-scale computations more effectively. The work involves modifying

the OmniFlow DSL to support these new keywords, implementing a rendering mechanism to translate them into cloud-specific workflow definitions, and evaluating the extended system's performance across different cloud providers.

1.1 Contributions

This research extends the OmniFlow DSL by introducing support for iteration and parallel execution, addressing a key limitation in existing serverless workflow orchestration solutions. By analyzing current function composition techniques across different cloud providers, this work identifies the challenges of interoperability and vendor lock-in that hinder the development of portable and scalable workflows.

To overcome these challenges, the proposed extension enhances OmniFlow by integrating new keywords that allow workflows to execute iterative processes and parallel tasks efficiently. The implementation of these features is accompanied by a translation mechanism that ensures compatibility with provider-specific workflow definitions, particularly for AWS Step Functions and Google Cloud Workflows.

Beyond the design and implementation, this work also evaluates the extended DSL by testing real-world workflow scenarios, assessing its correctness, performance, and scalability. Through these contributions, this research provides a more flexible and expressive approach to serverless workflow orchestration while maintaining cloud-agnostic portability.

The contributions and methodologies detailed in this dissertation were also compiled and published in a scientific paper. This paper was successfully peer-reviewed and presented at CISTI 2025 [7], the Iberian Conference on Information Systems and Technologies.

Since this is a continuation of an open source work, all the corresponding components of this work can be found publicly available in a GitHub repository [14]. The repository documentation on how to use the library is also provided in Markdown format. As a continuation of an open-source initiative, OmniFlow invites contributions from the wider developer community, enabling improvements to the framework itself as well as its documentation, implementation, tests, and benchmarks

1.2 Document outline

The remainder of this document is organized into several chapters, each addressing a key aspect of the research.

Chapter 2 provides background information on serverless computing, function composition, and workflow orchestration. It introduces the fundamental concepts necessary to understand the challenges of building cloud-agnostic workflows and explores existing

solutions, including proprietary approaches such as AWS Step Functions and Google Cloud Workflows, and research efforts, such as FaaSFlow and Triggerflow, highlighting their advantages and limitations. Also provides an overview of OmniFlow, the framework extended in this work.

Following this, Chapter 3 details the extension to the OmniFlow DSL to support iteration and parallel execution and combined parallel iteration workflows. It describes the changes introduced in the abstract workflow model, the new DSL keywords and the renderer changes needed to translate this new constructs into AWS and GCP definitions.

Chapter 4 presents the evaluation of the extended DSL through benchmark workflow. It describes the rendering process, going into the implementation challenges and comparing the performance between the original and extended version.

Finally, Chapter 5 summarizes the contributions, highlights the achieved scalability improvements and outlines possible future work enhancements to OmniFlow.

2 Background and Related Work

Serverless computing has changed the way developers create and manage workflows in the cloud, offering simplicity and scalability. However, this ease of use often comes with the drawback of vendor lock-in, as workflows are tightly coupled to the proprietary solutions of specific cloud providers. To mitigate this challenge, tools like OmniFlow have been introduced, to assist developers to design cloud-agnostic workflows that can be seamlessly deployed and managed across multiple cloud platforms.

2.1 OmniFlow

OmniFlow is a library that facilitates developers on the creation, deployment and possible migration of workflows between different cloud provides with the intention of reduction the vendor lock in with any specific cloud provider. It achieves this by creating a cloud-agnostic DSL that after define gets translated to one of the supported serverless functions cloud providers. These workflows are a set of serverless functions that are executed in cloud provider runtime with the intent of performing specific tasks. To be able to support various kinds of tasks these cloud providers support multiple functionalities in each step of a defined workflow such as conditionals steps, receiving inputs and output passing between steps, variable assignment, iterations and parallel steps.

The current OmniFlow implementation supports AWS and GCP cloud providers and some of their workflows functionalities such as conditionals, inputs and outputs in each step and variable assignment, lacking support for iteration steps and parallel steps. Although lacking support for some of the features, the current implementation already enables developers to be able to create simple sequential workflows which don't required going through any collections.

Workflows in OmniFlow DSL are defined in Kotlin language and have the following keywords in the root: a `name`, a `description`, an `input`, a list of `steps`, and a `result`. The `name` and `description` are merely descriptive, while the `input` designates the

variable name where the arguments will be stored. The `steps` keyword defines the set of instructions to be executed, and the `result` denotes the variable where the workflow's output will be stored, as show in Listing 2.1. As previous state the `steps` keyword takes a set of instructions more specifically it takes another keyword which is `step`, similar to the workflow the `step` also has a few keywords namely `name` which can be referenced in other steps, `description` which is descriptive, and `context` which defines the step behavior, such as **call**, **assign** and **switch**.

An example of a **call** step which makes an HTTP request is shown in Listing 2.2. For this **call** step a few additional keywords are needed namely `method`, `host`, `path` and `result`. These keywords specify the HTTP request method, the destination, the URI path, and the variable where the result of the corresponding request should be saved. Optionally other keywords may be defined, such as `authentication`, `body`, `headers`, `query` and `timeoutInSeconds`, which define other specifications for an HTTP request. Other step types follow a similar syntax, with additional specific keywords for each one. A full example where an addition of two random numbers followed by division by a third random number is shown in Listing A.1. This example shows most of the keywords mentioned before.

Since this work intends to extend this library, by adding support to the missing features namely iteration and parallel steps, a more detail of the proposed extensions will be given in Chapter 3.

```
1  {
2    name("calculatorWorkflow")
3    description("Calculator example")
4    input("args")
5    steps(
6      ...
7    )
8    result("divResult")
9  }
```

Listing 2.1: OmniFlow workflow definition

```
1  step {
2    name("Sum")
3    description("Sum 2 random numbers")
4    context(
5      call {
6        method(GET)
7        host("r1ro8xa7y8.execute-api.us-east-1.amazonaws.com")
8        path("/default/calculator")
9        query(
10         "number1" to variable("a"),
11         "number2" to variable("b"),
12         "op" to value("add")
13       )
14       result("sumResult")
15     }
16   )
17 }
```

Listing 2.2: OmniFlow step definition

2.2 Cloud Providers

Cloud computing became increasingly popular in recent years, due to its ability to simplify computation and storage needs while reducing costs and complexity, when compared to traditional computing, by allowing users to offload the responsibilities of handling the infrastructure, to public cloud providers. Instead of purchasing and maintaining their own hardware, users can rely on these public cloud providers to handle storage and computation. This way, they only pay for what they use, avoiding the high upfront cost in hardware and additional cost in specialized teams to maintain this traditional systems. Additionally, cloud services also give the option for automatic scale up to meet demands, saving users the trouble of doing this themselves. Several major cloud providers, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure, offer a variety of cloud computing services. Among these services, there are three clear categories, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Each mentioned categories has different levels of responsibility the user and the cloud provider has. In IaaS, users manage the full infrastructure resources. For PaaS, users manage applications and their configurations, while the provider handles the underlying OS, middleware, and runtime environment. While SaaS users simply utilize the software and the provider handles everything from infrastructure to application maintenance. A more granular service that PaaS can also offer is Function as a service (FaaS) where users write code and define functions the cloud provider manages everything else, including scaling, patching, and server management. FaaS services also support function composition, enabling users to chain functions together creating workflows. Since this work focuses on extending an existing library to support advanced function composition features, specifically iterations and parallel functionalities, we will focus on FaaS more specifically in AWS Step Functions and GCP Workflows. To illustrate these features, examples will be shown for both in the following section. For iteration functionality, a simple summation example will be used, while parallel functionality will be shown through two simultaneous HTTP requests.

2.2.1 Google Cloud Platform

Google Cloud Platform (GCP) supports function composition with the Workflows [5], which allows users to create and maintain serverless functions using YAML Ain't Markup Language (YAML). The workflows are defined with a number of specific keywords starting with the `main` which defines the main workflow to be executed, it is also possible to define sub workflows in the same level as `main` keyword as illustrate in the Listing 2.3.

Both the main and workflows have then defined a list of steps to be executed. These steps can have different behaviors such as assigning values to variables (**assign**), iterate through collections (`for`), and make HTTP requests (**call**). Workflows in GCP support advanced function composition features such as iterations and parallel flows. These functions are necessary for users to be able to create dynamic and efficient workflows. The following examples demonstrate GCP's capabilities in these areas.

```
main:
  steps:
    ...
subworkflow_name:
  steps:
    ...
```

Listing 2.3: Google Cloud Platform workflow definition

2.2.1.1 Iterations in GCP

GCP enables developers to iterate over collections with a straightforward syntax, as shown in Listing 2.4 which illustrates the use of the keyword `for` to iterate through the collection of numbers defined in the **assign** step, updating the value of the variable `sum` in each step by adding the previous total with the current value of the iteration.

```
main:
  steps:
    - init:
        assign:
          - sum: 0
          - list: [1, 2, 3, 4, 5]
    - summation:
        for:
          value: v
          in: ${list}
          steps:
            - sumStep:
                assign:
                  - sum: ${sum + v}
    - logSum:
        call: sys.log
        args:
          data: ${sum}
```

Listing 2.4: Google Cloud Platform iteration workflow

2.2.1.2 Parallel in GCP

GCP also supports parallel execution using the `parallel` keyword, allowing multiple branches to execute simultaneously. Listing 2.5 demonstrates a parallel workflow where two HTTP requests using the **call** keyword is used to retrieve both user and notification data concurrently. The keyword `shared` under the `parallel` allows variables to be shared between branches.

```
main:
  params: [args]
  steps:
    - init:
        assign:
          - user: {}
          - ntfy: {}
    - parallel_branches:
        parallel:
          shared: [user, ntfy]
          branches:
            - getUser:
                steps:
                  - getUserCall:
                      call: http.get
                      args:
                        url: ${"https://example.com/users/" + args.userId}
                      result: user
            - getNotification:
                steps:
                  - getNotificationCall:
                      call: http.get
                      args:
                        url: ${"https://example.com/ntfy/" + args.ntfy}
                      result: ntfy
    - logUser:
        call: sys.log
        args:
          data: ${user}
    - logNotification:
        call: sys.log
        args:
          data: ${ntfy}
```

Listing 2.5: Google Cloud Platform parallel workflow

2.2.1.3 Combined Iteration and Parallel Execution in GCP

GCP also enables workflows that combine iteration and parallel execution, as illustrated in Listing 2.6. This example demonstrates how multiple HTTP requests can be executed in parallel while iterating over a collection of post IDs, the use of the keyword `for` inside the keyword `parallel` allows each step of the iteration to be executed in parallel, as well the `shared` keyword which enables each parallel iteration assign both `total` and `posts` variables.

```
main:
  params: [args]
  steps:
    - init:
        assign:
          - total: 0
          - posts: [1, 2, 3, 4]
    - parallel_branches:
        parallel:
          shared: [total, posts]
          for:
```

```
12              value: postId
13              in: ${posts}
14              steps:
15                - getPostCommentCount:
16                    call: http.get
17                    args:
18                      url: ${"https://example.com/postComments/" + postId}
19                    result: numComments
20                - add:
21                    assign:
22                      - total: ${total + numComments}
```

Listing 2.6: Google Cloud Platform parallel iteration workflow

2.2.2 Amazon Web Services

Amazon Web Services (AWS) provides function composition through its Step Functions [1], which allows developers to deploy workflows using a JSON-based DSL. This DSL defines some keywords at the root level, namely `StartAt` which defines the name of the first step to be executed and the `States` that defines the list of steps to be executed as shown in the Listing 2.7. All steps have another set of keywords, for example `Type` which defines the type of the step, `Next` defines next run to be executed in the workflow, and `End` that defines if the specific step is the last step in the workflow. There are many additional keywords to define a step, some dependent on its type. Some of the types of steps that AWS Step Functions supports are iterations through the `Map` state and parallel execution through the `Parallel` state. These features enable developers to create robust and scalable workflows. However, there are important nuances to consider, particularly regarding shared state in iterations.

```
1  {
2    "StartAt": "DefineInput",
3    "States": {
4      "DefineInput": {
5        "Type": "Pass",
6        "Parameters": {
7          "Posts": [1, 2, 3, 4],
8          "Results": []
9        },
10       "Next": "LastStep"
11     },
12     "LastStep": {
13       "Type": "Pass",
14       "End": true
15     }
16   }
17 }
```

Listing 2.7: Amazon Web Services workflow definition

2.2.2.1 Iterations in AWS

AWS enables iteration over a collection using the 'Map' state, as demonstrated in Listing 2.8. A complete version can be seen in Listing B.1. The `Map` state allows each iteration to execute a defined sub workflows, with individual elements from the collection passed as input. While this approach is effective for isolated operations on collection items, the `Map` state does not allow iterations to share or update a common state. Each iteration is separated, which can be a limitation when an aggregated result or shared data is required.

```
1  {
2    "StartAt": "ProcessItem",
3    "States": {
4      "ProcessItem": {
5        "Type": "Map",
6        "ItemsPath": "$.Posts",
7        "Iterator": {
8          "StartAt": "Sum",
9          "States": {
10           "Sum": { "Type": "Pass", "End": true}
11         }
12       },
13       "End": true
14     }
15   }
16 }
```

Listing 2.8: Amazon Web Services Map definition

To address scenarios that require shared state between iterations, an alternative approach can be used, as shown in Listing 2.9 .A complete version can be seen in Listing C.1. This workflow explicitly manages iteration using a combination of `Pass` and `Choice` states, allowing shared variables to be updated across iterations. In this example, the workflow iterates over a list of numbers, maintaining a running sum that is updated at each step. By leveraging states like `Pass` and mathematical operations, the workflow can mimic a traditional loop with shared state.

This alternative approach, while more complex than the `Map` state, provides flexibility for workflows requiring cumulative or interdependent processing of collection elements.

