The Design, Productization, and Evaluation of a Serverless Workflow-Management System

Erwin van Eyk

Technische Universiteit Delft

TU Delft

The Design, Productization, and Evaluation of a Serverless Workflow-Management System

by

Erwin van Eyk

to obtain the degree of Master of Science in Computer Science
at the Delft University of Technology,
to be defended publicly on Friday June 21, 2019 at 9:30 AM.

Student number:	4176774	
Thesis committee:	Prof. dr. ir. Alexandru Iosup,	TU Delft & VU Amsterdam, supervisor
	Prof. dr. ir. Jan S. Rellermeyer	TU Delft
	Prof. dr. ir. Arie van Deursen	TU Delft
	Prof. dr. ir. Alessandro Bozzon	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

The need for accessible and cost-effective IT resources has led to the near-universal adoption of cloud computing. Within cloud computing, *serverless computing* has emerged as a model that further abstracts away operational complexity of heterogeneous cloud resources. Central to this form of computing is *Function-as-a-Service (FaaS)*; a cloud model that enables users to express applications as functions, further decoupling the application logic from the hardware and other operational concerns. Although FaaS has seen rapid adoption for simple use cases, there are several issues that impede its use for more complex use cases. A key issue is the lack of systems that facilitate the reuse of existing functions to create more complex, composed functions. Current approaches for serverless function composition are either proprietary, resource inefficient, unreliable, or do not scale. To address this issue, we propose an approach to orchestrate composed functions using reliably and efficiently with *workflows*. As a prototype, we design and implement Fission Workflows: an open-source serverless workflow system which leverages the characteristics of serverless functions to improve the (re)usability, performance, and reliability of function compositions. We evaluate our prototype using both synthetic and real-world experiments, which show that the system is comparable with or better than state-of-the-art workflow systems, while costing significantly less. Based on the experimental evaluation and the industry interest in the Fission Workflows product, we believe that serverless workflow orchestration will enable the use of serverless applications for more complex use cases.

Preface

I am both proud and relieved having finally completed this work. This thesis has truly become a defining point in my life. Although it started off with a broken collar bone and a tedious recovery, it has since then taken me across the globe, from an internship in California, to attending and speaking at conferences in the USA, Europe, and China. It has offered me the chance to collaborate with an international research group (SPEC RG), and it gave me the opportunity to meet all kinds of interesting people.

It also made me realize how much of this work was only possible with the support of my friends, colleagues, and family.

First, I want to thank my supervisor prof. Alexandru Iosup, for relentlessly pushing me to take risks. For giving me more than enough opportunities to participate in research projects, conferences, and other reasons to postpone that thesis thing for just another month. And, for being nothing but supportive along that long way.

Besides Alexandru, I am grateful for the fun times with the rest of the AtLarge team. Thank you, Sacheendra, Vincent, Laurens, Lucian, Ahmed, Alexandru Uta, and all others of the AtLarge team for the collaborations, numerous dinners, and coffee breaks.

For me personally, the internship at Platform9 Systems remains the highlight of this thesis. I enjoyed every moment of those three months in the bay area, which was for the most part due to all the great people at Platform9. In specific, I want to thank Soam Vasani, for championing the internship; for supporting me and my work without question; and, for remaining critical when needed.

I want to thank my family and friends. Martijn and Jaap for all the coffee. Luc and Kasper for occasionally cleaning their dishes. My parents, grandparents, Daniëlle, Ruben, Stefan, Mette-Marit, and Frank and Lianne for their support and, above all, their patience.

Finally, I want to thank my girlfriend, Tracy. You have been there, supporting me through the good times, and the bad times. I love you.

Erwin van Eyk
Delft, April 2019

Contents

List of Figures

List of Tables

1 Introduction

Cloud computing has been one of the great equalizers of computer technology. Whereas a decade ago only the largest enterprises could afford the enormous costs of globally distributed, highly available, and scalable systems, today anyone can deploy such a system in the cloud at a fraction of the costs [29]. Not only does cloud computing reduce the Capital Expenses (CapEx) of acquiring servers and other infrastructure, it also reduces the Operating Expenses (OpEx). *Cloud users*[1] defer the complexity of operating the infrastructure to dedicated *cloud providers*; multiplexing physical infrastructure and expensive (human) infrastructure expertise, saving costs through the economics of scale.

Yet, cloud computing is still resource inefficient and operationally complex [29]. Even key aspects that gave rise to cloud computing itself—reducing cost and complexity—are far from solved. Although cloud computing currently saves companies costs through the multiplexing of resources, the utilization of these multiplexed cloud resources is still low (35%, as observed in 2018 by Rightscale [112]). Moreover, the computing primitives that cloud providers offer today—such as Virtual Machines (VMs), containers, and network and storage options—remain complex. This requires companies to manage the distributed-system complexity themselves, with reports estimating that companies are losing $250 million annually due to a lack of expertise.

From this community-wide effort to reduce the complexity of deploying and managing global-scale, highly available cloud applications, and to improve the capabilities of the cloud providers to utilize their resources more efficiently, numerous systems, approaches, and companies have emerged. Examples of this evolution in computation include: containers (e.g., Docker and LXC), container and resource orchestrators (e.g., Kubernetes and Mesos), and application platforms (e.g., Cloud Foundry and Heroku). In this effort a new cloud model has emerged taking the cost and complexity concerns to a new extreme: *serverless computing*.

Within this emerging serverless computing model we focuses on the problem of systems-level support. We aim to address the lack of systems that orchestrate these serverless applications. In this thesis, we propose a novel approach to the composition of serverless applications based on workflows.

1.1. A Serverless Introduction

Although, the most common level of abstraction that cloud providers offer is the VM—an abstraction of a (part of a) physical machine, invented in the 1960s and refined in the 1970s [107]—cloud providers are continuously attempting to reduce complexity by offering resources at higher levels of abstraction. A consequence of this has been the recent emergence of serverless computing.

What is serverless computing? As proposed in our work with the SPEC WG CLOUD [132], we define serverless computing as *a form of cloud computing which allows users to run (1) event-driven and (2) granularly billed applications (3) without having to deal with operational logic*. A form of serverless computing is the FaaS model. The FaaS provider manages the lifecycle, and event-driven execution of

[1] For consistency, the terms related to cloud computing follow the definitions as defined by the NIST [38]

user-provided cloud functions. In this model, not a VM, but a cloud function (also referred to in this work as a *serverless function*, or a *FaaS function*) is the core abstraction. This abstraction is conceptually not too different from functions in functional programming or even from lambdas in lambda calculus [94].

The functions in the FaaS model are managed by a FaaS platform. The FaaS platform is responsible for the deployment and correct operation of the user-provided functions. Among its operational responsibilities are instrumenting and monitoring the performance of the function, along with automatically scaling—autoscaling—the resources assigned to the function based on the incoming workload. In Section 2.1.4 we describe the anatomy of a FaaS platform.

What makes serverless computing promising? The motivation behind the emergence of serverless computing is similar to that for cloud computing [132]: it further reduces complexity and cost of distributed applications. The function abstraction is high-level, which allows the cloud consumer to focus on the business logic—logic that is directly related to their use case—while delegating to the operator all or most of operational logic—which includes among others resource management, auto-scaling, and networking. This allows users to minimize their code base; leading to reduced time-to-market, and increased agility, all while the function is being deployed according to the cloud provider's expertise with fault-tolerance, high availability, and high scalability, etc.

Why did serverless emerge now? In earlier work [134], we found that the emergence of serverless computing is not mere coincidence; it is the result of vital developments in multiple fields of computer science. These developments indicate serverless could not have emerged even a decade ago—simply, the technology concepts and processes needed for the emergence of serverless computing did not exist. Looking only at the technology developments in virtualization in the past 20 years, the decreasing overhead of VMs and the emergence of alternative computation encapsulation models, such as containers, have enabled deploying functions on-demand easy and economical. Similarly, the maturing of comprehensive resource orchestrators (such as Kubernetes), provided stable middleware to build serverless platforms on top of. Finally, the programming models of event-driven architectures and the loose approaches based on ́functionsẃere needed for the clearly defined model of serverless computing.

Is the serverless market a hype? Currently, the market uptake of serverless computing and FaaS is already large. Serverless is also the fastest growing cloud computing field [112]. The serverless market currently worth around $5 billion and is predicted to be worth nearly $15 billion by 2023[2]. Correspondingly, many major public cloud providers already offer a FaaS solution. Their technological platforms for serverless are maturing: these cloud providers further offer other serverless products, such as serverless-friendly databases (e.g., AWS Aurora). In the open-source community, numerous FaaS platforms and serverless projects are actively being developed. Finally, the potential of serverless computing has recently triggered the interest of academia [63, 79, 83].

1.2. Problem Statement

However successful and fast-growing serverless computing is, various challenges exist in this relatively immature domain and are potentially holding back further adoption, as expanded upon extensively in our work with the SPEC RG CLOUD [132, 133]. We differentiate among these challenges: software engineering, operational, and performance-related challenges. Software engineering challenges include how to migrate legacy systems, and how to educate the next generation of software engineers in this paradigm. From an operational or system perspective, challenges include how to approach existing and new security concerns, how the life-cycle of these cloud functions is best managed, and how to satisfy the diverse Non-Functional Requirements (NFRs) that appear when a single (cloud) system must satisfy many different clients. Finally, from the domain of performance engineering, challenges arise such as how to compare the different, often closed-source, FaaS platforms, how to isolate the performance between concurrent function invocations, and how to find the right balance between performance and resource costs for different workloads.