2.2.2.2 Parallel in AWS

Parallel execution in AWS is implemented using the `Parallel` state, which enables multiple branches to execute concurrently. Listing D.1 illustrates a workflow with two branches, one retrieving user data and the other fetching notification data simultaneously. Each branch operates independently, and their outputs are aggregated once all branches complete execution.

```
1  {
2    "StartAt": "ParallelWork",
3    "States": {
```

```
4     "ParallelWork": {
5       "Type": "Parallel",
6       "Branches": [
7         {
8           "StartAt": "GetUser",
9           "States": {
10            "GetUser": { "Type": "Pass", "End": true }
11          }
12        },
13        {
14          "StartAt": "GetNotification",
15          "States": {
16            "GetNotification": { "Type": "Pass", "End": true }
17          }
18        }
19      ],
20      "End": true
21    }
22  }
23 }
```

Listing 2.9: Amazon Web Services Parallel definition

2.2.2.3 Combined Iteration and Parallel Execution in AWS

AWS workflows can combine iteration and parallel execution, as shown in Listing 2.10. A complete version can be seen in Listing E.1. This example iterates over a collection of posts, where each iteration processes a post in parallel by executing multiple HTTP requests. Although the `Map` state facilitates parallel operations for each element, it retains the limitation of not supporting shared state between iterations. As a result, aggregations or cumulative computations in this context must be handled after the `Map` state or by using custom approaches similar to the one in Listing C.1. Another approach to the aggregation or reduce step would be the use of a serverless function that would receive the aggregated results and output the reduced result.

```
1  {
2    "StartAt": "MapPosts",
3    "States": {
4      "MapPosts": {
5        "Type": "Map",
6        "ItemsPath": "$.Posts",
7        "Iterator": {
8          "StartAt": "ParallelRequests",
9          "States": {
10           "ParallelRequests": {
11             "Type": "Parallel",
12             "Branches": [
13               {
14                 "StartAt": "GetComments",
15                 "States": {
16                   "GetComments": { "Type": "Pass", "End": true }
17                 }
18               },
19               {
20                 "StartAt": "GetLikes",
```

```
21                  "States": {
22                    "GetLikes": { "Type": "Pass", "End": true }
23                  }
24                }
25              ],
26              "End": true
27            }
28          }
29        },
30        "End": true
31      }
32    }
33  }
```

Listing 2.10: Amazon Web Services combined iteration and parallel definition

2.3 Related Work

In this section, we review relevant related work on function composition in functions as a service (FaaS) platforms to understand the architecture, tools and applications that have been created to ease the use of these platforms.

2.3.1 FaaSFlow

A worker-side workflow scheduling pattern is proposed for serverless workflow execution [15]. Unlike traditional master-side scheduling, where workflow state and function management reside in a central node, this approach deploys a per-worker workflow engine on each node to handle function state and trigger local tasks. Additional, FaaSFlow also proposes an Adaptive Storage Library that leverages local memory for data transfer between functions which are running in the same worker nodes. Our work, in comparison, centers around easing the creation and migration process of workflows across cloud providers and leaving the runtime to them.

2.3.2 Triggerflow

In comparison with other systems that focus on short running workflows or have overheads from synchronize sizable workflows, Triggerflow [10] builds on top of Knative Eventing and Kubernetes technologies to build a platform for event based orchestration of serverless workflows with the focus on scalability and extensibility. It adopts an Event-Condition-Action (ECA) architecture with stateful triggers which can aggregate, filter, process and route events. Additionally also supports high-volume of events and autoscaling on demand. It contrasts with our goal, to support deployment across multiple cloud FaaS providers, since Triggerflow creates an open system with a generic model which aims to be an extensible replacement for cloud providers systems.

2.3.3 Serverless Workflow

Serverless Workflow provides a common syntax and tools for creating and managing serverless workflows across different environments [3]. It includes a DSL defining core syntax and semantics, automated tests for specification compliance, SDKs for workflow creation and validation, and a runtime for execution. Additionally, it offers utilities for development and debugging. Its primary focus is on simplifying workflow transitions between cloud providers to reduce vendor lock-in, relying on cloud providers support. Our focus is to extend OminFlow, a solution that uses a Kotlin DSL, that renders the workflow defined in its DSL directly to the target FaaS provider workflow definition.

2.3.4 Temporal Platform

The Temporal Platform [13] consists of the Temporal Server and Worker Processes, which together enable workflow execution. The server oversees tasks, records their completion, and maintains a persistent event history. Worker Processes poll task queues, execute tasks, and return results. A Temporal Application runs multiple concurrent Workflow Executions, each with its own local state. The platform serves as a generic workflow manager, replacing cloud providers by handling service calls, error management, and scaling within its own clusters. In contrast, we want to extend a library that provides an agnostic workflow DSL for deploying and migrating workflows across cloud providers, reducing vendor lock-in.

2.3.5 FaaSR

FaaSr [12] is designed for FaaS platforms, this system simplifies running R functions and workflows in the cloud, triggered by events like timers or code updates. It integrates with GitHub Actions, AWS Lambda, IBM Cloud, and OpenWhisk, providing functions for data transfer via S3-compatible storage. It also defines a provider-agnostic schema for sequential workflows. FaaSr [12] shares similar goals but focuses on sequential execution, whereas our goal is to support parallelism and iterations across cloud providers while leaving data transfer decisions to the developer.

2.3.6 Code

CODE [11] is a framework that aims to make easier the deployment of serverless functions across various cloud platforms. It provides an agnostic DSL that focus in reducing the amount of repetitive settings for each location and provider, similar to technologies like Terraform or Serverless Framework use. This approach reduces the amount of repetitive setup, which makes easier the deployment of such applications.

One of CODE's primary features is the integration with the storage systems of both AWS and GCP using the library GoStorage. With this library, developers can swap between

storage providers without having to rewrite code specifically for each SDK. CODE also automates the process of transferring deployment packages between regions and providers, following a "code once, deploy everywhere" principle. This eliminates much of the manual work associated with setting up serverless workflows across multiple locations.

Our library on the other only focus on creation of a DSL agnostic of the cloud providers that aims to facilitate the deployment and migration between cloud providers.

3 Proposed Solution

Building on the background presented in Chapter 2 this chapter describes the proposed solution. While the previous Chapter introduced serverless workflows and existing solutions, here we focus on the design and implementation of extending OmniFlow.

3.1 Introduction

Serverless computing has changed the way developers build and deploy cloud applications, by offering the capability of abstracting away infrastructure management. A key application of this paradigm is the creation of complex business logic through workflows, which compose and orchestrate individual, event-triggered functions. However, this approach often leads to a significant challenge: major cloud providers such as Amazon Web Services (AWS) and Google Cloud Platform (GCP) implement proprietary and often incompatible solutions for defining and executing these serverless workflows. This tight coupling to a provider's ecosystem creates portability challenges and results in vendor lock-in, restricting the ability to migrate or leverage services across different cloud environments.

The OmniFlow library, provides a way to compose and deploy serverless workflows across multiple cloud providers, such as Amazon and Google cloud platforms. Its main purpose is to reduce vendor lock in by defining a cloud-agnostic Domain Specific Language (DSL) that abstracts the platform specific implementation details. This DSL allows users to define workflows in a generic format, which OmniFlow then renders into the specific workflow definitions required by each cloud provider. At the core OmniFlow is an abstract workflow model that represents workflows as a sequence of steps, Figure 3.1 illustrates the original abstract workflow definition used in OmniFlow, which represents workflows as a composition of steps that can receive inputs and can return outputs. Each of these steps can perform actions like making HTTP requests, assigning variables or execute conditional logic. Once the workflow is defined with the DSL, the renderer handles the task of translating it into the correct target syntax, such as JSON for AWS step functions

or YAML for GCP workflows. Lastly, the library also includes tooling for deploying and managing these workflows, on the chosen cloud platform without requiring to manually handle configuration or deployment steps.

However, the initial version of OmniFlow only handles basic workflow functionalities namely sequential execution, conditional branching and variable assignments, lacking more complex features like iterations or parallel execution. This limitation means that workflows requiring repeating tasks over data or tasks that could benefit from running in parallel are not yet supported. Both of these features would be a requirement for building scalable and dynamic serverless applications, where reduced execution time and processing large datasets efficiently are essential.

This work extends OmniFlow by enabling support for both of these features, namely iteration and parallel steps, enabling the creation of such applications. It will allow workflows to handle repetitive tasks over collections and independent step execution. Therefore, increasing the scope of workflows that OmniFlow can support and make it more adaptable to modern serverless requirements. The next section will explain how these features were incorporated into OmniFlow. It will outline the necessary changes to the abstract workflow model, additions to the DSL and updates required to the render and deployment process.

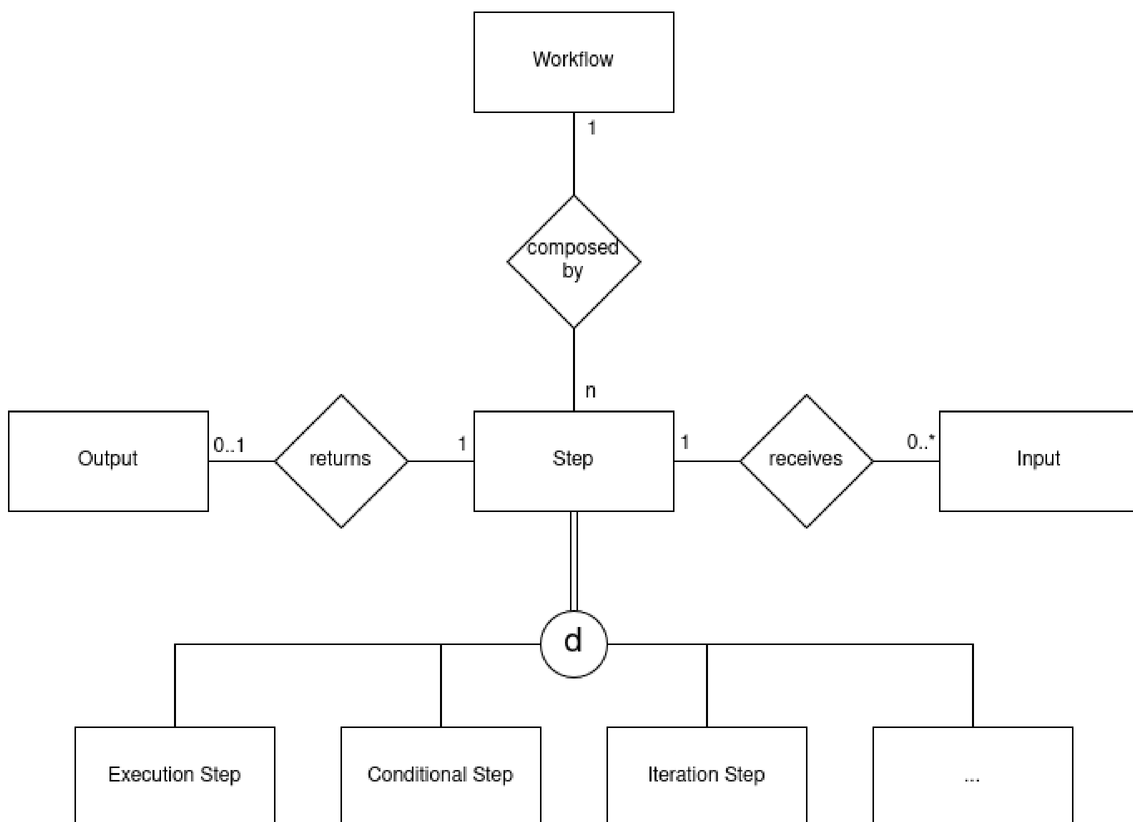


Figure 3.1: *Original abstract workflow definition*

3.2 Extending OmniFlow

As mentioned before, OmniFlow library is built by three fundamental components, illustrated in Figure 3.2, that help abstract different cloud providers. It's domain specific language to define abstract workflows, the renderer which translates those abstract workflows definitions to cloud specific ones and a tool to deploy those translated workflows to the specific cloud providers. Since this works plans to add new functionality to the existing library it only needs to modify two of these three components namely the DSL and the renderer, since after the translation is done the deployment doesn't change.

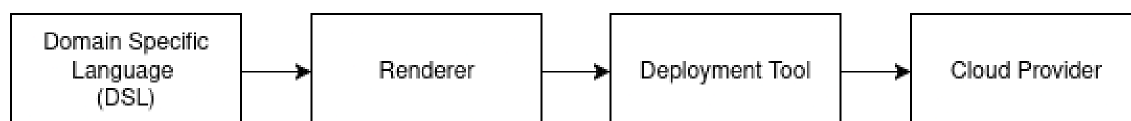


Figure 3.2: *Main OmniFlow components*

3.2.1 Domain Specific Language

The current implementations of OmniFlow DSL supports sequential workflows, conditional branching and variable assignment. However, more advance workflows often require dynamic iterations over collections and concurrent execution of tasks. These features are essential for enabling efficient and scalable serverless applications. By adding support for *iterations* and *parallel execution* into the DSL, the extended library allows users to build workflows capable of handling tasks such as batch processing, data aggregation and distributed computing. Figure 3.3 shows the added steps for iteration and parallel execution in the extended abstract workflow definition.

3.2.1.1 Iteration Execution

The extended DSL introduces a new keyword `iteration` that allows the capability of iterating through either a numeric range or a collection. Within this step the following keywords can be used, `value` which defines the variable that stores the values in each iteration, `steps` to define the sequence of steps to be executed in each iteration, and either `range` or `forEach` which allow the choice between iterating over a numeric range, or a collection.

The Listings 3.1 and 3.2 show a summation over a numeric range using `range` keyword and the summation over a collection (e.g., a list or hashmap keys) using `forEach` keyword.

```
1  iteration {
2    value("key")
3    range(1, 9)
4    steps(
5      step {
6        name("SumStep")
```

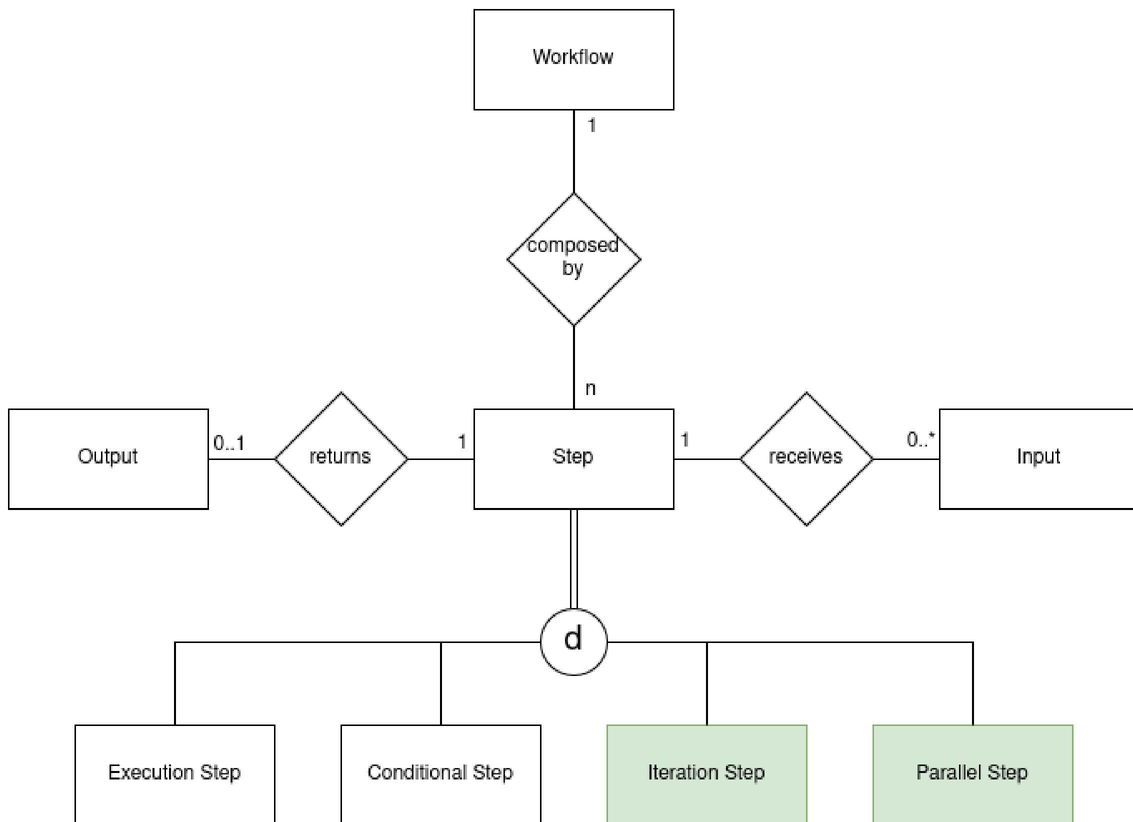


Figure 3.3: *Added steps to the original abstract workflow definition*

```
7          description("Add key value to result variable")
8          context(
9            assign {
10             variable("result" equal sum(variable("result"), variable("key")))
11           }
12         )
13       }
14     )
15   }
```

Listing 3.1: Range based iteration example

```
1  iteration {
2    value("key")
3    forEach(variable("listString")
4    steps(
5      step {
6        name("SumStep")
7        description("Add key value to result variable")
8        context(
9          assign {
10           variable("result" equal sum(variable("result"), variable("key")))
11         }
12       )
13     }
14   )
15 }
```

Listing 3.2: Collection based iteration example

3.2.1.2 Parallel Execution

The extended DSL also introduces the keyword `parallel` that allows the use of multiple `branches` that are executed in parallel. Each `branch` has a `name`, and its own sequence of `steps`. Listing 3.3 shows an example where two HTTP requests are made in parallel, to where the first step retrieves user data and the second one retrieves a notification.

```
parallel {
  branches(
    branch {
      name("branch1")
      steps(
        step {
            name("GetUserCall")
            description("Get User Data")
            context(
            call {
                method(GET)
                host("https://example.com")
                path("/users/1")
                result("user")
            }
            )
        }
      )
    },
    branch {
      name("branch2")
      steps(
        step {
          name("GetNotificationCall")
          description("Get Notification")
          context(
            call {
              method(GET)
              host("https://example.com")
              path("/ntfy/1")
              result("ntfy")
            }
          )
        }
      )
    }
  )
}
```

Listing 3.3: Parallel execution example

3.2.1.3 Parallel Iteration Execution

The `parallel` keyword also allows the use of `iteration` keyword in replace of the `branches` keyword. This makes possible the combination of parallelism with iterations, allowing the executing of each iteration in parallel. An example is shown in Listing 3.4, where the total amount of existing comments in all the posts are added together, with both the `call` and `assign` steps.

```
parallel {
  iteration {
    key("key")
    forEach(variable("listString"))
    steps(
      step {
        name("GetPostCommentCount")
        description("Get Post Comment Count")
        context(
          call {
            method(GET)
            host("https://example.com")
            path("/postComments/" + variable("id"))
            result("numComments")
          }
        )
      },
      step {
        name("Add")
        description("Add number of comments to total")
        context(
          assign {
            variable("total" equal sum(variable("total"), variable("numComments")))
          }
        )
      }
    )
  }
}
```

Listing 3.4: Parallel iteration example

3.2.2 Renderer

Internally, OmniFlow represents workflows as a hierarchical tree, where the root node corresponds to the workflow itself and each child node represents a step. Steps can contain their multiple child nodes, enabling the construction of complex workflows. To traverse this tree, OmniFlow uses a depth-first search algorithm, ensuring that each node is visited in a structured and predictable order. During traversal, the library relies on a strategy pattern to dynamically select the appropriate renderer for each step. The renderer associated with a node is responsible for translating the abstract definition of that step into its provider-specific representation, ensuring that the workflow can be consistently and correctly deployed across different cloud environments. Each provider implementation defines its own set of renderers, that handle the specific syntax and requirements. Alongside these renderers, the implementation also defines its own strategy for matching model classes to renderers, enabling the correct use of the renderer when the abstract workflow elements are transformed during the rendering process. This separation of concerns ensures that the internal model remains provider-agnostic, while the rendering logic remains fully customizable and extensible per cloud environment. Due to this extensibility, introducing previous mentioned functionalities involves adding

new renderers and an accompanying strategy for matching them, while modifications to existing code are only required when additional capabilities need to be incorporated into the library.

3.2.2.1 Iteration Execution

Implementing iteration was something straightforward in GCP, since it natively supports most of the features that we wanted to support with this work. Having native support for both iterating over a range and iterating over a collection, and being able to share data between each iteration. The same was not possible with AWS, although it does support a map function to iterate over a collection it doesn't allow sharing the variables between iterations. Although this limitation exists when going over a range, there's a workaround that becomes possible by doing a manual loop with conditions and jumps and breaking when reaching the end of the range. The same was not possible to be replicated for collection since we wouldn't know the size of the collection in rendering time, making it impossible to have a condition to break the loop.

3.2.2.2 Parallel Execution

For the parallel it was more simple since both AWS and GCP both natively support parallel steps, making the implementation a simple translation without the need for any workarounds.

3.2.2.3 Parallel Iteration Execution

Since parallel iteration is a mixture of both parallel and iteration, what we expect happened, leading to the same issues that appear for iterations for AWS. Implementing the same workaround was also possible, solving the same issue of sharing data between iterations. When it comes to GCP, implementation was straightforward since it also supported parallel iterations.

3.3 Implementation Details

As previously mentioned, OmniFlow models workflows as a hierarchical tree, implemented through a Kotlin interface Node that can have multiple child nodes, as seen in Listing 3.5. This interface is then implemented by multiple classes such as Workflow and Steps, and many others. Since in this work we add new type of steps, we need to understand the internal model so we are able to extended it. Presented in the Listing 3.6 is the definition of a Step in the internal model. As shown in the listing the step has the following properties: a `name` which can be referenced in other steps, `description` to explain the purpose of the step, `type` which defines the step behavior, such as **call**, **assign** and **switch**, `context` which is the data necessary to build each type of step and `next` which allows the developer to select the next step to jump into. The `next` keyword was something added as part of this work, and adds the ability for the developer to choose the next step to be executed explicitly.

```
1  interface Node {
2      fun childNodes(): List<Node>
3  }
```

Listing 3.5: Node Interface

```
1  data class Step(
2    val name: String,
3    val description: String,
4    val type: StepType = StepType.CALL,
5    val context: StepContext,
6    val next: String = "",
7  ) : Node {
8    override fun childNodes(): List<Node> {
9      return listOf(context)
10   }
11 }
```

Listing 3.6: Workflow and Step model

Due to this abstraction, adding new step types at the level of the DSL becomes a matter of adding new step contexts for each new step, as show in the both 3.7 and 3.8 listings. The Figure 3.4 demonstrates a class diagram where the relation between step, its context and these new implementations can be seen. For the iteration step type it contains: `value` that defines the variable which stores the current element in each iteration, `steps` define the sequence of steps executed in each iteration. In each specialization, an extra class with an additional property is defined. The `range` specialization uses the field `range` to define the numeric range to be used. In the case of `forEach` specialization the field `forEachVariable` which defines the collection to iterate over. On the case of the parallel step type, it has two different specializations, one for parallel which has only a list of *branch step context* and another for parallel iteration which instead has an *iteration step context*. In the case of the branch context it is similar to a workflow and has the same `name`, `description` and list of `steps` to be executed.

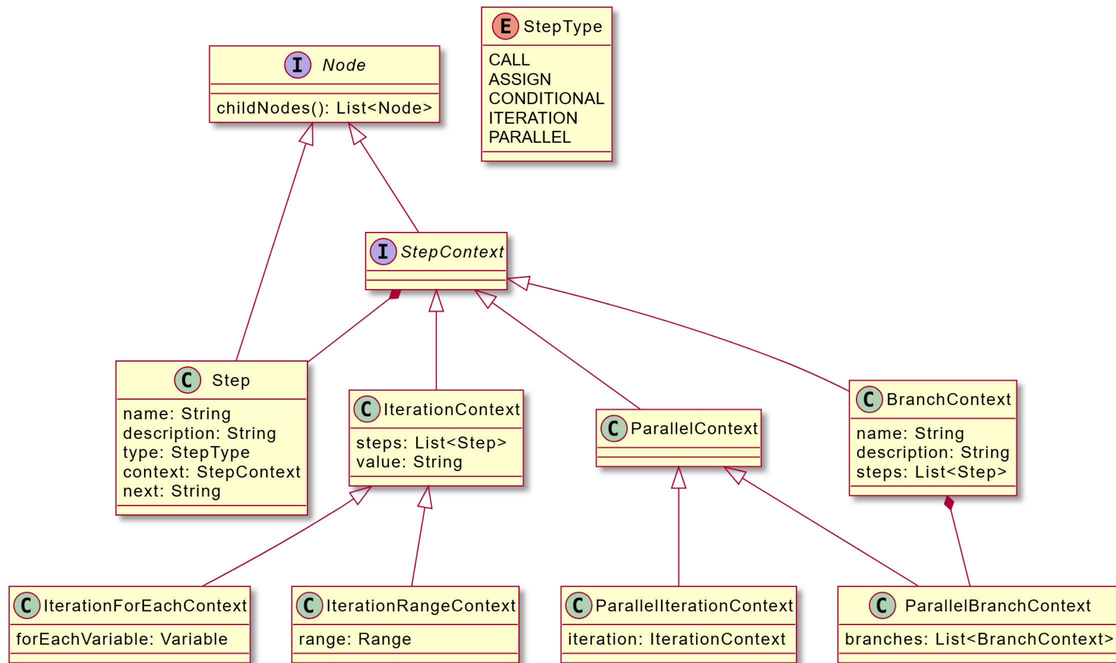


Figure 3.4: *OmniFlow step model class diagram*

```
open class IterationContext(open val value: String, open val steps: List<Step>) :
    StepContext {
  override fun childNodes(): List<Node> {
    return steps
  }
}

data class IterationRangeContext(override val value: String, override val steps:
    List<Step>, val range: Range) :
  IterationContext(value, steps)

data class IterationForEachContext(
  override val value: String,
  override val steps: List<Step>,
  val forEachVariable: Variable
) : IterationContext(value, steps)
```

Listing 3.7: Iteration step context

```
interface ParallelContext : StepContext

data class ParallelBranchContext(val branches: List<BranchContext>) :
    ParallelContext {
  override fun childNodes(): List<Node> {
    return branches
  }
}

data class ParallelIterationContext(val iterationContext: IterationContext) :
    ParallelContext {
  override fun childNodes(): List<Node> {
    return listOf(iterationContext)
  }
}
```

```

data class BranchContext(
  val name: String,
  val description: String? = null,
  val steps: List<Step> = emptyList(),
) : StepContext {
  override fun childNodes(): List<Node> {
    return steps
  }
}
```

Listing 3.8: Parallel and Branch step context

The next step in the rendering process is taking the model created and rendering the corresponding specific provider DSL. To ensure extensability, OmniFlow defines a **NodeRenderer**, shown in Listing 3.9, which corresponds to an interface that needs to be implemented for each pair of cloud provider specific DSL and *Node*, Figure 3.5 shows a class diagram where the relation between a Node and its renderer can be observed. The **NodeRenderer** defines three things, an element to hold the corresponding node to be rendered and two methods, one that gets called beginning of the rendering process for the node and end of the rendering process of a given node. Since the process of rendering follows a depth search first, if starts by calling the **beginRender** for the current node followed by rendering the child nodes and ending the calling *endRender* for the current node, passing a rendering context between all the nodes, so the process is able to keep state between nodes. Although NodeRenderer is of a generic type, the final output we desired is a string so to be able to support that the original implementation also has a specialization of this interface of string type, as seen in Listing 3.10, which then each cloud specific render outputs the specific DSL in YAML for GCP and in JSON for AWS.