[2] https://www.marketsandmarkets.com/Market-Reports/serverless-architecture-market-64917099.html

The goal of this thesis is to explore a key unsolved challenge: *How to support serverless computing based on functional composition through a distributed systems approach?* Although there are many examples of how FaaS functions can solve simple use cases, such as re-sizing a user-provided image [90] or sending a notification upon an event [93], it becomes surprisingly complicated when the use case is no longer trivial. Assuming that a user is able to invoke other functions from within a function, the user suddenly has to manage communication complexity. For instance, the user has to handle cases of (transient) errors, time outs, long-tail latencies, etc. This goes against the characteristic of serverless computing that the user should not be concerned with operational logic. Besides the added operational complexity, there are also concerns when calling cloud functions from other cloud functions that can not be handled correctly within the current model of short-lived, stateless cloud functions. We can not handle long-running orchestrations of functions, or ensure the completion of composed functions in case of temporary network or system unavailability.

1.3. Main Research Questions

We decompose this problem of finding an effective approach to function composition in serverless computing into four research questions:

RQ1 What are the requirements to support cloud function compositions?

Both the technical and the business aspects of the FaaS model lead to a new type of workload. To understand how this influences function composition, we need to analyze the specific characteristics, constraints, workloads, that appear in practical use cases.

RQ2 How do approaches for the composition of cloud functions compare?

Using the requirements found for RQ1, we can survey approaches to composition to determine which options are relevant to serverless functions. By comparing both existing and emerging approaches for orchestrating FaaS functions, we can determine which approaches are most promising, and where our research should be focused on.

RQ3 How to design a system for composing cloud functions?

Currently, there is a lack of research on the design of orchestration systems which focus specifically on the FaaS function model. By exploring the design space of such systems, we aim to understand the design implications and opportunities that this model brings to such systems.

RQ4 How to evaluate systems for function composition experimentally?

To understand the operation of the proposed system, we need to quantify it experimentally. Although analytical models could be useful, the novelty of the serverless technology makes it unlikely that the models could be validated and calibrated. More experimental results seem to be needed, before an analytical approach, or a simulation-based alternative, could produce provably meaningful results. Yet, no standardized tools or methods currently exist for experimentally evaluating function composition systems.

1.4. A Distributed Systems Approach

In this work, we approach the problem statement and the subsequent research questions with a distributed systems approach—a combination of conceptual, technical, and experimental work.

To address *RQ1* and *RQ2*, we investigate the current state-of-the-art. We perform an extensive requirements analysis for function composition in FaaS, in which we focus on the current and prospective use cases. Starting from these requirements, constraints, and use cases of function composition in serverless computing, we survey the different approaches and prior work in composition. Our analysis of the results reveals that a workflow-based approach fits the requirements and constraints of serverless function composition.

Based on the requirements analysis and survey, we address *RQ3*, through the design of a Workflow Management System (WMS) that facilitates effective and efficient cloud function composition using workflows. The design relies on state-of-the-art concepts in (distributed) systems and on concepts taken from control theory. As a part of this design, we propose a scheduler model for FaaS.

We implement the design not merely as a minimal prototype, but as a proof-of-concept meant for production use. In fact, the resulting software artifact is now the core of a professional serverless product, Fission Workflows. Implementing a new concept for this level of technology readiness [62] raises non-trivial implementation challenges, and leads to insights gained and lessons learned.

Finally, to answer *RQ4*, we propose an experimental approach for systematic performance evaluation, and evaluate the prototype with diverse experiments targeting different aspects. We evaluate the fault tolerance of the prototype by emulating scenarios, such as a crash of the workflow engine, and the temporary loss of connectivity between the prototype and the underlying FaaS. We further introduce experiments that evaluate the scalability under increasing workloads through stress testing—in which we vary specific parameters, such as the workflow structure, persistence mechanisms, and scheduling policies. To put the experiment results into the wider context, we evaluate performance and cost under realistic conditions of the prototype, comparing it to existing, state-of-the-art workflow management systems (Apache Airflow, AWS Step Functions, and Azure Logic Apps).

1.5. Main Contributions

The work leading to this thesis includes contributions of various kinds: *Conceptual* contributions proposing a new design, policy, etc.; *Experimental* contributions proposing ne w approaches, and techniques to evaluate serverless computing; *Software Artifact* contributions, such as a new software system developed with modern software engineering methods, and published on GitHub. Considering all these types of contributions, this thesis contains the following contributions:

1. (Conceptual) A requirements analysis for function composition (Chapter 3).
2. (Conceptual) A qualitative survey and analysis of the current state-of-the-art of FaaS platforms and approaches to serverless function composition (Chapter 4).
3. (Conceptual) The design of Fission Workflows: a serverless workflow management system to compose cloud function with a Serverless Workflow Language (Chapter 5).
4. (Software Artifact) The Fission Workflows prototype and product (Chapters 5, 6, and 7).
5. (Conceptual) The design of a scheduling architecture for scheduling serverless workflows, and of mechanisms and policies associated with this (Chapter 6).
6. (Experimental) The design of an experimental setup for evaluating and comparing serverless composition approaches (Chapter 8).
7. (Software Artifact) The SimFaaS platform for emulating the performance, resource usage, and cost characteristics of FaaS platforms (Chapter 8).
8. (Experimental) Quantitative results and an analysis of the proposed system, implemented as a near-production-ready prototype (Chapter 8).
9. (Conceptual) Publications analyzing the field, challenges and directions of serverless computing research:

   (i) **The SPEC RG CLOUD group's research vision on FaaS and serverless architectures** [132] (WoSC, 2017, lead author).

   As one of the first vision papers in the serverless computing domain, the SPEC RG CLOUD's initial vision paper on serverless computing aimed to address community issues; it proposed structured definitions for key concepts and placed these within the wider context of cloud computing. We further posed high-level perspectives and over 20 challenges in a number of domains (software engineering, performance engineering, and systems engineering) as a call to action to the academic community to contribute to the emerging serverless computing effort.

   The article presented the motivation, perspectives, and the open challenge of serverless function composition, which gave rise to this thesis. Furthermore, the definitions and descriptions related to serverless computing used in this thesis stem from this work (see Section 1.1, and Section 2.1).

(ii) **A SPEC RG CLOUD Group's Vision on the Performance Challenges of FaaS Cloud Architectures** [133] (ACM ICPE, 2018, lead author).

In our second vision paper, we (the SPEC RG CLOUD) focused further on the challenges in our domain of expertise, that of performance engineering and evaluation. For 6 performance challenges, we presented the current obstacles, existing work, and potential solutions. Additionally, we outlined the roadmap of the SPEC RG CLOUD with regards to a subset of these challenges: a reference architecture for FaaS platforms, and developing a comprehensive FaaS benchmark.
The outlined scheduling challenges specifically gave rise to an initial investigation of the challenges and opportunities in scheduling serverless workflows in Chapter 6.

(iii) **Serverless is More: From PaaS to Present Cloud Computing** [134] (IEEE Internet Computing, 2018, lead author).

This article aimed to put the emerging of serverless computing into the wider context of (cloud) computing. Presenting a chronological overview of the technologies and processes fundamental to serverless computing, we argued that serverless computing is an inevitable next step and could not have emerged a decade ago. The article further provides an overview of the key benefits of serverless computing, and presents the key obstacles remaining towards further adoption.
In relation to this thesis, the article identified the domain of WMSs, which motivated the focus of this work on workflows, influencing Chapters 4, and 5. A shortened version of this chronological overview of the emerging of serverless-related concepts and systems, has been integrated in the introduction to serverless computing (Section 1.1).

(iv) **The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms** [135] (under submission at IEEE Internet Computing, 2019, lead author).

For this article we surveyed near 50 FaaS platforms and other systems related to operating FaaS architectures. Based on this survey we developed a comprehensive reference architecture for a FaaS platform along with common operational patterns.
Part of this work has been reproduced in the background on FaaS platforms (Section 2.1.4). The survey on FaaS platforms (Section 4.1) influenced and was influenced by the SPEC work. Finally, the FaaS emulator (Section 8.1.1.4) used in the experimentation is based on the reference architecture.

(v) **Quantifying Cloud Performance and Dependability: Taxonomy, Metric Design, and Emerging Challenges** [66] (ACM TOMPECS, 2018, co-author).

This work, in collaboration with the SPEC RG CLOUD group, re-examined traditional system properties and considered new system properties, towards a re-design of classic benchmark metrics, such as expressing performance as throughput and latency.
As a co-author, aside from participating in general discussions and smaller editing work, I was responsible for the last part of the title; addressing the emerging challenges in metric design and performance evaluation—where these challenges in part are caused by the increasing presence of serverless services.
Findings from this article were incorporated in the experimental evaluation (Chapter 8) in the thesis, to select appropriate metrics to evaluate the prototype with.

(vi) **An Analysis of Workflow Formalisms for Workflows with Complex Non-Functional Requirements** [136] (HotCloudPerf, 2018, co-author).

In collaboration with Laurens Versluis, we investigated formalisms for expressing workflows with dynamic properties: Directed Acyclic Graphs (DAGs), Petri Nets, and Business Process Model and Notation (BPMN). After a qualitative evaluation, we found that DAGs seem most favourable of the three to represent workflows with dynamic non-functional requirements.
The findings of the article are reiterated in the background on workflows in Section 2.2.1. Furthermore, they also motivated the decision to survey solely DAG-based workflow languages (Section 4.4), and WMSs (Section 4.3). The findings further influenced the design of the Serverless Workflow Language—it is based on the DAG formalism—in Section 5.2.