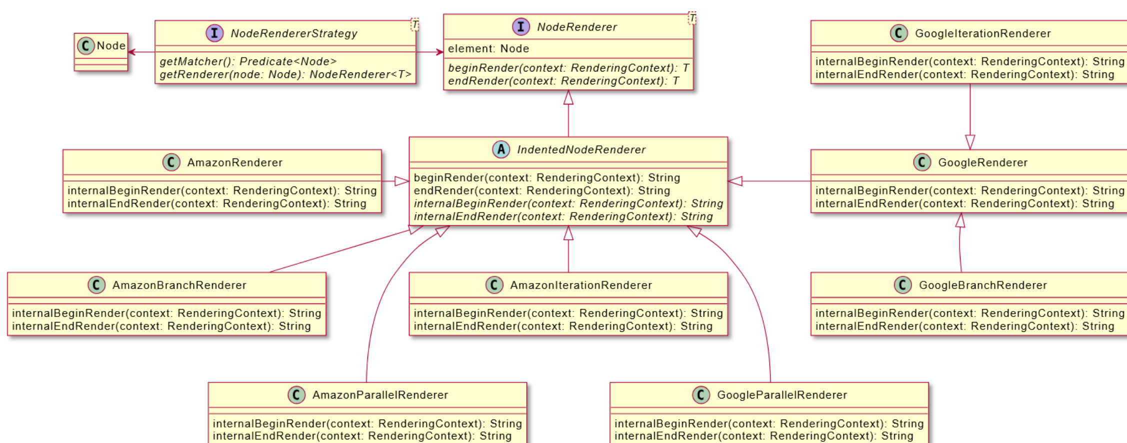


Figure 3.5: *OmniFlow node renderer class diagram*

```
interface NodeRenderer<T> {
  val element: Node

  fun beginRender(renderingContext: RenderingContext): T

  fun endRender(renderingContext: RenderingContext): T
```

```
7 }
```

Listing 3.9: Node Renderer

```
1  abstract class IndentedNodeRenderer : NodeRenderer<String> {
2
3    override fun beginRender(renderingContext: RenderingContext): String {
4      val indentedRenderingContext = renderingContext as IndentedRenderingContext
5      val render = internalBeginRender(indentedRenderingContext)
6      indentedRenderingContext.incIndentationLevel()
7      return render
8    }
9
10   override fun endRender(renderingContext: RenderingContext): String {
11     val indentedRenderingContext = renderingContext as IndentedRenderingContext
12     indentedRenderingContext.decIndentationLevel()
13     return internalEndRender(indentedRenderingContext)
14   }
15
16   protected abstract fun internalBeginRender(renderingContext:
       IndentedRenderingContext): String
17
18   protected abstract fun internalEndRender(renderingContext:
       IndentedRenderingContext): String
19 }
```

Listing 3.10: Indented Node Renderer

With the addition of parallel, the need of being able to support sub workflows inside of workflows became an issue to solve, since inside a parallel branch can be seen as multiple sub workflows that will be executed. To be able to support those in the rendering step, a change to both cloud provider original rendering context was needed. Because a depth first search algorithm is used in the rendering process a stack of rendering contexts became necessary to be able to support sub workflows, since each rendering context saves the state of the given workflow, and it needs to keep it for the whole rendering process. This was done using a companion object, so the same instance of the stack would be accessible in all the rendering context, as shown in Listing 3.11 and 3.12. Additionally, for AWS the need to resolve hosts was also necessary due to the limitation that AWS imposes of using their API gateway to be able to make requests to external services, this is needed so it's possible to reuse the same workflow definition to be able to deploy in both AWS and GCP with only needing to define an additional mapping between a host and an API gateway.

```
1  class GoogleRenderingContext(
2    indentationLevel: Int = 0,
3    stringBuilder: StringBuilder = StringBuilder(),
4    termContext: TermContext = object : TermContext {},
5  ) : IndentedRenderingContext(indentationLevel, stringBuilder, termContext) {
6    companion object {
7      private var innerRenderingContext: MutableList<GoogleRenderingContext> =
         mutableListOf()
8    }
9    ...
```

```
10 }
```

Listing 3.11: Google Rendering Context

```
1  class AmazonRenderingContext(
2    indentationLevel: Int = 0,
3    stringBuilder: StringBuilder = StringBuilder(),
4    termContext: TermContext = object : TermContext {},
5  ) : IndentedRenderingContext(indentationLevel, stringBuilder, termContext) {
6    companion object {
7      private val innerRenderingContext: MutableList<AmazonRenderingContext> =
       mutableListOf()
8      private val hostResolver: MutableMap<String, String> = mutableMapOf()
9    }
10   ...
11 }
```

Listing 3.12: Amazon Rendering Context

Due to the addition of being able to have different levels of rendering context, it is necessary to make it transparent to existing node renderers to minimize code change. To accomplish this the decorator pattern is used to wrap the *beginRender* and *endRender* methods from *IndentedNodeRenderer* and pass the correct rendering context to it, this can be seen in both Listing 3.13 and 3.14.

```
1  abstract class GoogleRenderer() : IndentedNodeRenderer() {
2    override fun beginRender(renderingContext: RenderingContext): String {
3      val context = (renderingContext as GoogleRenderingContext).
       getLastRenderingContext()
4      return super.beginRender(context)
5    }
6
7    override fun endRender(renderingContext: RenderingContext): String {
8      val context = (renderingContext as GoogleRenderingContext).
       getLastRenderingContext()
9      return super.endRender(context)
10   }
11 }
```

Listing 3.13: Google Renderer

```
1  abstract class AmazonRenderer() : IndentedNodeRenderer() {
2    override fun beginRender(renderingContext: RenderingContext): String {
3      val context = (renderingContext as AmazonRenderingContext).
       getLastRenderingContext()
4      return super.beginRender(context)
5    }
6
7    override fun endRender(renderingContext: RenderingContext): String {
8      val context = (renderingContext as AmazonRenderingContext).
       getLastRenderingContext()
9      return super.endRender(context)
10   }
11 }
```

Listing 3.14: Amazon Renderer

3.3.1 Google Renderers

As previous mentioned, GCP integration was straightforward since it supported all the new functionalities we wanted to add, so creating these new renderers was a matter of mapping to the correct GCP DSL for the structure.

3.3.1.1 Parallel Renderer

Since a Parallel can contain multiple branch context, one node renderer needs to be defined for each one. Starting with branch node renderer since it is straightforward as seen in the Listing 3.15, only need to define a name for the branch, use the `steps` keyword and incrementing the indentation so the child nodes can have the correct indentation.

```
1  class GoogleBranchRenderer(private val branchContext: BranchContext) :
       GoogleRenderer() {
2    override val element: Node = branchContext
3
4    override fun internalBeginRender(renderingContext: IndentedRenderingContext):
       String =
5      render(renderingContext) {
6        addLine("- ${branchContext.name}:")
7        incIndentationLevel()
8        add("steps:")
9        incIndentationLevel()
10     }
11   override fun internalEndRender(renderingContext: IndentedRenderingContext) =
12     render(renderingContext) {
13       decIndentationLevel()
14       decIndentationLevel()
15     }
16 }
```

Listing 3.15: Google Branch Node Renderer

As stated before to be able to support parallel, a stack of rendering context was needed to make the current implementation to work with these new sub workflows. As such the *internalBeginRender* in the parallel renderer in addition to using the GCP keywords `parallel` and `branches`, there is also the need to add a new rendering context to the stack as show in line 10 in the Listing 3.16. This new rendering context is then pop from the stack in the *internalEndRender*, line 27, at the end of the process of rendering this step. With this and the use of the *GoogleRenderer* previously mentioned it becomes transparent for child steps which rendering context is being used.

```
1  class GoogleParallelRenderer(private val parallelContext: ParallelContext) :
       IndentedNodeRenderer() {
2    override val element: Node = parallelContext
3
4    override fun internalBeginRender(renderingContext: IndentedRenderingContext):
       String {
5      val currentContext = (renderingContext as GoogleRenderingContext).
       getLastRenderingContext()
6      val innerContext = GoogleRenderingContext(
7        indentationLevel = currentContext.getIndentationLevel() + 1,
```

```
8        termContext = currentContext.termContext
9      )
10     currentContext.appendInnerRenderingContext(innerContext)
11     return render(currentContext) {
12       when (parallelContext) {
13         is ParallelIterationContext -> add("parallel:")
14         is ParallelBranchContext -> {
15           addLine("parallel:")
16           tab {
17             add("branches:")
18           }
19         }
20       }
21       incIndentationLevel()
22       innerContext.incIndentationLevel()
23     }
24   }
25   override fun internalEndRender(renderingContext: IndentedRenderingContext):
       String {
26     val googleRenderingContext = (renderingContext as GoogleRenderingContext)
27     val innerContext = googleRenderingContext.popLastRenderingContext()
28     val currentContext = googleRenderingContext.getLastRenderingContext()
29     val sharedVariables = currentContext.getVariables().map { it.variable.name }.
       toSet()
30       .intersect(innerContext.getVariables().map { it.variable.name }.toSet())
31     return render(currentContext) {
32       if (sharedVariables.isNotEmpty()) {
33         tab { add("shared: [${sharedVariables.joinToString(",")}]") }
34       }
35       decIndentationLevel()
36       innerContext.decIndentationLevel()
37     }
38   }
39 }
```

Listing 3.16: Google Parallel Node Renderer

3.3.1.2 Iteration Renderer

As seen in Listing 3.17 the implementation of *internalBeginRender* starts by adding the `for` keyword from GCP DSL, followed by `value` keyword to define that variable that will hold the current iteration element. After that, either `range` or `in` is used depending on if the iteration step is going over a range or a collection respectively. Ending on `steps` keyword to define the steps to be executed in each iteration before incrementing the indentation level. After all the child nodes are called, the *internalEndRender* removes the previous increment on the indentation level, so next steps at the same level can have the right indentation.

```
1  class GoogleIterationRenderer(private val iterationContext: IterationContext) :
       GoogleRenderer() {
2    override val element: Node = iterationContext
3
4    override fun internalBeginRender(renderingContext: IndentedRenderingContext):
       String =
5      render(renderingContext) {
```

```
6        addLine("for:")
7        tab {
8          addLine("value: ${iterationContext.value}")
9          when (iterationContext) {
10           is IterationRangeContext -> addLine("range: [${iterationContext.range.min
     }, ${iterationContext.range.max}]")
11           is IterationForEachContext -> addLine("in: \${${iterationContext.
     forEachVariable.name}}")
12         }
13         add("steps:")
14       }
15       incIndentationLevel()
16     }
17   override fun internalEndRender(renderingContext: IndentedRenderingContext) =
18     render(renderingContext) {
19       decIndentationLevel()
20     }
21 }
```

Listing 3.17: Google Iteration Node Renderer

3.3.2 Amazon Renderers

In the case of the Amazon some workarounds were necessary to be able to support some of the functionality, due to this limitation in support some parts were partially supported.

3.3.2.1 Parallel Renderer

Similar to the GCP node renderers for AWS, the same renderers need to be implemented. Due to being a bit more complex to build the AWS JSON, both the *AmazonBranchRenderer* and *AmazonParallelRenderer* ended up needing to also push and pop from the rendering context stack, in contrast to GCP which only needed to implement this logic in the *GoogleParallelRenderer*. Since the code for push and pop from the rendering context stack is the same it was omitted from both the Listing 3.18 and 3.19 only keeping the differences.

```
1  class AmazonBranchRenderer(private val branchContext: BranchContext) :
     IndentedNodeRenderer() {
2    override val element: Node = branchContext
3    override fun internalBeginRender(renderingContext: IndentedRenderingContext):
       String {
4      ...
5      return render(currentContext) {
6        addLine(AMAZON_OPEN_OBJECT)
7        incIndentationLevel()
8        innerContext.incIndentationLevel()
9        addLine("${AMAZON_START_AT}\"${branchContext.steps.first().name}\",")
10       add(AMAZON_STATES)
11     }
12   }
13   override fun internalEndRender(renderingContext: IndentedRenderingContext):
       String {
14     ...
15     return render(context) {
```

```
16          addLine(AMAZON_CLOSE_OBJECT)
17          declndentationLevel()
18          innerContext.declndentationLevel()
19          if (context.getNextStepNameAndAdvance() != null) {
20            add(AMAZON_CLOSE_OBJECT_WITH_COMMA)
21          } else {
22            add(AMAZON_CLOSE_OBJECT)
23          }
24        }
25      }
26    }
```

Listing 3.18: Amazon Branch Node Renderer

```
1   class AmazonParallelRenderer(private val parallelBranchContext:
        ParallelBranchContext) : IndentedNodeRenderer() {
2     override val element: Node = parallelBranchContext
3     override fun internalBeginRender(renderingContext: IndentedRenderingContext):
        String {
4       ...
5       return render(currentContext) {
6         addLine(AMAZON_PARALLEL_TYPE)
7         add(AMAZON_START_BRANCHES)
8       }
9     }
10    override fun internalEndRender(renderingContext: IndentedRenderingContext):
        String {
11      ...
12      return render(context) {
13        addLine(AMAZON_CLOSE_ARRAY_WITH_COMMA)
14        if (nextStepName == null) {
15          add(AMAZON_END)
16        } else {
17          add("$AMAZON_NEXT\"${nextStepName}\"")
18        }
19      }
20    }
21  }
```

Listing 3.19: Amazon Parallel Node Renderer

3.3.2.2 Iteration Renderer

The complexity appears once we implement loops, since `map` step from Amazon doesn't support sharing data between each iteration, we have to make a workaround. For that, whenever we are traversing the tree of nodes and find a specific iteration step we replace for a loop based on goto's as seen in Listing 3.20 where around the steps defined by the programmer we add four new steps. The first one to initialize a counter, the second step to increment the previous variable and also to be the jump point to continue the loop, following this step are the steps defined by the programmer, next is a condition step that validates either to continue the loop or to go to the final step which is an empty step.

```
1   private fun IterationRangeContext.childNodes(prefix: String): IterationRangeContext
        = this.copy(
2     steps = listOf(
```

```
    step {
      name("${prefix}InitializeCounter")
      description("Auto generated")
      context(
        assign {
          variables(variable("${value}.$") equalTo Value(range.min - 1))
        }
      )
    },
    step {
      name("${prefix}IncrementCounter")
      description("Auto generated")
      context(
        assign {
          variables(variable("${value}.$") equalTo value("States.MathAdd($.${value
    }, 1)"))
        }
      )
    }).map { it.build() }
    .plus(steps)
    .plus(
      listOf(
        step {
          name("${prefix}Loop?")
          description("Auto generated")
          context(
            switch {
              conditions(
                condition {
                  match(variable(value) lessThan Value(range.max))
                  jump("${prefix}IncrementCounter")
                },
              )
              default("${prefix}EndLoop")
            }
          )
        },
        step {
          name("${prefix}EndLoop")
          description("Auto generated")
          context(
            assign { }
          )
        }).map { it.build() })
)
```

Listing 3.20: Amazon Iteration Range Loop

After those iteration steps are rewritten in the traversing process, it follows the same process defined before for rendering steps. Due to this rewrite of the workflow it helps to keep this loop as a sub workflow as we end up using a parallel step with one branch from AWS as it can be seen in the line 7 in the Listing 3.21. The rest of the listing shows the mapping to the specific AWS DSL. Since we are using sub workflows here, we need to do the same push and pop from the rendering stack, although it's omitted in the listing.

```
class AmazonIterationRenderer(private val iterationContext: IterationContext) :
    IndentedNodeRenderer() {
  override val element: Node = iterationContext
```

```
3
4   override fun internalBeginRender(renderingContext: IndentedRenderingContext):
      String {
5     ...
6     return render(currentContext) {
7       addLine(AMAZON_PARALLEL_TYPE)
8       addLine(AMAZON_START_BRANCHES)
9       incIndentationLevel()
10      innerContext.incIndentationLevel()
11      addLine(AMAZON_OPEN_OBJECT)
12      incIndentationLevel()
13      innerContext.incIndentationLevel()
14      addLine("${AMAZON_START_AT}\"${innerContext.getNextStepNameAndAdvance()}\",")
15      add(AMAZON_STATES)
16    }
17  }
18  override fun internalEndRender(renderingContext: IndentedRenderingContext):
      String {
19    ...
20    return render(context) {
21      addLine(AMAZON_CLOSE_OBJECT)
22      decIndentationLevel()
23      innerContext.decIndentationLevel()
24      if (context.getNextStepNameAndAdvance() != null) {
25        addLine(AMAZON_CLOSE_OBJECT_WITH_COMMA)
26      } else {
27        addLine(AMAZON_CLOSE_OBJECT)
28      }
29      decIndentationLevel()
30      innerContext.decIndentationLevel()
31      addLine(AMAZON_CLOSE_ARRAY_WITH_COMMA)
32      if (nextStepName == null) {
33        add(AMAZON_END)
34      } else {
35        add("$AMAZON_NEXT\"${nextStepName}\"")
36      }
37    }
38  }
39 }
```

Listing 3.21: Amazon Iteration Node Renderer

3.3.3 Node Renderer Strategy

To be able to give information to the rendering process which node matches with what node rendered, a strategy pattern was used by the previous work. It defines an interface *NodeRendererStrategyFactory* that has two methods *getMatcher* which returns a Predicate to validate if a given node is of the correct type, and *getRenderer* that given a *Node* returns the correct *NodeRenderer* as seen in Listing 3.22. Using this strategy, adding support for the parallel, branch and iteration step types become straightforward, only needing to implement specializations of this *NodeRendererStrategyFactory*. One such example can be seen in Listing 3.23. Finally, this new strategy needs to be register together with the existing strategies.

```
interface NodeRendererStrategyFactory<R> {
  fun getMatcher(): Predicate<Node>

  fun getRenderer(node: Node): NodeRenderer<R>
}
```

Listing 3.22: Node Renderer Strategy

```
class GoogleBranchStrategyFactory : NodeRendererStrategyFactory<String> {
  override fun getMatcher(): Predicate<Node> =
    DefaultPredicate(BranchContext::class)

  override fun getRenderer(node: Node): NodeRenderer<String> =
    GoogleBranchRenderer(node as BranchContext)
}
```

Listing 3.23: Google Branch Strategy

4 Evaluation

This chapter presents a performance evaluation of the extended OmniFlow. It presents the choice of metrics used and what new benchmarks were created. The results of those benchmarks are then presented, and conclusions upon them are taken, to measure both the performance on newly added features and as existing features.

4.1 Overview

Since this library core functionality is translation between its agnostic DSL and a cloud provider specific one, the parts that are most important to benchmark are the rendering process and the deployment process. Since this work didn't involve changing the deployment process, we can discard its benchmarks and only focus on the rendering process. Since we are extending on previous work, it makes sense that we use the same metrics and methodology introduced. This way we can ensure consistency and comparability between both works to see if either the performance improved, stayed the same or degraded. In addition to this comparison between before and after, new benchmarks have been added to evaluate the new features. This approach makes it possible to quantify the impact of the new features, highlight improvements, and identify potential trade-offs introduced by the extended functionality.

As previous mentioned the core of this library is the translation, with that in mind previous work defined various different types of workflows that would show how the implementation behaves. Following a logarithmic scale, encompassing values such as 10, 100, 1.000, 10.000, and 100.000, it simulated how the system would perform as we increased the number of steps in a given workflow. Since the complexity of the workflow also affects the performance, it also defined a set of workflows that would validate different parts of the implementation. Taking this in mind, additional workflows were defined to also validate the newly implemented features for both iteration and parallel steps. Section 4.2 provides more details of each configuration.

With the new set of workflows defined, each one was executed 100 times to obtain an average execution time. This was evaluation was done in both the previous implementation and the new one to also compare and understand the impact that old workflows might have with the new implementation, being it negative or positive.

4.2 Rendering Process

This section goes into detail on what and how each workflow used in the benchmark is defined.

4.2.1 Independent

This represents the most simple workflow, which is composed of independent steps that don't have any dependencies between each other. This means that rendering each step doesn't need any data from the previous step to be rendered. More specifically, there is neither a conditional behavior nor a dependency in the result of the previous step.

4.2.2 Variables

Since variables are essential to storage and manipulate data in workflows, it was important to evaluate workflows with steps that would either initialize or use those variables. This benchmark makes some steps that initialize variables using the **assign** step, which later is used by others. This creates a dependency between the steps, increasing the complexity of the rendering process.

4.2.3 Binary Conditions

Conditions allow the control over program flow by making decisions based on expressions. Since this is a fundamental aspect when creating workflows, it is important to evaluate how the renderer behaves with them. For this benchmark, the focus is one of binary expressions, such as if-else expression, that alter the execution path of the workflow.

4.2.4 Multiple Decisions

Other than binary conditions, the workflow might have multiple possibles paths and not just two as example of a switch cases. To encapsulate those, the benchmark creates workflows that include switch steps, where certain operations can be skipped while executing other.

4.2.5 Iteration

One of the basic statements of programming is iterations, which allows the repetition of the execution of specific code. To be able to create complex workflows, iterations are a necessity, as such iterations need to be evaluated. For this benchmark, two different approaches were taken, the `forEach` iteration which applies the same operation to each element in a collection, and the **range** iteration, which repeats a step for a specified number of times. This evaluates the behavior of the render process with this new type of step.

4.2.6 Parallel

Parallel computing is a way that enables the execution of code to be done at the same time. Since this allows to create workflows more efficient due to enabling to processing tasks simultaneous, it is important to also evaluate this. For this benchmark, two variates were used to evaluate this new parallel step, one that creates a single branch with multiple steps and another with multiple branches with multiple steps.

4.2.7 Parallel with Iteration

Parallel iteration combines the concepts of repetition and concurrency, enabling multiple iterations to be executed at the same time. This approach allows workflows to become more efficient by applying iterations concurrently instead of sequentially. For this benchmark, the same two approaches use for iterations are use the `forEach` iteration and the **range** iteration. This evaluates how the render process manages workflows that combine both iteration and parallel execution.

4.3 Results

The evaluation was performed on a Lenovo laptop equipped with an AMD Ryzen 7 5825U processor and 40 GB of memory. To guarantee comparability with prior work, the same methodology and workload configurations were adopted. Two sets of benchmarks were analyzed: the original results from the baseline implementation and the new results obtained with the extended version of OmniFlow.

The benchmark uses the workflow structures identified before, as well as using a logarithmically ranged for the number of steps. This design not only ensured consistency with the previous evaluation but also allowed the impact of the new features, namely iteration and parallel steps, to be measured alongside existing constructs.

4.3.1 Independent

The results for workflows composed of independent steps show a substantial improvement in the rendering performance of the extended implementation. At 10 000 steps, AWS execution times improved from 152 ms to 33 ms, and GCP improved from 109 ms to 33 ms. At the largest scale of 100 000 steps, AWS improved from 2079 ms to 631 ms, and GCP from 1207 ms to 342 ms, Figure 4.1. These results demonstrate that variable handling in the new implementation introduces minimal overhead, while consistently outperforming the baseline across all tested scales.

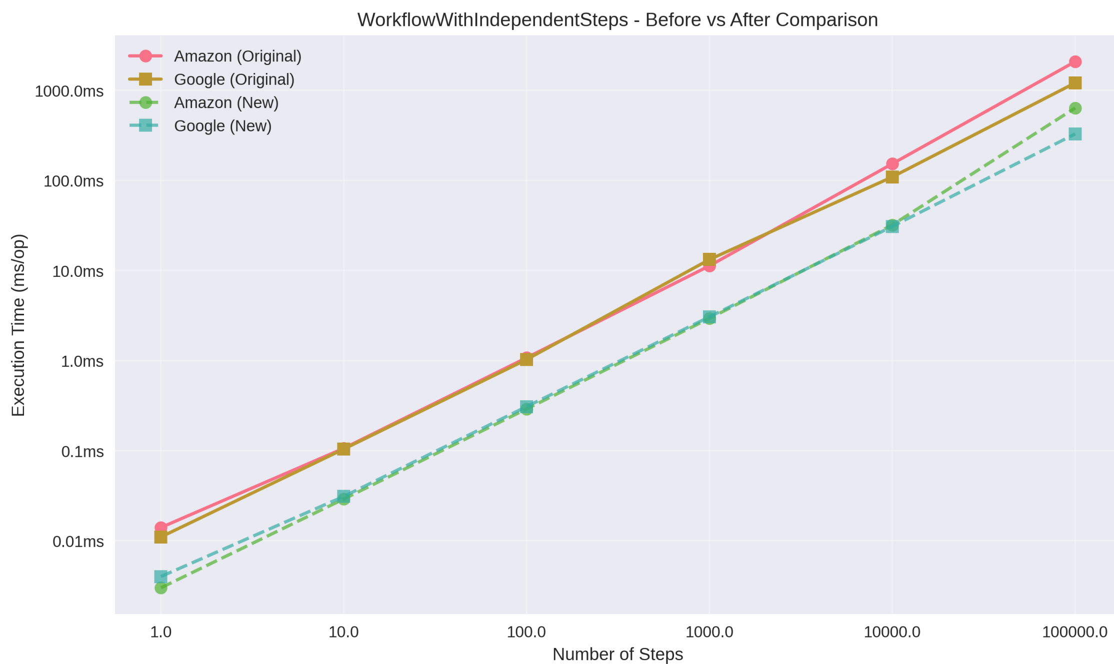


Figure 4.1: *Independent workflow benchmark original vs new implementation*

4.3.2 Variables

Workflows relying heavily on variable usage also benefited greatly from the reworked renderer. In the baseline, workflows of 10 000 steps required over 101 ms on AWS and 80 ms on GCP, while the new version reduced these to approximately 31 ms on AWS and 26 ms on GCP. At 100 000 steps, the extended renderer achieved around 607 ms for AWS and 279 ms for GCP, compared to over 1586 ms and 776 ms respectively in the original, Figure 4.2. This represents a reduction of nearly 61%, indicating that the optimizations in the rendering engine scale well as the workflow size increases.

4.3.3 Binary Conditions

For workflows including binary conditions, the extended implementation again demonstrated significant improvements. At 10 000 steps, AWS dropped from 94 ms in the original to 29 ms, and GCP from 92 ms to 25 ms. At 100 000 steps the difference is even

Figure 4.2: *Variables workflow benchmark original vs new implementation*

more pronounced, with AWS reduced from 1724 ms to 581 ms, and GCP from 1187 ms to 273 ms, Figure 4.3. This indicates that conditional branching is now handled much more efficiently, contributing to overall workflow scalability.

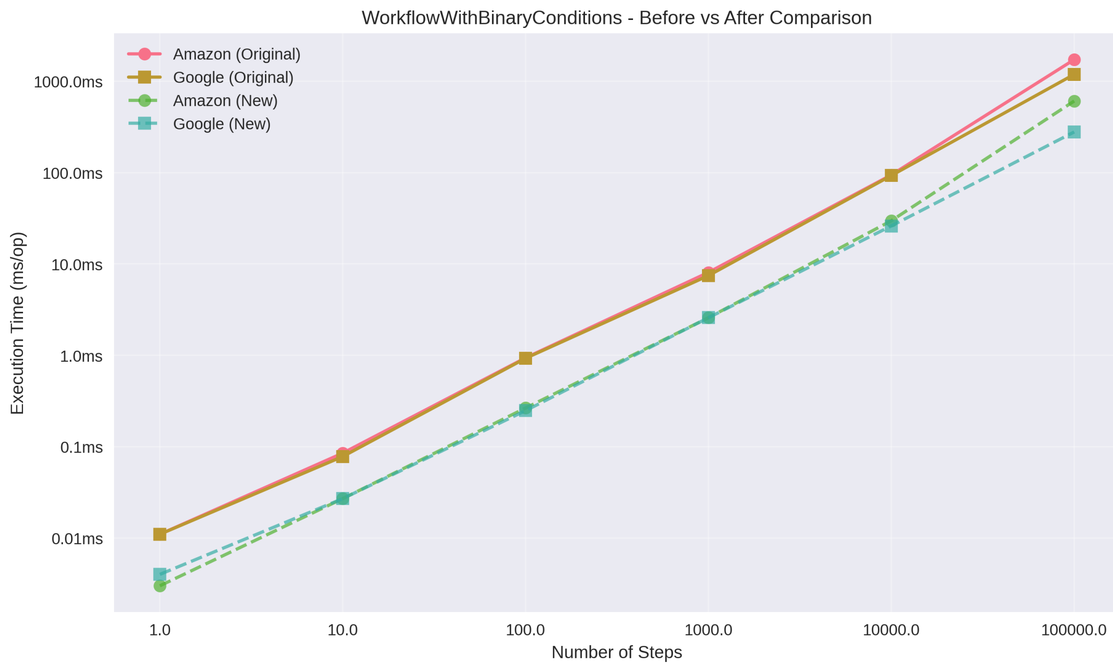


Figure 4.3: *Binary condition workflow benchmark original vs new implementation*

4.3.4 Multiple Decisions

Workflows with multiple decision points also showed a consistent performance gain. In the original implementation, a workflow with 10 000 steps required around 109 ms for AWS and 96 ms for GCP. The extended implementation reduced these values to 35 ms and 33 ms, respectively. At 100 000 in behaves the same, reducing from 1932 ms to 671 ms for AWS and 1310 ms to 353 ms for GCP, Figure 4.4. These results reinforce that the improvements to conditional rendering scale well as the complexity of decision-making logic increases.