(vii) **Massivizing Computer Systems: a Vision to Understand, Design, and Engineer Computer Ecosystems through and beyond Modern Distributed Systems** [74] (ICDCS, 2018, co-author).

The article introduces Massivizing Computer Systems (MCS), a domain of computer science focusing on understanding, controlling, and evolving successfully ecosystems of computer systems. This domain. We describe principles, challenges, and use cases—which includes serverless systems.

As a co-author, I was responsible for positioning serverless use cases within the MCS domain. Similarly, within the article I framed earlier and ongoing work on serverless research within the larger perspective of ecosystems.

(viii) **The OpenDC Vision: Towards Collaborative Datacenter Simulation and Exploration for Everybody** [73] (ISPDC, 2017, co-author).

Here, we (the AtLarge Team) posed our vision of OpenDC, a comprehensive, open-source, platform for simulating datacenters, to explore of various datacenter concepts and technologies, using existing and new scientific methods, enabling new education practices and topics, and leading to the creation of new software and data artifacts.

As a co-author, my main contribution was framing the OpenDC vision for the exploration of serverless computing research.

(ix) **Package-Aware Scheduling of FaaS Functions** [20] (HotCloudPerf, 2018, co-author).

This article aimed to improve the execution of FaaS functions by caching specific (parts of the) sources of functions in or near the resources where the functions could be deployed. The proposed package-aware scheduling algorithm increases the hit rate of the package cache and, as a result, reduces the overall latency of executing the cloud functions.

As a co-author, I was responsible for posing the technical work into the larger serverless perspective. This included introducing relevant serverless concepts, and ensuring that the work matched the constraints of the serverless model.

(x) **The AtLarge Vision on the Design of Distributed Systems and Ecosystems** [75] (ICDCS, 2019, co-author).

In this article we—the AtLarge group—describe the AtLarge design framework in an effort to promote the notion of design research within the domain of distributed systems.

Within this effort, my contribution consisted of describing our research experience and insights with regards to designing serverless systems. It contains descriptions of our on-going collaborations with SPEC RG CLOUD, and the work related to this thesis.

1.6. Reading Guide

This chapter has introduced the field of serverless computing, the problem of cloud function composition, and the approach taken towards solving this problem. The next chapter (Chapter 2) provides more background information on the topics underlying this thesis: control theory, workflow orchestration, and a more in-depth overview of serverless computing. We analyze the requirements, constraints, and use cases of composition in serverless computing in Chapter 3. Using the results of this analysis, we survey the state-of-the-art systems, languages, and approaches for function composition in Chapter 4. Based on the survey, in Chapter 5, we present the design of a workflow-based function composition system prototype. As a part of this prototype, we propose a hierarchical scheduling model for scheduling in FaaS and propose three scheduling policies for the prototype's scheduler, in Chapter 6. Chapter 7 details the implementation of the prototype—Fission Workflows—and documents the transition from prototype to product. In Chapter 8 we experimentally evaluate the prototype. Finally, in Chapter 9 we conclude and present directions for future work.

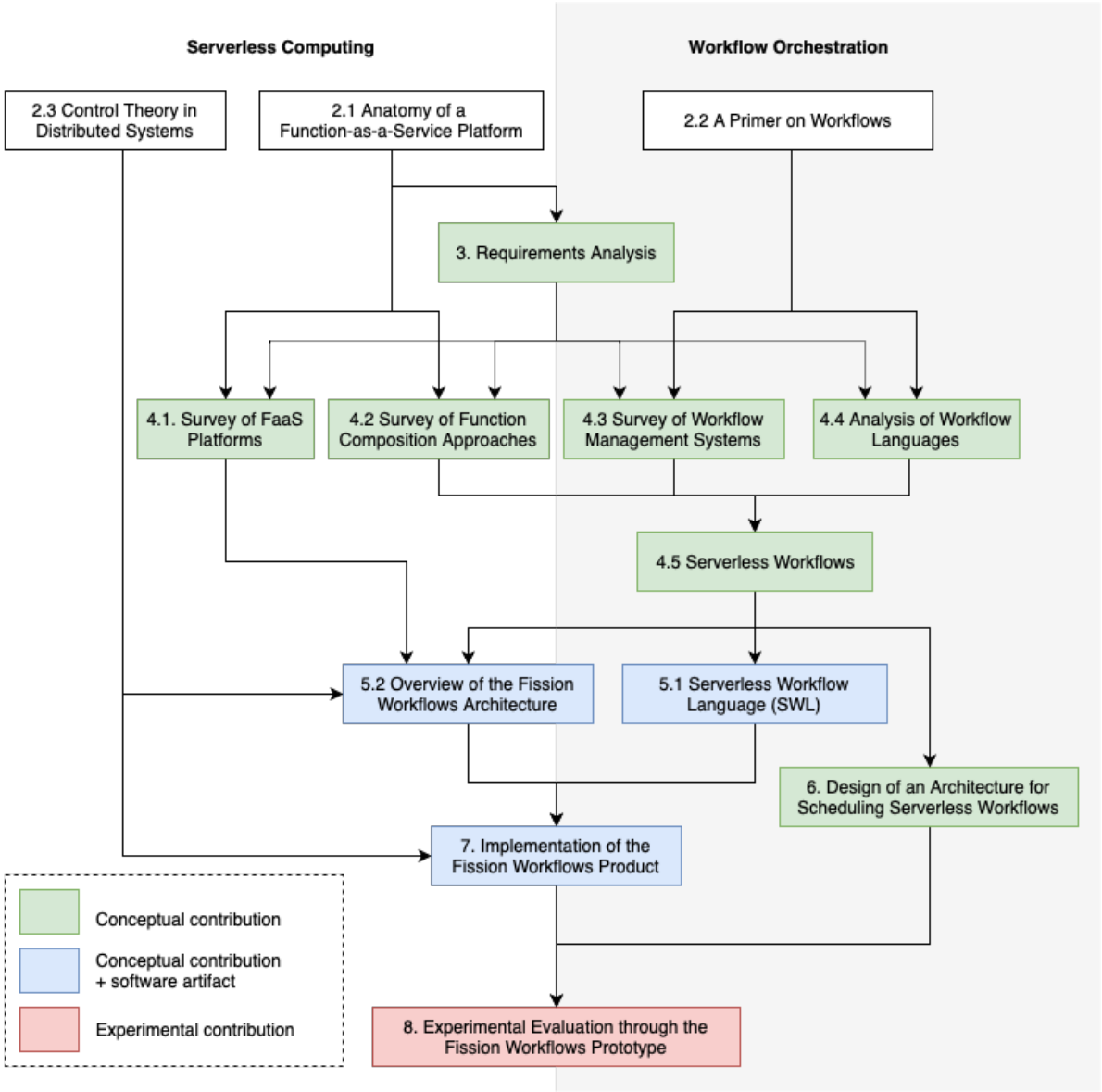


Figure 1.1: The structure of this thesis. An arrow pointing from section A to section B indicates that section B builds upon section A.

2

Background

Serverless computing did not spontaneously materialize—it is the result of various developments in existing fields, including virtualization, cloud computing, event-driven architectures, and workflow orchestration [134]. Similarly, the orchestration of functions in this emerging domain is not an entirely new problem either; the composition of applications, tasks, functions, and queries has been tackled in diverse ways in other domains. In the remainder of this section, we provide background on these related topics relevant to one or more of the contributions in this thesis:

- We continue the introduction to serverless computing with a deep-dive into the **internals of a FaaS platform** in Section 2.1. The goal for this background section is to improve the functional and non-functional properties of our FaaS function composition prototype (Chapter 5, Chapter 6, and Chapter 7), we need to have a thorough understanding of these internals of these FaaS platforms.
- In Section 2.2 we provide a survey of the current landscape of **workflows**. This section is relevant throughout the thesis, because we identify *serverless workflows* as a novel type of workflows, propose a workflow-based approach for cloud function composition (Chapter 4), and design and implement a prototype workflow-based composition engine (Chapter 5 and 7).
- Section 2.3 is an introduction to **control theory** and, specifically, its application in distributed systems. Its relevance can be found in both the prototype (see Chapter 5 and Chapter 7) and the underlying systems (Fission, Kubernetes)—where control theory is used to improve the resiliency of these distributed systems.

This chapter draws extensively on our prior work. Section 2.1 on FaaS principles and platforms incorporates the definitions and concepts introduced by the vision papers of the SPEC RG CLOUD [132, 133], and work with the AtLarge team [134]. The FaaS reference architecture (Section 2.1.4) was adapted from the SPEC RG CLOUD work on a FaaS reference architecture. Section 2.2 bases much of the workflow formalism descriptions on our prior work on workflow formalisms [136].

2.1. Anatomy of a Function-as-a-Service Platform

Although we already introduced Function-as-a-Service (FaaS) and serverless computing in the introduction of this work (Section 1), in this section we dive deeper into the FaaS platforms. The FaaS model is key within serverless computing, to the point that part of the community sees serverless being a synonym for FaaS.

Because the focus of this thesis is to find a general approach to function composition for serverless (or FaaS) functions, it is key to have a thorough understanding of the space. To this end, the goal of this section is to uncover the key concepts, components, and workflows within FaaS platforms. With this understanding we can improve our analysis of the specific constraints, caveats and opportunities that a FaaS platform provides to the composition approach. This background information is therefore relevant throughout the thesis.

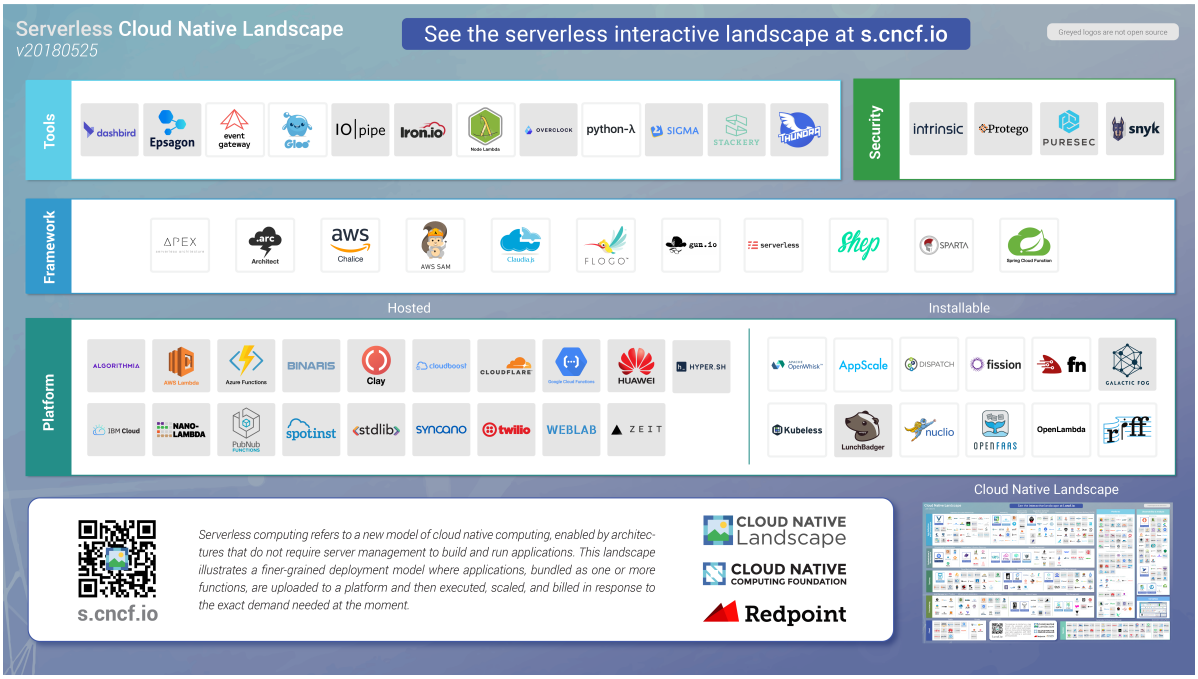


Figure 2.1: An informal overview of the state-of-the-art in serverless computing as of July 2018 (reproduced from https://github.com/cncf/landscape/).

This background section limits the scope to focus specifically on the internals (conceptual and technical) of a FaaS platform. For a higher-level overview of the serverless computing field see the CNCF Serverless WG white-paper [139], or our prior work [132, 134].

The FaaS landscape is diverse and rapidly evolving[1]. Next to ongoing work of our group to create comprehensive survey of the field [135], the CNCF serverless WG has published an informal survey (in the form of a visual landscape, as reproduced in Figure 2.1) of the current industry-based FaaS platforms. Figure 2.1 visualizes one of the few clear distinctions that can be made in the approach taken by FaaS platforms. Many cloud providers offer their own closed-source FaaS platform as part of their cloud offer (see AWS Lambda, Google Cloud Functions, Azure Functions). On the other hand, there are FaaS platforms that choose a vendor-neutral, open-source approach. In addition to this survey, several FaaS platforms have emerged from research: Snafu by Spillner [122], OpenLamda by Hendrickson et al. [63], and SAND by Akkus et al. [23]

Based on an initial FaaS platform survey (described in Section 4.1), our prototype (see Chapter 7) builds upon the Fission: one of the leading, open-source FaaS platforms, which focuses on performance and simplicity for developers. To provide the reader with an intuition how the abstract concepts of FaaS model works out in practice, we use Fission as a case study throughout the remainder of this section.

2.1.1. Principles of Serverless Computing

In our prior work [132] (and in parallel work by others [32, 139]) we found that serverless computing has three key characteristics:

1. **Granular billing** (or usage-based billing): the user of a serverless model is charged only when the application is actually active.

2. **Minimal operational logic**: operational logic, such as resource management and auto-scaling, is delegated to the infrastructure, making those concerns of the infrastructure operator.

3. **Event-driven**: interactions with serverless applications are designed to be event-driven; the serverless application only is deployed and active when events need to be processed.

[1] From the start to the end of this thesis, many new FaaS platforms have appeared, radically changed, or discontinued.

To illustrate, FaaS contains all three serverless characteristics: (1) users only pay for the milliseconds of CPU and memory used to execute the function; (2) users do not have to manage nodes or resources associated with the function; (3) functions are only executed when triggered by an event (such as a HTTP request or a pub-sub message). In addition to adhering to these three key characteristics of serverless computing, the FaaS model provides the user with a key primitive: a serverless function.

2.1.2. A Serverless Function

The notion of the *function* in FaaS has been a key part of its success. The flexibility of allowing users to define arbitrary functions to be deployed, while at the same time offering guarantees to the cloud operator by placing intentional constraints on the model, makes the FaaS model an attractive proposition to both parties.

The essence of a serverless function[2] is that it creates an explicit decoupling of the abstract computation from the resources. Other popular primitives in this domain (virtual machines, containers), require users to carefully construct a deployment environment for (potentially) long-running processes. Instead, in FaaS, a serverless function is similar to how we think about functions in programming languages and mathematics: functions represent an abstract computation, leaving the mapping of it to concrete resources to a specialized component (such as a compiler or CPU).

From a FaaS operator's perspective, a function is a black-box with specific guarantees. Although how cloud operators also view containers and virtual machines of users as black-boxes, the FaaS model offers guarantees and constraints to this black-box model:

1. **Finite runtime**: By enforcing time constraints, cloud operators are ensured that functions will run within a specific time interval. This upper-bounded time constraint allows the cloud operators to improve prediction and management the resources assigned to these functions.

2. **Stateless**: Functions do not hold (vital) state across functions executions. Although, some FaaS platforms do not explicitly prohibit passing of state between function executions, a user is not guaranteed that the state exists. This allows cloud operators to safely destroy, create, and migrate functions without needing to be prevent the loss of state.

3. **Atomic**: Serverless functions, similar to their programming language counterparts, are considered to be an atomic unit of execution. When a function is executed, it runs to completion, returning either a result or an error. The explicit absence of a complex manipulation of the function, preserves the intuitive execution model for users. At the same time, this model allows cloud operators to relatively easily implement and maintain support for functions in multiple programming languages.

The consequence of the constraints and the black-box model, is that the execution model of a FaaS function is intuitive to understand. A function can be represented as taking in arbitrary input (along with an optional context, which includes information about the execution, such as deadlines or SLAs), and returning output. Similarly, errors are generally not treated as a special control flow. Instead, errors are treated as values, which can include error codes, error messages, and other details to help clarify the root cause of the error.

Indeed, the conceptual model of a FaaS function builds upon *functions* in functional programming, and—even more abstractly—the notion of *lambdas* notions in lambda calculus [94]. Due to the limited scope of this thesis, further research into the implications and opportunities of these conceptual parallels is considered future work.

It is beyond the scope of this work to provide a detailed specification of this FaaS model. Several initiatives have emerged to address this issue, such as by the CNCF Serverless WG [139], FaaSlang [69], and AWS [117]. However, at the moment of writing, the community has not yet reached consensus on a comprehensive specification. This lack of consensus could also signal that the function model as it is, is still missing features or failing to address all concerns. For example, should we distinguish between pure and impure functions? Can we (and should we) adapt the model to fit data-intensive and edge computing use cases?

[2]throughout this work we use "serverless function", "cloud function", and "FaaS function" interchangeably

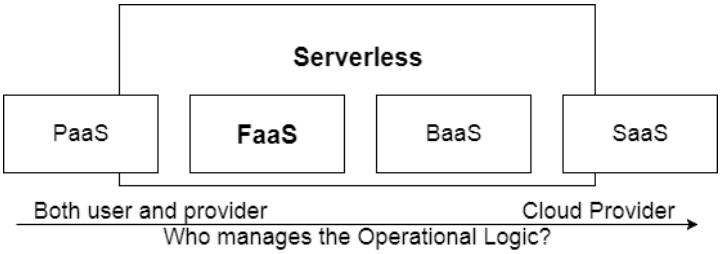


Figure 2.2: The FaaS model in relation to serverless computing (reproduced from [132]).

Although the notion of a function is by no definition novel, the use of it as a explicit, self-contained primitive in distributed systems is. Even though functions (or queries, or tasks) have been introduced and reintroduced over the past decades, each raises important conceptual and technical challenges related to operation according to the principles of FaaS [134]. Common Gateway Interface (CGI) provided a language-agnostic plug-n-play approach to deploying 'functions' behind a HTTP server, but it lacked the self-contained, distributed nature of serverless functions. Others, such as stored procedures in databases, IBM CICS transactions, and Google App Engine background tasks, are constraint to a specific environment or system. Closest to FaaS, Remote Procedure Call (RPC) and Remote Method Invocation (RMI) [127], provide a generic notion of invoking a function, but lack cross-language support and separation between business and operational logic.