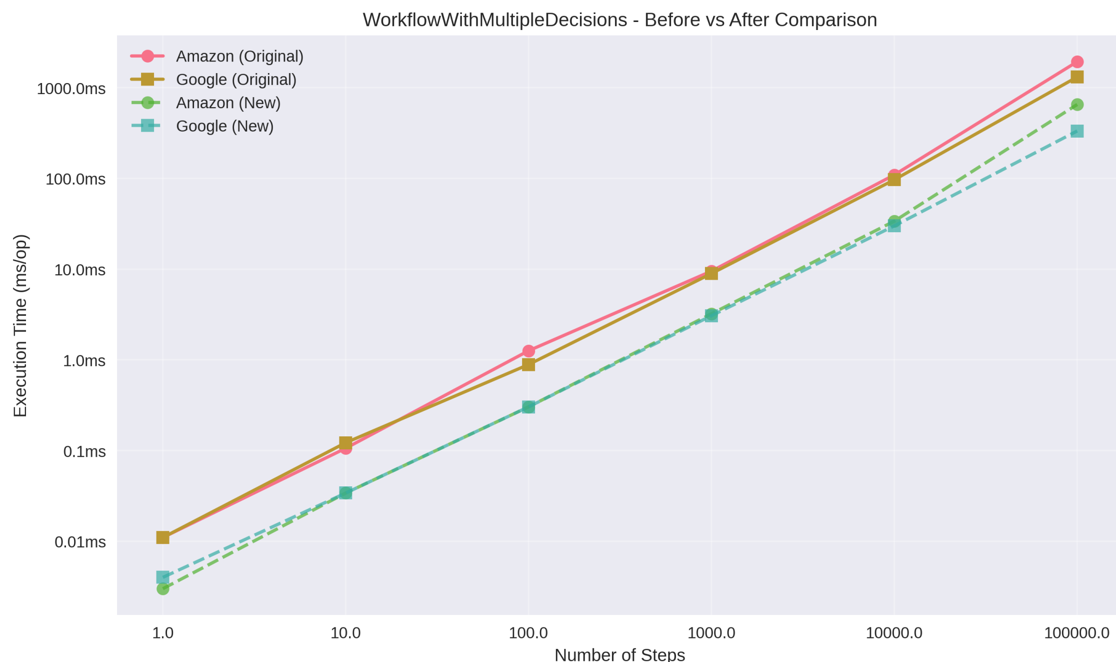


Figure 4.4: *Multiple decisions workflow benchmark original vs new implementation*

4.3.5 Iteration

The addition of iteration step was evaluated with both `forEach` and `range`. Both patterns scale linearly with the number of steps. For `forEach`, at 10 000 steps AWS workflows completed in 38 ms, while GCP required around 34 ms. At 100 000 steps, AWS required 685 ms and GCP around 383 ms, Figure 4.5. Similarly, `range` at 10 000 steps AWS workflows completed in 39 ms, while GCP required around 35 ms, while at 100 000 AWS required 702 ms and GCP around 388 ms, Figure 4.6. These values are only slightly higher than independent workflows, showing that iteration support introduces modest overhead while remaining efficient.

4.3.6 Parallel

The benchmarks for parallel workflows showed the expected additional cost of branch management, but performance remained efficient. A workflow with a single parallel branch

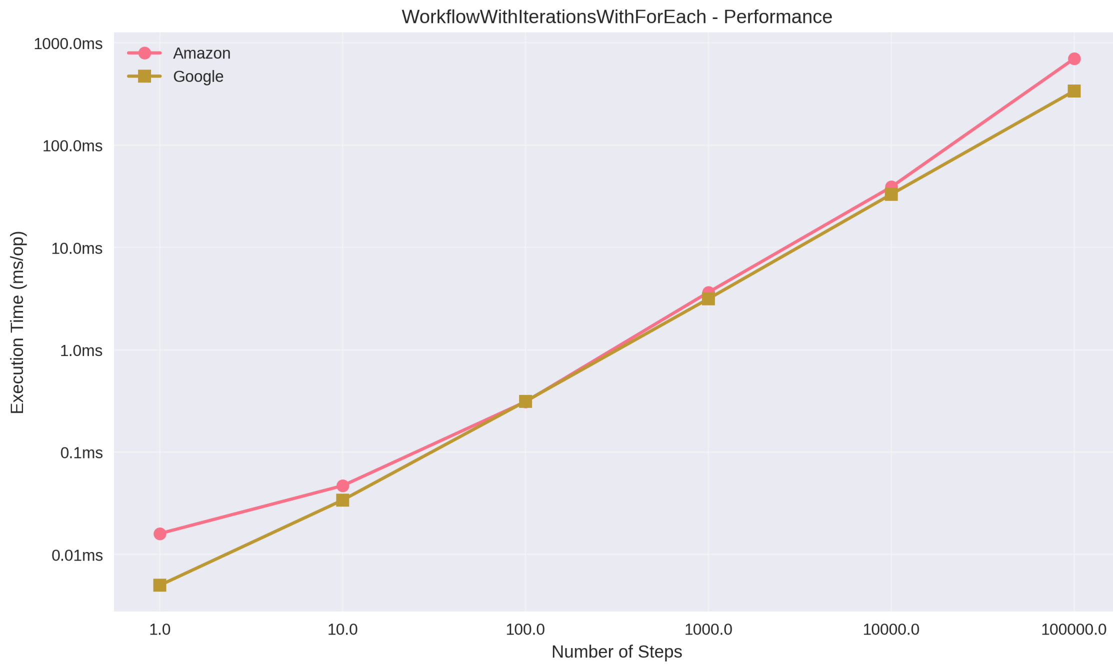


Figure 4.5: *Iteration forEach workflow variation benchmark*

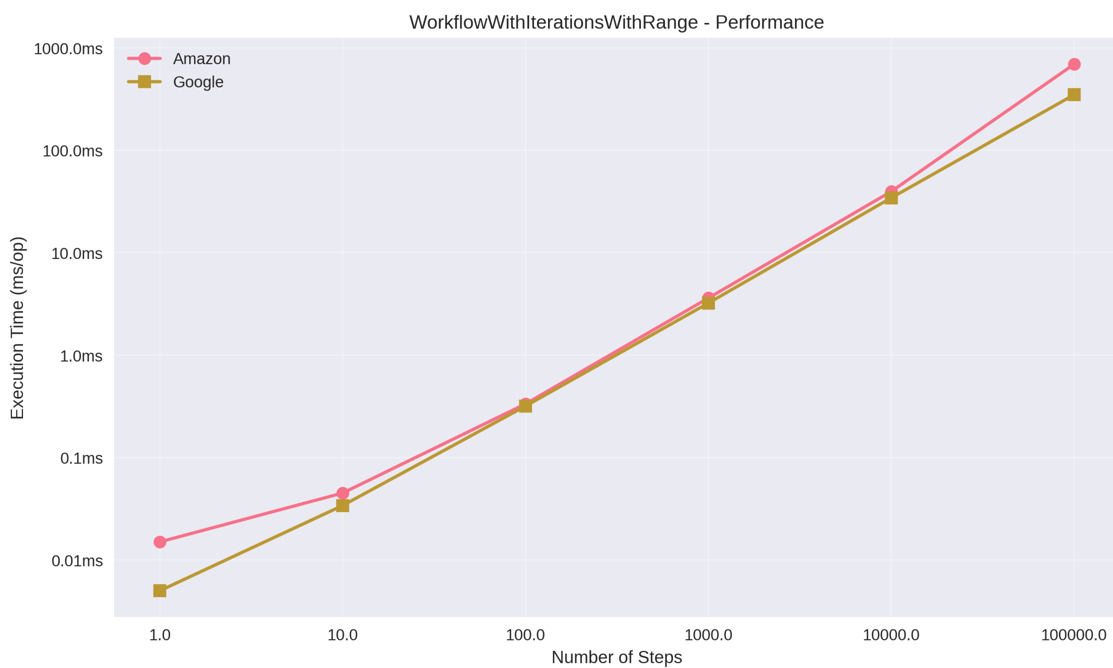


Figure 4.6: *Iteration range workflow variation benchmark*

executed in with 38 ms and 36 for 10 000 steps on both AWS and GCP respectively, while 100 000 steps required 742 ms on AWS and 385 ms on GCP, Figure 4.7. For multiple branches, AWS executed 10 000 steps in 49 ms and 100 000 steps in 493 ms, while GCP executed 10 000 steps in 46 ms and 100 000 in 493 ms, Figure 4.8. These results suggest that parallel branching is feasible at scale, with overheads proportional to branch complexity.

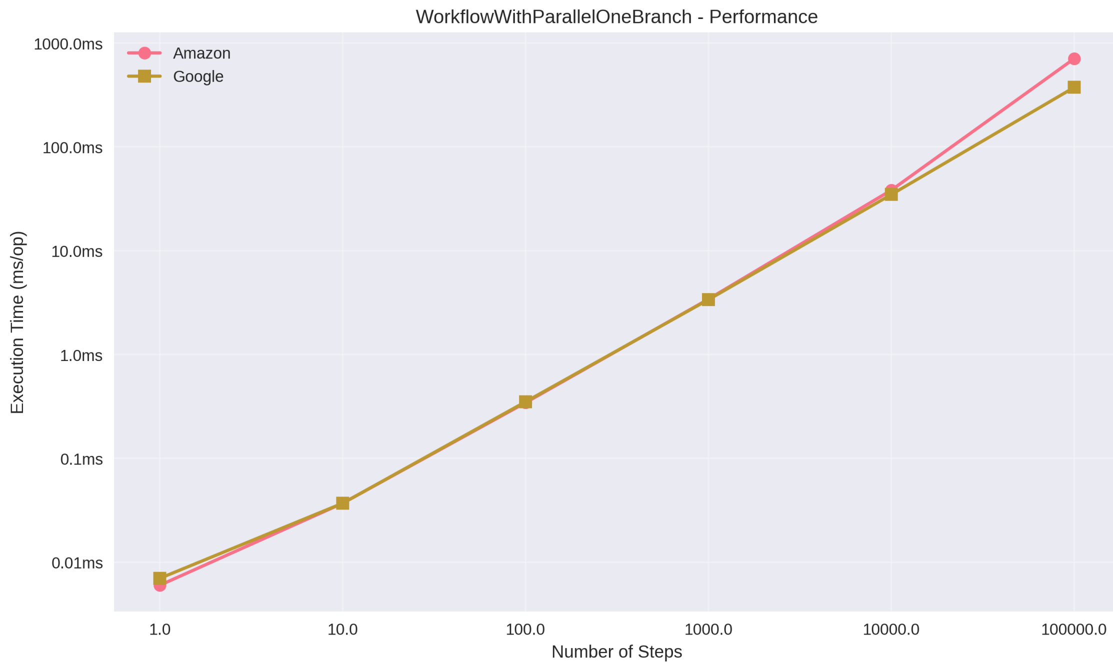


Figure 4.7: *Parallel with one branch workflow benchmark*

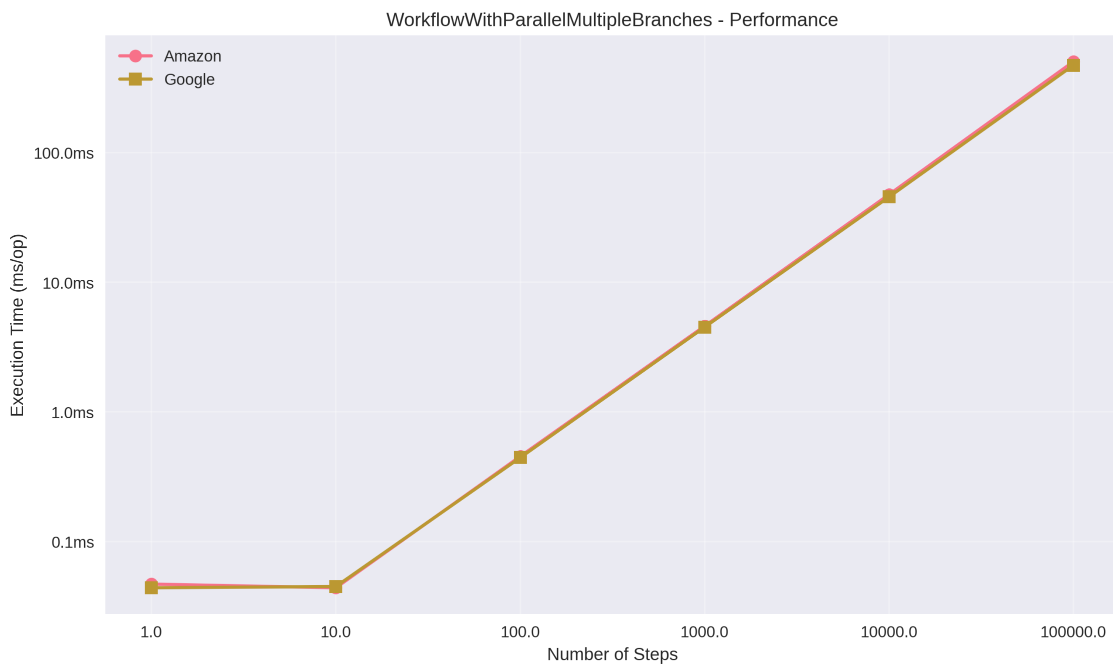


Figure 4.8: *Parallel with multiple branch's workflow benchmark*

4.3.7 Parallel with Iteration

The combination of parallelism with iteration presented the most demanding workloads. For `forEach` iterations executed in parallel, performance remained close to sequential iterations, with 100 000 steps requiring around 792 ms for AWS and 389 ms for GCP, as presented in Figure 4.9. However, range-based parallel iterations revealed limitations. At 100 000 steps, AWS required 7954 ms, several orders of magnitude higher than other workflows, while GCP stayed under 365 ms, Figure 4.10. This indicates that either the

combination of large numeric ranges with parallel execution on AWS introduces significant overhead, or an issue with the implementation due to the need of loop unrolling in AWS need for support.

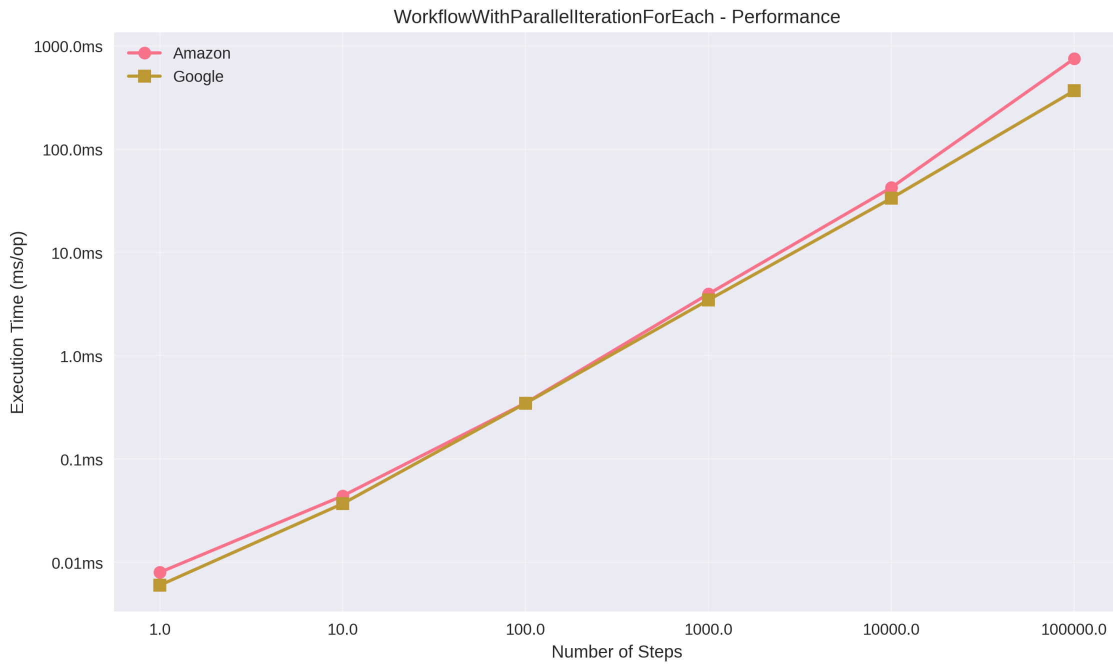


Figure 4.9: *Parallel iteration forEach workflow variation benchmark*

Figure 4.10: *Parallel iteration range workflow variation benchmark*

4.3.8 Improvements

Across all the benchmarks, the results confirm that there was an improvement with the new implementation when comparing with the original. The performance could be attributed to the update of the underlying Java platform and core libraries from version 11 to version 17, which has improvements in the garbage collector. But since the garbage collector was executed before each benchmark as to minimize the influence of the metrics, it doesn't seem to be the major contributor. On the other hand, the refactor in rendering process which minimized reliance on the Jackson library for internal data handling shows the bigger differences when profiling the benchmark. By changing to native Kotlin data structures, the overhead associated with JSON serialization and deserialization was significantly reduced. The use of these data structures help to reduce the frequency of computationally expensive string splitting operations.