2.1.2.1. A Fission Function

In Fission the function model is implemented on top of the HTTP protocol, in a way similar to a REST service[3]. The reasons for this are three-fold: (1) HTTP support is ubiquitous across programming languages, allowing Fission to support functions written in most programming language; (2) It is intuitive to software developers, who are generally familiar with working with HTTP communication and REST-based interfaces; And, (3) the synchronous request-response execution model of HTTP [111] maps well to the function model.

The function request is represented as a HTTP request, in which the body contains the input to the function, and any metadata is mapped to HTTP headers and URL query values. The function itself is wrapped in a programming language-specific HTTP application server (called an *environment*), which exposes a consistent interface for executing the function. On the completion of the execution, the function returns the function output again as a HTTP response. The success or failure of the function execution is mapped to the HTTP status code (any code equal or larger than 400 signalling an error). The output or error message is mapped to the response body, and the metadata is mapped to HTTP headers.

2.1.3. FaaS within the Serverless Ecosystem

As visualized in Figure 2.2, FaaS plays a central role within serverless computing. However, serverless computing does not consist solely out of FaaS; the serverless ecosystem contains a wide diversity of models and systems which adhere to the three characteristics of serverless computing: no operational logic, pay-per-use, and event-driven [132].

Within the field of expressing *computation*, FaaS is not the sole 'serverless' option. We could choose a more higher-level model to express computation in a specific context, such as using queries to serverless databases or Backend-as-a-Service (BaaS) or even certain Software-as-a-Service (SaaS) products. On the other hand and somewhat more controversial, cloud providers[4] have started offering *serverless containers*, arguing that containers for which the user does not manage or maintain the physical resources should also be considered as serverless computing.

Besides the computation, other aspects of applications call for their own solutions in serverless computing. For the expression of data flows in a system, a serverless ecosystem relies on the correct and efficient distribution of events (for example, AWS SQS). Similarly, at rest, the serverless ecosystem contains solutions that offer data storage following the principles and constraints of the serverless model

[3] `https://en.wikipedia.org/wiki/Representational_state_transfer`

[4] See `https://aws.amazon.com/fargate/` and `https://azure.microsoft.com/en-us/services/container-instances/`

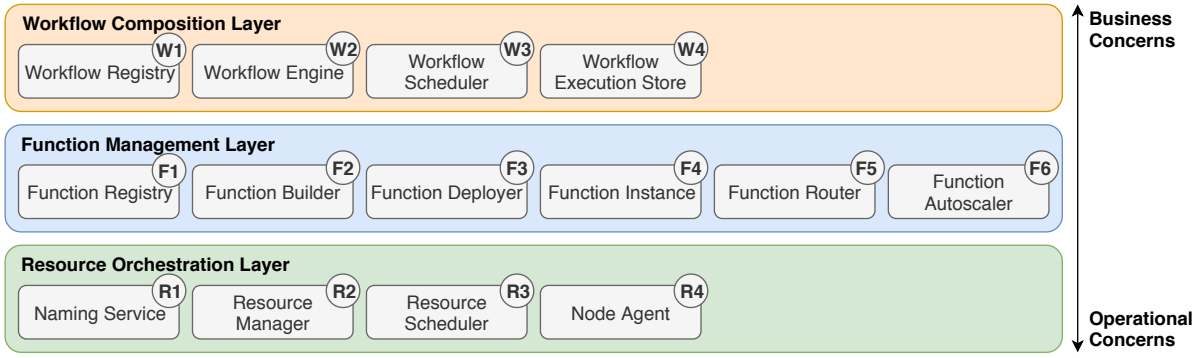


Figure 2.3: A reference architecture of FaaS platforms (reproduced from [135]).

(for example AWS Aurora).

With FaaS starting to mature, industry and academia have started to realize the need for a broader serverless ecosystems surrounding the FaaS platform. In the academia, researchers are extending and altering the FaaS model to handle data [116] more efficiently and decrease the performance overhead [24]. In the industry, serverless options have emerged in every conceivable domain[5], from authentication, to monitoring, to message gateways. A recent example is the emergence of the Knative[6] platform, which consists of three main serverless pillars: *serving* (the traditional FaaS); *eventing* (management and delivery of events); *build* (source-to-artifact build orchestration).

2.1.4. Architecture of a FaaS Platform

Although the landscape of FaaS platforms in Figure 2.1 appear diverse, the components within these platforms are rather similar. In work with the SPEC RG CLOUD group—covered in [133], and fully described in work under submission at IEEE Internet Computing [135]—we have created a reference architecture for this emerging type of systems.

The result of this work is reproduced in Figure 2.3, in which we identified the key components across FaaS platforms. Though descriptions for each of the components can be found in our on-going work[135], we focus here solely on the *Function Management Layer*[7].

The *Function Management Layer* contains the core components responsible for the (operational) lifecycle of individual FaaS functions: deploying *function instances*, executing functions triggered by events, and elastically scaling functions up and down.

Whereas the *Resource Orchestration Layer* is concerned with the management of arbitrary resources, the Function Management Layer manages arbitrary functions. At this layer, the components rely on the lower-level layer for the correct management of the resources.

At its core, the Function Management Layer consists of:

1. **Function Registry**: serves as a local or remote repository of functions. In practice, this registry is often further split into a *function metadata store*, for low-latency look-ups of function metadata, and a *function store*, which contains the binaries of the function (the *function-code*).

2. **Function Builder**: transforms function sources into deployable functions. Functions typically have to undergo transformations (e.g., compiling, validating, dependency resolving) before they are stored in the *function registry* or deployed by the Function Deployer.

3. **Function Deployer**: ensures an instance of an arbitrary function, a Function Instance (see next component), is deployed. The Function Deployer combines the configuration stored in the Function Registry, the parameters supplied by the requester, and other factors into a decision of how

[5]For more details see the serverless landscape in Figure 2.1 or the community-curated serverless list (https://github.com/anaibol/awesome-serverless).

[6]`https://github.com/knative/`

[7]The *Workflow Orchestration Layer* in the reference architecture is largely influenced by this thesis and based on prior work described in Section 2.2.4

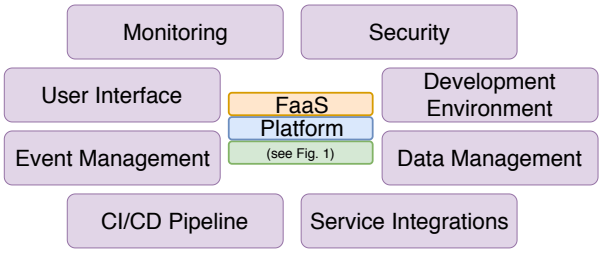


Figure 2.4: A reference ecosystem of FaaS platforms (reproduced from [135]).

the function should be deployed, determining, among others, how many and what kind of resources it should receive. Though it decides *how* the Function Instance should be deployed, the deployment of the Function Instance itself is delegated to the Resource Orchestration Layer.

4. **Function Instance**: is a self-contained worker—typically a container—capable of handling function executions. For scalability, a function can have multiple, concurrent Function Instances deployed.

5. **Function Router**: routes incoming requests or events (internally) to the correct Function Instance. If no Function Instance is available, the Function Router queues the events to await the deployment of new instances.

6. **Function Autoscaler**: monitors the demand and supply of resources, and elastically scales the number of Function Instances—adding or removing instances as needed.

2.1.4.1. FaaS Reference Ecosystem

The reference ecosystem in Figure 2.4 positions the FaaS platform (described by the reference architecture) as the functional core of a broader FaaS ecosystem. Production-grade FaaS platforms, such as those from major cloud providers, contain additional components to improve developer experience, maintainability, event management, and data management [135].

Components in the reference ecosystem include:

1. **User Interface**: allows users to manage and interact with the FaaS platform. In practice, user interfaces for FaaS platforms are diverse, ranging from simple command-line interfaces, to full-fledged editors and visual interfaces.

2. **Developer Environment**: reduces the difficulty of developing functions and workflows. The FaaS platform achieves this by integrating with popular editors and tools, providing local FaaS emulators for proprietary FaaS platforms, and providing various tools to further improve the developer workflow.

3. **Event Management**: manages the lifecycle of the events that trigger function executions on a FaaS platform. Although the FaaS reference architecture deploys and executes functions based on these events, it does not manage the lifecycle of these events. Where and how the events are sourced, processed, and routed (to eventually arrive at the FaaS platform) is delegated to an Event Management system.

4. **Data Management**: manages the lifecycle of data in the FaaS ecosystem—how the data is stored and transferred. Ideally, a Data Management system stores and transfers data efficiently in-between the functions, ensuring low-latency and availability for a low cost. Current approaches range from reuses of general microservice architectures, to simple integration with an external data storage service, such as AWS S3, to uses of advanced hardware [83].

5. **Continuous Integration (CI) and Deployment (CD)**: manages the lifecycle of function sources. Although the FaaS reference architecture includes the Function Builder component for transforming source code into a deployable function, this represents only a small and final step in the overall code-lifecycle in larger organizations. Depending on those processes, the CI/CD pipeline typically

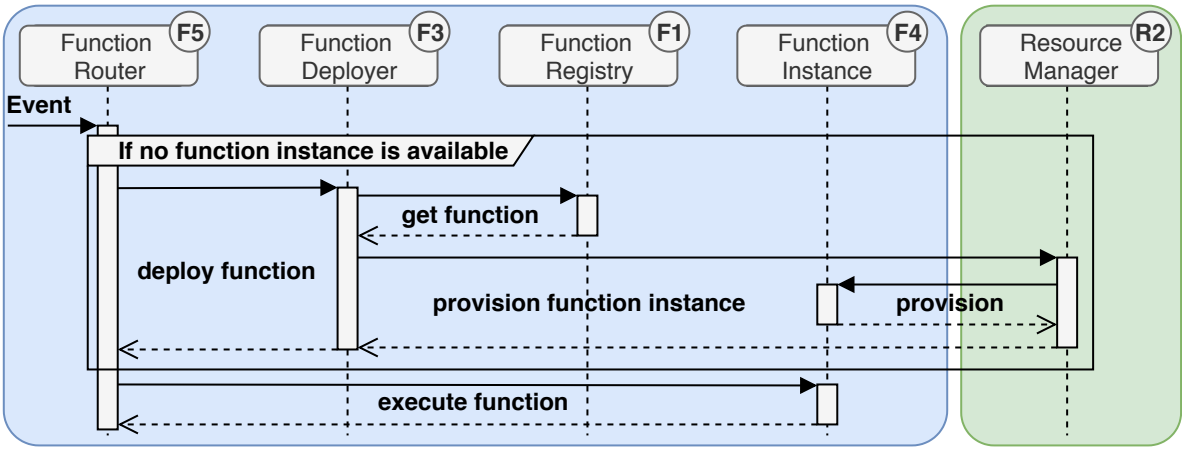


Figure 2.5: The function execution operational pattern function (reproduced from [135]).

consists of staging environments, complex dependency resolving, multiple stages of testing, governance checking, and (manual) approval steps—before the function is submitted to the Function Builder.

6. **Service Integration**: manage integrations between the FaaS and other cloud services. FaaS platforms typically include off-the-shelf integration with other cloud services, for example, AWS Lambda fully integrates with AWS' authentication and authorization service, allowing users to call other AWS services from their functions without an additional security process[8].

7. **Monitoring and Logging**: monitors and logs metrics on arbitrary layers and components, for live or later visualization and analysis. The collected data is typically used for billing customers, alerting maintenance in case of anomalies, and providing users with data for their own operational analysis.

8. **Security**: ensures proper user (and function) authorization and authentication to access and change parts of the FaaS platform.

2.1.4.2. Operational Patterns

In this section we describe the *operational patterns* that we identified in [135]: common solutions that FaaS developers have included in their FaaS architectures to address prevalent problems. The patterns we identify show how the components of the reference architecture can interact with each other to facilitate common functionality.

In the remainder of this section we present the two operational patterns relevant for this background section, namely function building and function execution:

1. **Function building**: A common operational pattern is that of a user submitting functions, as source-code, to the FaaS platform. The source-code traverses the CI/CD pipeline defined by the user (see Figure 2.4) before arriving at the FaaS platform. After initial validation, the source-code is sent to the Function Builder, which transforms it into a deployable function.

   This process depends on the implementation of the FaaS platform—it can be non-existent—but typically involves build steps that are language-specific, such as compiling Java to bytecode or resolving Python dependencies. Once the building process has completed, the resulting artifacts are stored in the Function Registry. From there, the function is available to the FaaS platform as a deployable function.

2. **Function execution**: Function execution in FaaS (Figure 2.5) inherently involves the function deployment, as function instances scale up from zero and scaled back to zero after inactivity. First, an event, such as a HTTP request, arrives at a Function Router, which triggers the deployment

[8] https://docs.aws.amazon.com/lambda/latest/dg/accessing-resources.html

of a function instance if none is available—a *cold start*. Then, the Function Deployer fetches the function (metadata) from the Function Registry to decide the appropriate configuration of the function instance. It then tasks the Resource Manager with ensuring that a function instance with the appropriate configuration is deployed. Once the function instance is fully deployed, the event can be passed to it to start the function execution.

If a function instance is already available for handling the execution, the expensive *cold start* process can be bypassed. The Function Router directs the event to the existing function instance—be it directly over RPC, or using a message queue.

2.1.5. Technical Considerations

In general, FaaS platforms all focus on three different characteristics: resiliency, performance isolation, and performance.

2.1.5.1. Fault Tolerance

FaaS projects tend to avoid reinventing the wheel; deferring as much operational logic as possible to a resource manager—which in many cases is Kubernetes. In such cases, the *function scheduler* maps the function not to actual resources, but to the primitives used by the resource manager. By deferring the concerns, FaaS platforms are generally relatively small projects (though still complex, due to the extensive dependencies and integrations), and inherit the resiliency and fault-tolerance mechanisms of the underlying resource manager.

2.1.5.2. Performance Isolation

For functional and performance isolation, nearly all platforms have taken similar technical considerations, which are seen as 'best practices' in the industry. Function instances are generally using containers for isolation (with runtimes like Docker, LXC or RKT). The use of container is seen as a middle-ground between performance (it is slower than non-sandboxed processes) and performance isolation (it is less safe than full virtualization) [53]. Optimizing this trade-off of performance and performance isolation, is an active research area, with Koller et al. [84] proposing removing unnecessary overhead of the (Linux) operating system from the virtualized function instance.

2.1.5.3. Performance

Despite the benefits of the extensive dependencies on performance isolation techniques and resource managers, these tend not to have been designed for the performance requirements needed for FaaS workloads [130]. For this reason, FaaS platforms have various performance optimizations:

1. **Function instance reuse**: Instead of deploying a function for each request, platforms keep the deployed function instance running for some time (known as the *cooldown*) in the anticipation of subsequent requests. A function request that ends up in an already deployed function instance is said to have a *hot start*, in contrast to a request that has a *cold start* having to wait for the function instance to deploy.

2. **Function runtime pooling**: Instead deploying a unique function instance for each type of function, we generalize the function instances as much as possible. Instead of having a *python-A* and a *python-B* function instance, only differing in a single installed package, we deploy generic *python* runtime, which we *specialize* into *A* or *B* just before executing the function. This eliminates a large part of the cold start process, while being able to use the generic *python* runtimes for multiple functions allows the pool of generic function runtimes to remain small.

3. **Function prefetching**: To deploy a function, the FaaS platform needs to download the function from the registry and transfer it to the machine where the function instance will be deployed. Although for simple, small functions this downloading is quick, functions with many static sources or dependencies can quickly grow to be 10s or 100s of MBs in size—making the function transfer a large (if not the largest) contributor to the deployment time. FaaS platforms can minimize the time spent on fetching these functions by caching near the machines that need them ahead of time. In previous work [20], we looked at more granular prefetching: fetching specific parts (in this case specific Python dependencies) to optimize the amount of time spent on downloading dependencies for the most required dependencies.

4. **Predictive autoscaling**: Because the FaaS platform has full control over the deployment lifecycle of the function instances, it also has extensive control over the methods to scale the resources assigned to a function. Building on extensive existing work for both reactive and predictive autoscaling [108], FaaS platforms can improve the performance of functions at scale by utilizing their knowledge of the function, workload and deployment environment to predictively deploy (*prewarm)* function instances. We explore prewarming further in Chapter 6.

Whether these considerations, and the architecture, hold true for closed-source platforms is difficult to judge at the moment of writing. However, it seems likely that these platforms follow the same architecture, because both Knative (derived from Google's internal FaaS, and underlying Google Cloud Run) and OpenWhisk (underlying Bluemix Functions and Adobe Functions) seem like representative of enterprise-level FaaS platforms. Furthermore, Wang et al. [137] found that the largest cloud providers use a combination of virtual machines and containers to isolate functions, and use similar performance optimizations.

2.2. A Primer on Workflows

Because we decide further in the thesis on the use of workflows to represent function compositions, the goal of this background section is to provide an overview of the field of workflow orchestration, and its applications. For this work we focus specifically on two key areas within the workflows domain: (1) surveying the approaches to defining and expressing of workflows (relevant for Chapter 4 and Chapter 3); and (2), investigating the internals of state-of-the-art WMSs (relevant for Chapter 5 and Chapter 7). For the survey on workflow approaches, this section draws extensively from joint work with Laurens Versluis, in which we performed a comprehensive survey of workflow formalisms [136].

In this primer we cover the following key aspects of workflow orchestration that are most relevant for this thesis:

1. Section 2.2.1 describes the abstract notion of a workflow and related concepts. It also introduces workflow patterns that will be referred to throughout the thesis.

2. In Section 2.2.2 we survey the workflow formalisms used to represent workflows: BPMN, Petri Nets, and DAGs. Based on previous work [136], we focus on the support of complex workflow patterns, usability, and the extensibility of each formalism.

3. We examine approaches to workflow development and the different stages of workflow processing in Section 2.2.3.

4. Finally, in Section 2.2.4 we provide an overview into components that make up a workflow management system, in light of the workflow engine prototype.

2.2.1. Workflow Concepts

On an abstract level, a *workflow* consists of a set of interdependent *tasks* that need to be executed to achieve a specific goal. [36] This goal can be a final result or the side-effects produced by the tasks. A task within a workflow has the following properties: dependencies on the software or service used by the task to perform its computation (software flow), dependencies on data (data flow), and dependencies on other tasks (control flow).

These tasks can be arranged in arbitrary ways to express more complex dependency relations. The *workflow patterns* depicted in Figure 2.6 are fundamental patterns, and are supported in even the most basic WMS [36]:

1. **Parallel**: tasks that do not have a dependency relation can be executed in parallel.

2. **Pipeline**: tasks that form a chain or sequence indicate that each task depends on the completion of the task preceding it. Given this relationship, the WMS can only execute these tasks one-by-one (in serial).