The introduction of iteration and parallel constructs demonstrated acceptable overheads, validating their practicality for real-world workloads. These optimizations ensure that the extended OmniFlow is not only more feature-rich, but also more scalable and efficient.

4.4 Case Study Sentiment Classification Workflow

In addition to the benchmarks presented in this chapter, a more practical workflow was developed to demonstrate how the extended DSL can be applied to a real-world scenario. This example provides a qualitative evaluation of the DSL usability, which complements the quantitive benchmark results. This case study focus on an AI based sentiment classification workflow, which makes use of the iteration and parallel step types introduced in this work to process multiple text inputs concurrently.

This workflow combines the use of parallel, iteration, and conditional branching to illustrate how the extended DSL can model a sentiment analysis system. Figure 4.11 shows an overview of the workflow using OmniFlow DSL.

The workflow begins by defining a list of texts to classify and an empty feedback map which hold the results for each text, shown in Listing 4.1. These variables are used in the subsequent steps of the workflow.

```
1 step {
2   name("AssignInputs")
3   context(
4     assign {
5       variables(
6         variable("inputTexts") equalTo value(listOf("Awesome!", "This is bad.", "
      Neutral Response."))
7       )
8       variables(variable("feedback") equalTo value(mapOf<String, Value<*>>()))
9 }))
```

Listing 4.1: Initialization of input variables

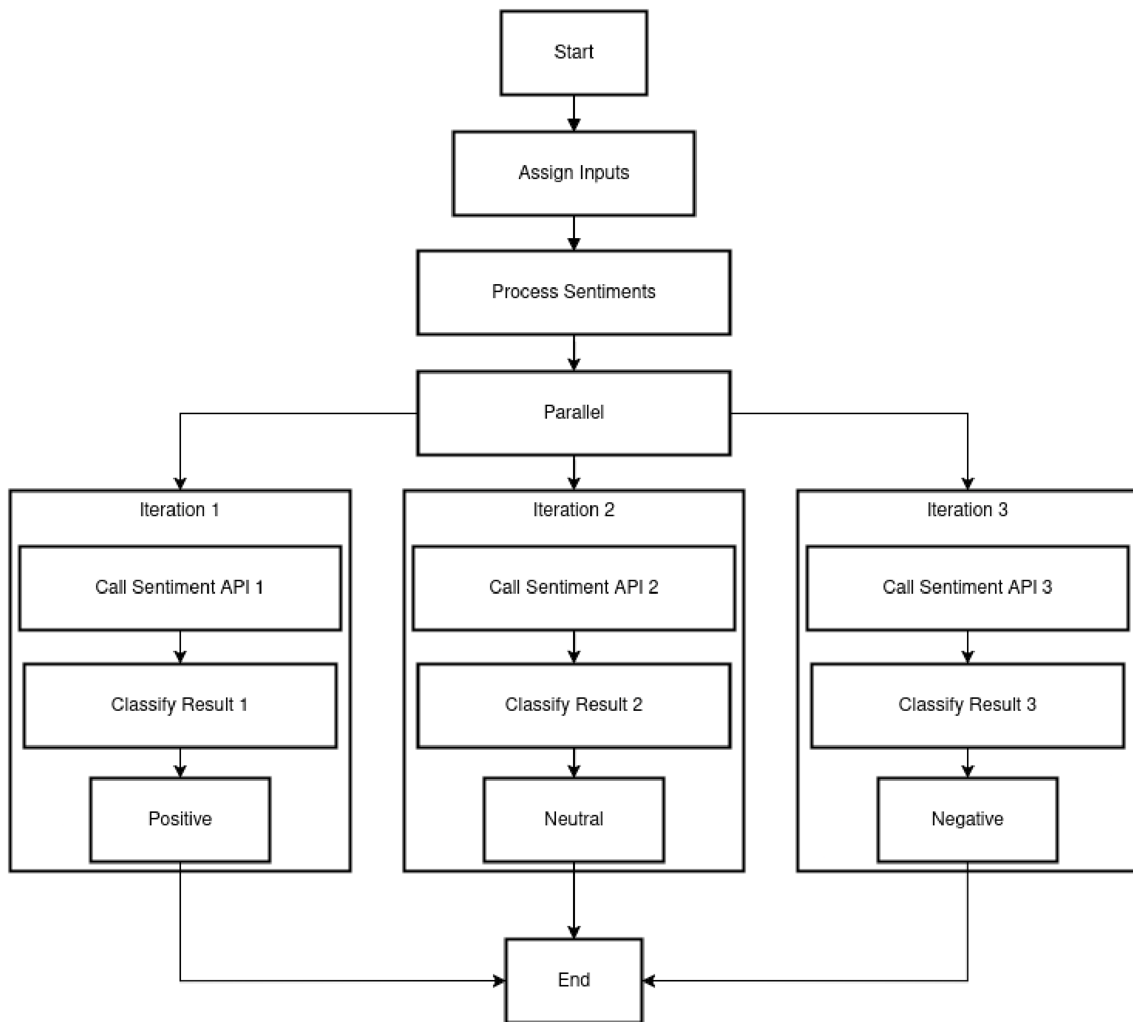


Figure 4.11: *Sentiment Classification Process represented with OmniFlow DSL*

Following the initial setup, the workflow uses a parallel iteration construct to iterate over the list of texts, invoking a sentiment classification API for each item in parallel, as shown in 4.2.

```
1  step {
2    name("ProcessSentiments")
3    context(
4      parallel {
5        iteration {
6          key("textItem")
7          forEach(variable("inputTexts"))
8          steps(
9            step {
10             name("CallSentimentAPI")
11             context(
12               call {
13                 method(HttpMethod.POST)
14                 host("https://example.com")
15                 path("/analyze")
16                 header(...)
17                 body(mapOf("text" to variable("textItem")))
18                 result("sentiment")
19                 resultType(ResultType.BODY)
20         })},
```

Listing 4.2: Parallel iteration and API call for sentiment classification

Finally, the workflow maps the sentiment result return from the API and updates the feedback variable according to the predicted label, shown in the Listing 4.3.

```
1          step {
2            name("ClassifyResult")
3            context(
4              switch {
5                conditions(
6                  condition {
7                    match(variable("sentiment").withKey("sentiment_score") equalTo
     value(1))
8                    jump("PositiveFeedback")
9                  },
10                 condition {
11                   match(variable("sentiment").withKey("sentiment_score") equalTo
     value(-1))
12                   jump("NegativeFeedback")
13               })
14               default("NeutralFeedback")
15         })},
16         step {
17           name("PositiveFeedback")
18           context(
19             assign {
20               variables(variable("feedback").withKey("textItem") equalTo value("
     Positive"))
21           })
22           next("Continue")
23         },
24         step {
25           name("NegativeFeedback")
26           context(
27             assign {
28               variables(variable("feedback").withKey("textItem") equalTo value("
     Negative"))
29           })
30           next("Continue")
31         },
32         step {
33           name("NeutralFeedback")
34           context(
35             assign {
36               variables(variable("feedback").withKey("textItem") equalTo value("
     Neutral"))
37           })
38           next("Continue")},
```

Listing 4.3: Handling of sentiment classification results

A complete workflow listing is provided in the Appendix F. This workflow demonstrates how we can use the extended DSL compose more advance workflow. Each text is processed independently in parallel, and the `parallel` and `iteration` constructs enable concurrent execution without relying on provider specific syntax.

5 Conclusion

This work extended the OmniFlow framework by incorporating advanced composition features essential for modern serverless applications. The work directly addresses the problem of vendor lock-in, where workflows become tied to the proprietary solutions of a single cloud provider. The goal was to enhance OmniFlow's Domain-Specific Language (DSL) to give developers a more capable and portable method for orchestrating complex serverless functions.

To add more advanced capabilities, the OmniFlow DSL was first expanded with iteration and parallel execution support. New keywords were introduced, and the rendering component was updated to generate the correct definitions for Amazon Web Services (AWS) and Google Cloud Platform (GCP). As a result, a distinction between the suppliers was seen. A custom workaround was needed for AWS because its Map iterator lacks shared-state support, a problem not found with GCP's more direct implementation.

Performance benchmarks confirmed the success of these enhancements and also yielded an unexpected benefit. The refactored renderer made the new implementation significantly faster than the original across all workflow types, a result of platform updates and a shift to native data structures over the Jackson library for internal operations. While the new iteration and parallel features scaled efficiently with acceptable overhead, the evaluation uncovered a critical limitation: range-based parallel iterations on AWS performed orders of magnitude slower than other configurations. This bottleneck could be a direct consequence of the loop unrolling workaround needed to support this feature on AWS, or a problem with the implementation.

5.1 Future work

Based on the findings of this work, some paths for future work can be taken to improve the capabilities of OmniFlow. The primary one that comes from this work is addressing the significant performance bottleneck observed with range based parallel iteration on

AWS. Future research should investigate alternative implementation that can mitigate the increase in overhead when increasing the size of the workflow. Another direction is extending the supported cloud providers by adding as an example Microsoft Azure. Since this work is a continuation of OmniFlow which is a continuation of QuickFaaS would be interesting integrating both, to have a unified tooling set to be able to define not just workflows but also the code that gets executed in custom serveless functions. Last but not least, improving the usability of OmniFlow through additional tooling such as a validation layer to check for errors in the creation procesynciousss, or having a graphical user interface to help generate workflows while minimizing the amount of code needed.

In closing, this project transforms OmniFlow into a more complete and practical tool for serverless orchestration. It provides developers the means to build sophisticated, dynamic workflows that remain portable across different cloud environments, directly contributing to a more flexible and interoperable serverless ecosystem.

Bibliography

[1] Amazon. *Amazon Web Services Step Functions*. https://aws.amazon.com/pt/step-functions/. Accessed: 2024-12-04 (cit. on p. 10).

[2] Bernardo José Mateus Costa, Filipe Bastos de Freitas, José Manuel de Campos Lages Garcia Simão. "Function composition in Function-as-a-Service Platforms". MA thesis. Instituto Superior de Engenharia de Lisboa, 2023 (cit. on p. 1).

[3] C. Commons. *Serverless Workflow*. https://serverlessworkflow.io/. Accessed: 2024-11-27 (cit. on p. 14).

[4] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, David A. Patterson. "Cloud programming simplified: A Berkeley view on serverless computing". In: *arXiv preprint arXiv:1902.03383* (2019) (cit. on p. 1).

[5] Google. *Google Cloud Platform Workflows*. https://cloud.google.com/workflows?hl=en. Accessed: 2024-12-04 (cit. on p. 7).

[6] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. "Serverless Computing: A Survey of Opportunities, Challenges, and Applications". In: *ACM Computing Surveys, vol. 54, no. 11s* (2022), pp. 1–32 (cit. on p. 1).

[7] Iberian Conference on Information Systems and Technologies. *CISTI 2025*. https://www.cisti.eu/2025/index.php/en/w. Accessed: 2025-09-30 (cit. on p. 2).

[8] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, Olivier Tardieu. "The serverless trilemma: Function composition for serverless computing". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. ACM. 2017, pp. 89–103 (cit. on p. 1).

[9] Kamil Kojs. "A Survey of Serverless Machine Learning Model Inference". In: *arXiv preprint arXiv:2311.13587v1* (2023) (cit. on p. 1).

[10] Pedro García-López, Aitor Arjona, Josep Sampe, Aleksander Slominski and Lionel Villard. "Triggerflow: Trigger-based orchestration of serverless workflows". In: *Proceedings of the 14th ACM international conference on distributed and event-based systems* (2022), p. 1 (cit. on p. 13).

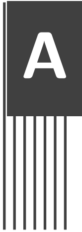
[11] Sashko Ristov, Simon Brandacher, Mika Hautz, Michael Felderer, Ruth Breu. "CODE: Code once, deploy everywhere serverless functions in federated FaaS". In: *Future Generation Computer Systems* 160 (2024), pp. 442–456 (cit. on p. 14).

[12] Sungjae Park, R. Quinn Thomas, Cayelan C. Carey, Austin D. Delany, Yun-Jung Ku, Mary E. Lofton, Renato J. Figueiredo. "FaaSr: Cross-Platform Function-as-a-Service Serverless Scientific Workflows in R". In: *IEEE 20th International Conference on e-Science (e-Science)* (2024), p. 1 (cit. on p. 14).

[13] T. Technologies. *Temporal Platform*. https://temporal.io/. Accessed: 2024-11-27 (cit. on p. 14).

[14] Tiago Silva. *Omniflow*. https://github.com/ISEL-DEETC/omni-flow. Accessed: 2025-09-30 (cit. on p. 2).

[15] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. "Faasflow: Enable efficient workflow execution for function-as-a- service". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* 1.1 (2022), 782–796 (cit. on p. 13).