3. **Data Distribution** (or fan-out, or scatter): multiple tasks depend on the completion of a single *scatter* task. This fan-out of a single to N tasks allows workflows to transition from a pipeline to a parallel execution.

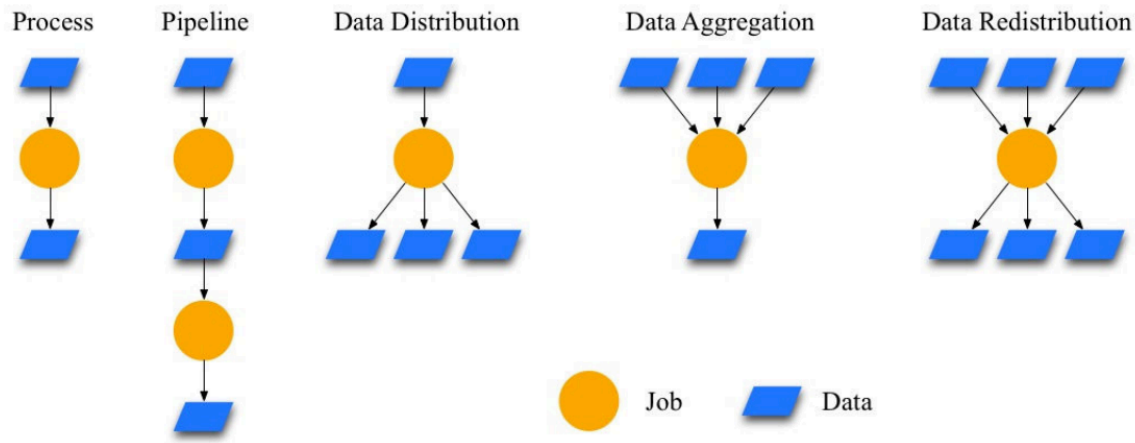


Figure 2.6: Primitive workflow patterns (reproduced from [36]).

4. **Data Aggregation** (or fan-in, or gather): a single *gather* task depends on two or more tasks; it only starts executing once all tasks have completed. This allows the synchronization of the control flow and the collection of the data generated by the parallel tasks.

By combining these workflow patterns, users can create arbitrary complex workflows. This allows workflow users to model complex processes, such as processing satellite imagery (Montage [35]) or gravitational wave detection (LIGO [61]).

In addition to this minimal definition of a workflow, numerous extensions have been proposed: conditional primitives to support loops, conditional branches; exception handling primitives to allow workflows to introduce complex error handling structures; signal handling primitives to allow external events to influence the workflow at runtime; attaching non-functional requirements to tasks or workflows to influence the decisions made by the workflow scheduler; and, complex data transfer primitives to alter the way tasks fetch and emit their data. With these extensions, surveys, such as the Workflow Pattern Initiative by Van der Aalst et al. [114], report close to a 100 distinct workflow patterns.

Regardless of the supported extensions, the WMS offers users the ability to express their computations in a declarative way, explicitly separating control and data flow. Expressing applications and processes as workflows has become a well-established practice in various disciplines, including scientific computing, big data processing and business process management [91].

2.2.2. Workflow Formalisms

Besides the different types of workflow patterns and features, there are multiple ways to express workflows. These *workflow formalisms* fit different types of workflows better. The most prevalent among these workflow formalisms are BPMN, Petri Nets and DAGs [136].

A workflow formalism is a formal grammar, which allows users to express the workflow concepts using consistent, machine-readable semantics. There are several reasons for establishing formalisms for workflows, including allowing users to work with common syntax that is parse-able by workflow management systems.

2.2.2.1. BPMN

The BPMN Version 2.0 (BPMN 2.0) is a visual standard for designing and modeling (business) workflows [103]. It is often used to design workflows at a high-level, targeted at human readability. Furthermore, it is most used business process model [119]. With its GUI-only based focus, users use complex editors to visually alter their BPMN workflows.

Although in this work we argue for a workflow type with more complex workflow pattern support, we avoid the use of BPMN for several reasons. As argued in our prior work [136], BPMN is highly focused on the business modeling domain, and generally not popular outside of that. It is overly complex, containing unnecessary and obscure primitives [143], many of which are definable using other BPMN 'primitives' [39]. This complexity makes it not only difficult for users to understand these workflows,

```
version: '2.0'
create_vm_and_send_email:
  type: reverse
  input:
    - vm_name
    - image_id
    - flavor_id
  output:
    result: "<% $.vm_id %>"
  tasks:
    create_vm:
      action: "nova.servers_create name=<% $.vm_name %>"
      publish:
        vm_id: "<% $.id %>"
    search_for_ip:
      action: nova.floating_ips_findall instance_id=null
      publish:
        vm_ip: "<% $[0].ip %>"
    associate_ip:
      action: "nova.servers_add_floating_ip server=<% $.vm_id %>"
      requires: [search_for_ip]
    send_email:
      action: "send_email to='admin@mysite.org' body='Vm is created'"
      requires: [create_vm, associate_ip]
```

Figure 2.7: An example of a DAG-based workflow in the OpenStack Mistral WMS written in YAML.

but also leads to vendors of BPMN-supporting workflow languages and WMSs implementing BPMN features in different ways (or not at all) [140].

2.2.2.2. Petri Nets

Petri nets are state-transition systems that extend elementary nets [96]. They are useful as a tool for describing and studying information processing systems that feature concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic characters.

Similarly to BPMN we refrain from using Petri Nets in this thesis. Although the arguments against Petri Nets are less strong than against BPMN, in our prior work [136] we found several characteristics of Petri Nets—popularity within the distributed systems (academic and industry) community, and its complexity—that make it less favourable than using DAGs.

2.2.2.3. Directed Acyclic Graphs (DAGs)

The DAG formalism is the least complex of the workflow formalisms [136]. It contains only a couple primitives needed to express the workflow patterns discussed in Section 2.2.1 and visualized in Figure 2.6. As shown in the real-world example Mistral workflow (Figure 2.7), with the DAG formalism the workflow definition explicitly defines the tasks and the dependencies to form a directed acyclic graph.

However, a consequence of this simplicity is also that these DAG-based workflows are the least expressive; there are no primitives to support complex control flow logic, such as conditional branching or loops. Typical uses of the DAG formalism do not need complex control flow constructs; for example, scientific workflows are generally built for a highly-specific 'flow', avoiding the need for conditional branching or loops [36].). However, for other use cases, the formalism must be extended to support them.

Next to the least complex formalism, the DAG formalism is also the least well-defined format: no community-wide specification exists. This has caused the DAG-based WMSs and research to propose various variations of the DAG formalism. Yu et al. [141] argue that besides the supporting parallel and sequential structures, a DAG-based workflow also supports a choice-structure—executing a specific

Properties	BPMNs	Petri Nets	DAGs
Complexity	6	4	2
Utilization	$< 6\%$	100%	100%
Supports loops	Yes	Limited	No
SNFR	Limited	No	No
Primairy domain	Business	Computer Science	Computer Science
Popularity	4%	3%	38%

Table 2.1: Comparison of BPMNs, Petri Nets and DAGs based on the metrics related to extensibility (reproduced from [136]).

task when specific conditions are met. Pegasus with its DAX workflow language extends the formalism with, among others, resource definitions (catalogs) [50]. OpenStack Mistral [5] (see Figure 2.7) and Azure Logic Apps both provide expression support to allow users to perform data selections and manipulations without needing explicit tasks for those [3]. Other systems extend the DAG with notions described in Section 2.2.1: sub-workflows, workflow interrupts, and advanced error handling.

2.2.2.4. Formalism for Serverless Workflows

In this work we focus our efforts on utilizing DAGs to model serverless workflows (see Section 4.5). In our prior work [136] we already found that extending the DAG formalism for non-functional requirements is favorable, compared to BPMN and Petri Nets, as reproduced in Figure 2.1. The *complexity* is calculated based on the number of primitive constructs (or symbols) in the formalism. The *utilization* is measured by the number of these constructs needed to represent common workflows. Seen as making the execution model of workflows (needlessly) complex the *supports loops* metric indicates if the formalism supports unbounded loops in workflow definitions. The *Support for Non-Functional Requirements (SNFR)* metric measures the formalism support for user-defined, task-level, non-functional requirements (such as task versioning strategies and security requirements). Finally, the *popularity* metric provides a measure of traction that each of the formalisms has in the distributed systems community, which was gathered by analyzing top distributed systems conferences over the last years[9].

Although the specific goal of the prior work was to compare the workflow formalisms to find the most suitable formalism to support task-level NFRs, the comparison also allows us to conclude that the DAG formalism should be used for serverless workflows. The DAG formalism is the most usable, as it provides the similar utilization with a minimal set of constructs. It does not impose a complex execution model with unbounded loop support and other complex control flow constructs built-in. Finally, it is most popular within the target domain of distributed systems by a wide margin. Unsurprisingly, the current state-of-the-art WMSs that are closest to the domain of serverless workflows (such as Apache Airflow, OpenStack Mistral [5], and AWS Step Functions [2]) all explicitly use DAGs to model workflows.

2.2.3. Workflow Definition and Development

Similar to highly-expressible, high-level programming languages being compiled to low-level machine code, how users express workflows often differs from the the workflow definitions that end up executed. This process, moving from user-level workflow descriptions to executable workflows, is visualized in Figure 2.8. Based on the taxonomy of Yu et al. [141] and the Pegasus process [50], the process consists of three phases: design, build, and execute. The design phase comprises the user expressing a workflow in a (high-level) workflow language. The build phase consists of the workflow getting processed into an abstract workflow definition. Finally, in the execute phase the *abstract workflow definition* is turned into an executable, *concrete workflow definition*.