A Example of a calculation workflow using OmniFlow DSL

```
1  {
2    name("calculatorWorkflow")
3    description("Calculator example")
4    steps(
5      step {
6        name("InitVariables")
7        description("Initialize variables")
8        context(
9          assign {
10           variable("a" equal Random().nextInt())
11           variable("b" equal Random().nextInt())
12           variable("c" equal Random().nextInt())
13         }
14       )
15     },
16     step {
17       name("Sum")
18       description("Sum 2 random numbers")
19       context(
20         call {
21           method(GET)
22           host("r1ro8xa7y8.execute-api.us-east-1.amazonaws.com")
23           path("/default/calculator")
24           query(
25             "number1" to variable("a"),
26             "number2" to variable("b"),
27             "op" to value("add")
28           )
29           result("sumResult")
30         }
31       )
32     },
33     step {
34       name("Condition")
35       description("condition")
36       context(
37         switch {
38           conditions(
39             condition {
40               match(variable("c") equalTo value(0))
41               jump("Assign1ToC")
```

```
42              },
43              condition {
44                match(variable("c") greaterThan value(0))
45                jump("DivWithC")
46              }
47            )
48            default("Assign1ToC")
49          }
50        )
51      },
52      step {
53        name("Assign1ToC")
54        description("If c equal to 0 affect C with 1")
55        context(
56          assign {
57            variable("c" equal 1)
58          }
59        )
60      },
61      step {
62        name("DivWithC")
63        description("Divide the previous result by a random value")
64        context(
65          call {
66            method(GET)
67            host("r1ro8xa7y8.execute-api.us-east-1.amazonaws.com")
68            path("/default/calculator")
69            query(
70              "number1" to variable("sumResult"),
71              "number2" to variable("c"),
72              "op" to value("div")
73            )
74            result("divResult")
75          }
76        )
77      }
78    )
79    result("divResult")
80  }
```

Listing A.1: OmniFlow simple calculation workflow

B Amazon Web Services map workflow

```
{
  "StartAt": "DefineInput",
  "States": {
    "DefineInput": {
      "Type": "Pass",
      "Parameters": {
        "Posts": [1, 2, 3, 4],
        "Results": []
      },
      "Next": "MapPosts"
    },
    "MapPosts": {
      "Type": "Map",
      "ItemsPath": "$.Posts",
      "ItemSelector": {
        "PostId.$": "$$.Map.Item.Value"
      },
      "ItemProcessor": {
        "StartAt": "GetPostCommentCount",
        "States": {
          "GetPostCommentCount": {
            "Type": "Task",
            "Resource": "arn:aws:states:::http:invoke",
            "Parameters": {
              "ApiEndpoint": "https://example.com/users/$.args.userId",
              "Authentication": {
                "ConnectionArn": "arn:aws:events:eu-north-1:123456789012:connection/Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
              },
              "Method": "GET"
            },
            "ResultPath": "$.NumComments",
            "End": true
          }
        }
      },
      "ResultPath": "$.Results",
      "Next": "End"
    },
    "End": {
      "Type": "Pass",
      "End": true
```

```
42         }
43     }
44 }
```

Listing B.1: Amazon Web Services map workflow

C Amazon Web Services iteration workflow

```
{
  "StartAt": "InitializeIndex",
  "States": {
    "InitializeIndex": {
      "Type": "Pass",
      "Parameters": {
        "Index": −1,
        "Sum": 0,
        "List": [10, 11, 12, 13, 14]
      },
      "Next": "IncrementIndex"
    },
    "IncrementIndex": {
      "Type": "Pass",
      "Parameters": {
        "Index.$": "States.MathAdd($.Index, 1)",
        "List.$": "$.List",
        "Sum.$": "$.Sum"
      },
      "ResultPath": "$",
      "Next": "MockWork"
    },
    "MockWork": {
      "Type": "Pass",
      "Parameters": {
        "Index.$": "$.Index",
        "List.$": "$.List",
        "Sum.$": "States.MathAdd($.Sum, States.ArrayGetItem($.List, $.Index))"
      },
      "ResultPath": "$",
      "Next": "Loop?"
    },
    "Loop?": {
      "Type": "Choice",
      "Choices": [
        {
          "Variable": "$.Index",
          "NumericGreaterThanEquals": 4,
          "Next": "Log"
        }
      ],
      "Default": "IncrementIndex"
```

```
43        },
44        "Log": {
45          "Type": "Pass",
46          "Parameters": 0,
47          "Next": "Success"
48        },
49        "Success": {
50          "Type": "Succeed"
51        }
52      },
53      "TimeoutSeconds": 3
54    }
```

Listing C.1: Amazon Web Services iteration workflow

D Amazon Web Services parallel workflow

```
{
  "StartAt": "Parallel Branches",
  "States": {
    "Parallel Branches": {
      "Type": "Parallel",
      "Next": "Log",
      "Branches": [
        {
          "StartAt": "GetUserCall",
          "States": {
            "GetUserCall": {
              "Type": "Task",
              "Resource": "arn:aws:states:::http:invoke",
              "Parameters": {
                "ApiEndpoint": "https://example.com/users/$.args.userId",
                "Authentication": {
                  "ConnectionArn": "arn:aws:events:eu-north-1:123456789012:connection/Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
                },
                "Method": "GET"
              },
              "End": true
            }
          }
        },
        {
          "StartAt": "GetNotificationCall",
          "States": {
            "GetNotificationCall": {
              "Type": "Task",
              "Resource": "arn:aws:states:::http:invoke",
              "Parameters": {
                "ApiEndpoint": "https://example.com/notification/$.args.notificationId",
                "Authentication": {
                  "ConnectionArn": "arn:aws:events:eu-north-1:123456789012:connection/Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
                },
                "Method": "GET"
              },
              "End": true
            }
```

```
40             }
41           }
42         ]
43       },
44       "Log": {
45         "Type": "Pass",
46         "End": true
47       }
48   }
49 }
```

Listing D.1: Amazon Web Services parallel workflow

E Amazon Web Services parallel iteration workflow

```
{
  "StartAt": "DefineInput",
  "States": {
    "DefineInput": {
      "Type": "Pass",
      "Parameters": {
        "Posts": [ 1, 2, 3, 4 ],
        "Results": []
      },
      "Next": "MapPosts"
    },
    "MapPosts": {
      "Type": "Map",
      "ItemsPath": "$.Posts",
      "ItemSelector": {
        "PostId.$": "$$.Map.Item.Value"
      },
      "ItemProcessor": {
        "StartAt": "GetPostCommentCount",
        "States": {
          "GetPostCommentCount": {
              "Type": "Task",
              "Resource": "arn:aws:states:::http:invoke",
              "Parameters": {
                "ApiEndpoint": "https://example.com/users/$.args.userId",
                "Authentication": {
                  "ConnectionArn": "arn:aws:events:eu-north-1:123456789012:connection/Stripe/81210c42-8af1-456b-9c4a-6ff02fc664ac"
                },
                "Method": "GET"
              },
              "ResultPath": "$.NumComments",
              "End": true
          }
        }
      },
      "ResultPath": "$.Results",
      "Next": "ReduceComments"
    },
    "ReduceComments": {
      "Type": "Pass",
      "Parameters": {
```

```
42             "RunningTotal": 0,
43             "Index": 0,
44             "ArrayLength.$": "States.ArrayLength($.Results)",
45             "Results.$": "$.Results"
46           },
47           "ResultPath": "$",
48           "Next": "AccumulateComments"
49         },
50         "AccumulateComments": {
51           "Type": "Choice",
52           "Choices": [
53             {
54               "Variable": "$.Index",
55               "NumericLessThanPath": "$.ArrayLength",
56               "Next": "UpdateReduceLoop"
57             }
58           ],
59           "Default": "LogTotalComments"
60         },
61         "UpdateReduceLoop": {
62           "Type": "Pass",
63           "Parameters": {
64             "RunningTotal.$": "$.RunningTotal",
65             "Index.$": "$.Index",
66             "ArrayLength.$": "$.ArrayLength",
67             "Results.$": "$.Results",
68             "Item.$": "States.ArrayGetItem($.Results, $.Index)"
69           },
70           "ResultPath": "$",
71           "Next": "AddToTotal"
72         },
73         "AddToTotal": {
74           "Type": "Pass",
75           "Parameters": {
76             "RunningTotal.$": "States.MathAdd($.RunningTotal, $.Item.NumComments)",
77             "Index.$": "States.MathAdd($.Index, 1)",
78             "ArrayLength.$": "$.ArrayLength",
79             "Results.$": "$.Results"
80           },
81           "ResultPath": "$",
82           "Next": "AccumulateComments"
83         },
84         "LogTotalComments": {
85           "Type": "Pass",
86           "Parameters": {
87             "FinalTotalComments.$": "$.RunningTotal"
88           },
89           "End": true
90         }
91       }
92     }
```

Listing E.1: Amazon Web Services parallel iteration workflow

F Sentiment classification workflow

```
1  workflow {
2    name("ParallelSentimentClassifier")
3    description("Classify sentiment for multiple texts in parallel")
4    steps(
5      step {
6        name("AssignInputs")
7        description("Set the list of texts")
8        context(
9          assign {
10           variables(
11             variable("inputTexts") equalTo value(
12               listOf(
13                 "Awesome!",
14                 "This is bad.",
15                 "Neutral Response."
16               )
17             )
18           )
19           variables(variable("feedback") equalTo value(mapOf<String, Value<*>>()))
20         }
21       )
22     },
23     step {
24       name("ProcessSentiments")
25       description("Run sentiment classification in parallel for each input")
26       context(
27         parallel {
28           iteration {
29             key("textItem")
30             forEach(variable("inputTexts"))
31             steps(
32               step {
33                 name("CallSentimentAPI")
34                 description("Calls a sentiment classification API")
35                 context(
36                   call {
37                     method(HttpMethod.POST)
38                     host("https://sentiment.soik.eu")
39                     path("/analyze")
40                     header(
41                       HEADER_CONTENT_TYPE to value(CONTENT_TYPE_APPLICATION_JSON),
42                       "X-API-Key" to value("your-secret-api-key-here"),
```

```
43                       )
44                       body(mapOf("text" to variable("textItem")))
45                       result("sentiment")
46                       resultType(ResultType.BODY)
47                     }
48                   )
49                 },
50                 step {
51                   name("ClassifyResult")
52                   description("Handles result")
53                   context(
54                     switch {
55                       conditions(
56                         condition {
57                           match(
58                             variable("sentiment").withKey("sentiment_score")
59                                 equalTo
60                                 value(1)
61                           )
62                           jump("PositiveFeedback")
63                         },
64                         condition {
65                           match(
66                             variable("sentiment").withKey("sentiment_score")
67                                 equalTo
68                                 value(-1)
69                           )
70                           jump("NegativeFeedback")
71                         }
72                       )
73                       default("NeutralFeedback")
74                     }
75                   )
76                 },
77                 step {
78                   name("PositiveFeedback")
79                   description("Sets feedback to positive")
80                   context(
81                     assign {
82                       variables(
83                         variable("feedback").withKey("textItem") equalTo value("
      Positive")
84                       )
85                     }
86                   )
87                   next("Continue")
88                 },
89                 step {
90                   name("NegativeFeedback")
91                   description("Sets feedback to negative")
92                   context(
93                     assign {
94                       variables(
95                         variable("feedback").withKey("textItem") equalTo value("
      Negative")
96         workflow {
97   name("ParallelSentimentClassifier")
98   description("Classify sentiment for multiple texts in parallel")
99   steps(
100    step {
```

```
101         name("AssignInputs")
102         description("Set the list of texts")
103         context(
104           assign {
105             variables(
106               variable("inputTexts") equalTo value(
107                 listOf(
108                   "Awesome!",
109                   "This is bad.",
110                   "Neutral Response."
111                 )
112               )
113             )
114             variables(variable("feedback") equalTo value(mapOf<String, Value<*>>()))
115           }
116         )
117       },
118       step {
119         name("ProcessSentiments")
120         description("Run sentiment classification in parallel for each input")
121         context(
122           parallel {
123             iteration {
124               key("textItem")
125               forEach(variable("inputTexts"))
126               steps(
127                 step {
128                   name("CallSentimentAPI")
129                   description("Calls a sentiment classification API")
130                   context(
131                     call {
132                       method(HttpMethod.POST)
133                       host("https://sentiment.soik.eu")
134                       path("/analyze")
135                       header(
136                         HEADER_CONTENT_TYPE to value(CONTENT_TYPE_APPLICATION_JSON),
137                         "X-API-Key" to value("your-secret-api-key-here"),
138                       )
139                       body(mapOf("text" to variable("textItem")))
140                       result("sentiment")
141                       resultType(ResultType.BODY)
142                     }
143                   )
144                 },
145                 step {
146                   name("ClassifyResult")
147                   description("Handles result")
148                   context(
149                     switch {
150                       conditions(
151                         condition {
152                           match(
153                             variable("sentiment").withKey("sentiment_score")
154                                 equalTo
155                                 value(1)
156                           )
157                           jump("PositiveFeedback")
158                         },
159                         condition {
160                           match(
```

```
161                                 variable("sentiment").withKey("sentiment_score")
162                                     equalTo
163                                     value(-1)
164                               )
165                               jump("NegativeFeedback")
166                             }
167                           )
168                           default("NeutralFeedback")
169                         }
170                       )
171                     },
172                     step {
173                       name("PositiveFeedback")
174                       description("Sets feedback to positive")
175                       context(
176                         assign {
177                           variables(
178                             variable("feedback").withKey("textItem") equalTo value("
    Positive")
179                           )
180                         }
181                       )
182                       next("Continue")
183                     },
184                     step {
185                       name("NegativeFeedback")
186                       description("Sets feedback to negative")
187                       context(
188                         assign {
189                           variables(
190                             variable("feedback").withKey("textItem") equalTo value("
    Negative")
191                           )
192                         }
193                       )
194                       next("Continue")
195                     },
196                     step {
197                       name("NeutralFeedback")
198                       description("Sets feedback to neutral")
199                       context(
200                         assign {
201                           variables(
202                             variable("feedback").withKey("textItem") equalTo value("
    Neutral")
203                           )
204                         }
205                       )
206                       next("Continue")
207                     },
208                     step {
209                       name("Continue")
210                       description("Continue to next step")
211                       context(
212                         assign {
213                           variables(
214                             variable("_") equalTo value("")
215                           )
216                         }
217                       )
```

```
218                     }
219                   )
220                 }
221               }
222             )
223           }
224         )
225         result("feedback")
226 }                     )
227                         }
228                       )
229                       next("Continue")
230                     },
231                     step {
232                       name("NeutralFeedback")
233                       description("Sets feedback to neutral")
234                       context(
235                         assign {
236                           variables(
237                             variable("feedback").withKey("textItem") equalTo value("
      Neutral")
238                           )
239                         }
240                       )
241                       next("Continue")
242                     },
243                     step {
244                       name("Continue")
245                       description("Continue to next step")
246                       context(
247                         assign {
248                           variables(
249                             variable("_") equalTo value("")
250                           )
251                         }
252                       )
253                     }
254                   )
255                 }
256               }
257             )
258           }
259         )
260         result("feedback")
261 }
```

Listing F.1: Sentiment classification workflow