2.2.3.1. User-Level Workflow Language

While these systems generally share similar approaches to scheduling, task execution, and monitoring, they vary widely in their approach to how users are required to express workflows. Based on existing work and systems, we identify the following categories of approaches to user-level workflow languages:

1. **Data Serialization Language**: A frequently occurring approach to workflow definition is to have the user define the workflow using a data serialization language (such as JSON, YAML, or XML).

[9] For more details on the setup and results of the comparison see [136].

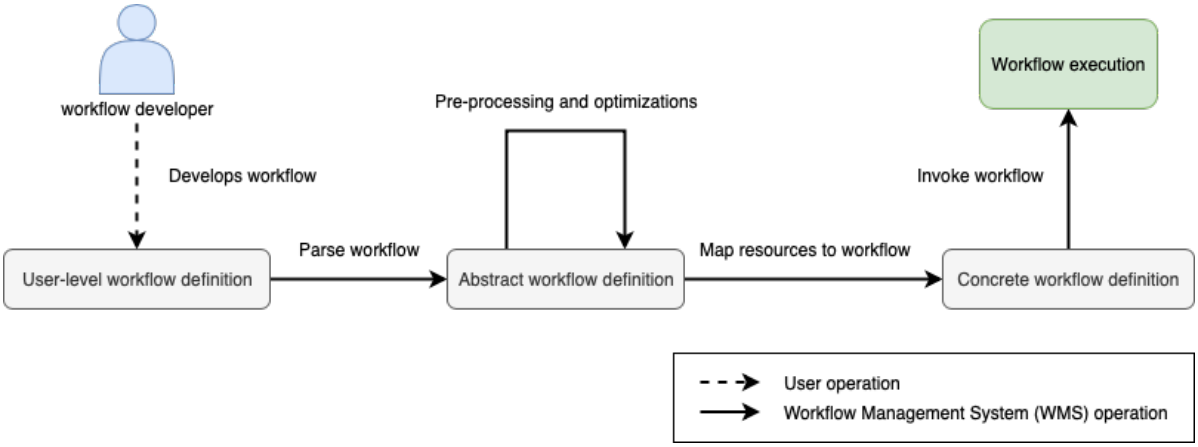


Figure 2.8: The workflow lifecycle: from development to execution.

Examples of WMS using this approach: OpenStack Mistral [5], AWS Step Functions [2], Azure Logic Apps [3], and CWL [27]. From the workflow system's point of view this approach is easiest to implement, as it translates (almost) one-to-one to the abstract workflow definition. From the user's point of view, one argument in favor of this approach is that it allows them to implement their own specific higher-level workflow languages and compile them down to these easily-parseable data formats.

2. **Programming Language Library**: Similar to using a data serialization language, some WMSs offer a library or module in a popular programming language to express the workflows, with examples being: Luigi [17], Apache Airflow, and Pegasus [50]. Although the user is still defining the workflow explicitly with the help of the workflow API, the user can make use of programming language constructs (e.g., encapsulation, abstraction, iteration) to reduce the effort of writing (complex) workflows. A key challenge for this type of workflow expression is to ensure that users understand the constraints of the workflow definition; often not all features of the programming language are supported within the definition of the workflow.

3. **Domain-Specific Language**: To avoid a library artificially constraining an existing programming language to define workflows, some WMSs opt to define their own Domain-Specific Language (DSL), such as Azkaban [10][10], and WDL [70]. Although with a DSL the workflow developers can carefully craft the workflow language to fit the workflow model, making it more expressive than a data language and more fitting than a library approach, this approach does lack existing tooling, such as a linter, editor, or parser—all of which have to be developed.

4. **User-Defined Orchestrator**: Extending the *programming language library* approach beyond its limits, some WMS systems (e.g., Azure Durable Functions, and AWS SWF) support users expressing workflows imperatively. The advantage of this approach is that it allows users to use all the control flow features of the workflow language, such as conditional branches, loops, and error handling. However, the challenge with this approach is to accurately translate this the workflow generated by the orchestrator into an abstract workflow, something that has to be achieved by simulation, static analysis and inference, or constraining the orchestrator to—again—use a specific API within the programming language.

5. **Visual Programming**: A WMS can avoid textual definition of workflows all together by offering a visual editor to compose workflows [141]. This is particularly popular in the domain of business processing [138], where WMSs often target non-expert, non-technical users as to define workflows. These users, often lacking any programming experience, may find it more intuitive to express and understand workflows in a visual way.

[10] As of September 2018, Azkaban is moving from their DSL-based Flow 1.0 language to YAML in Flow 2.0

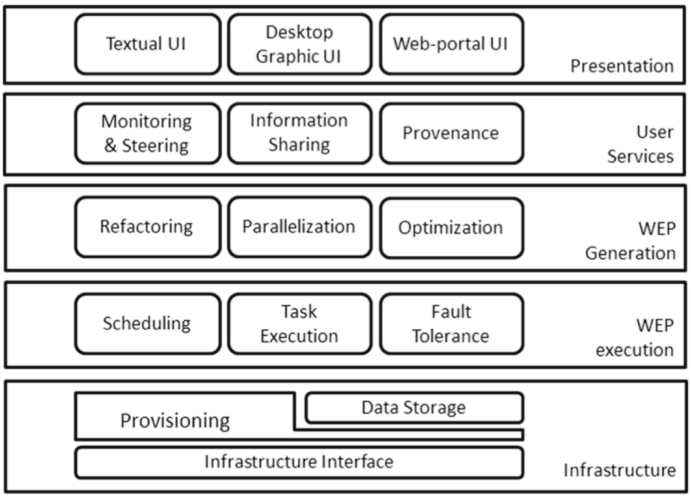


Figure 2.9: A reference architecture for workflow management engines (reproduced from [91]).

2.2.3.2. Abstract Workflow Definition

When the user submits a workflow, regardless of the approach used by the user to express the workflow, the workflow is parsed and processed into an abstract workflow definition. This definition, which follows one of the main workflow formalisms (see Section 2.2.2), ensures that regardless of the frontend used to create the workflow, all workflows end up in a consistent format. This allows all workflows to pass through the same workflow pre-processing and optimization pipeline (see Section 2.2.4.1).

An abstract workflow definition does not yet refer to any specific resources or other execution-specific configuration [141]. This allows the workflows to be executed in differing configurations, or even in differing workflow engines. Instead, the mapping the abstract workflow to resources happens in the concrete workflow definition.

2.2.3.3. Concrete Workflow Definition

Once the WMS needs to ensure that a workflow is ready for execution, it transforms the abstract workflow definition into a concrete workflow definition [141]. This requires the WMS to evaluate the resource requirements of the abstract workflow definition, and map them to concrete resources. Furthermore, it resolves and verifies the location of the data and tasks.

In contrast to the abstract model, the concrete workflow definition is workflow engine-specific, and deployment-specific. The definition should be considered immutable, akin to the way low-level machine code is considered immutable. Finally, this concrete definition is considered 'executable'; no additional information or processing is needed for the workflow to be executed.

2.2.4. Building a Workflow Management System

Workflow Management Systems (WMSs) systems can be categorized in three categories: scientific, business processes, and data processing. In the business process domain, the current state-of-the-art systems (e.g., Camunda[11], Bonitasoft[12], and jBPM[13]) focus on BPMN 2.0 [103] support. In the field of e-science, Kepler [26], FireWorks [76] and Pegasus [50] are still fundamental both to research in WMSs, and research using these scientific workflows. Finally, in the data processing (or DevOps) domain, the current WMSs (e.g., Apache Airflow, Luigi, OpenStack Mistral) target connecting big data pipelines, and orchestrating complex DevOps tasks (such as system provisioning, continuous integration, and analytics).

Although the domains of these WMSs widely vary, the components that make up these systems tend to be common across these systems. The recurrence of these components has lead to the emergence of multiple reference architectures and taxonomies of the field of workflow management systems.

[11] https://camunda.com/
[12] https://www.bonitasoft.com
[13] https://www.activiti.org/

To provide an overview of these common components, we summarize key reference architectures of WMSs [91, 141].

2.2.4.1. Workflow Preprocessing

Before being executed workflows go through various steps of preprocessing (the WEP Generation layer in Figure 2.9). Because these steps do not have a specific order or all present in every system, the following is an unordered overview of the steps that are commonly found in WMSs:

1. **Workflow optimizations**: WMSs can alter the workflow through optimizations, by removing redundant tasks and dependencies, changing task configurations, or re-configuring the workflow structure to improve the performance [91]. For example, Deelman et al. [50] propose to remove tasks of which the output is already available, and cluster tasks using the same data together, before the workflow invocation. In the Taverna WMS Cohen-Boulakia et al. [47] propose refactoring steps to detect and remove duplicate or unnecessary tasks, particularly in parallel sections of the workflow.

2. **Workflow validation**: Before accepting the submitted workflow, a WMS validates the correctness of the workflow. Although the lack of full knowledge on the resources and the behaviour of the tasks prevent a WMS to fully verify a workflow's correctness, various checks can be done on a workflow-level, such as verifying that all referenced tasks exist, that there are no cyclic dependencies in the workflow, and that there are no out-of-bound resources requests.

3. **Resource resolving**: The WMS resolve the resources—such as the data files requested, services needed, and tasks needed to be executed—requested by the workflow ahead of the execution [50]. This practice, in part a validation of the workflow, ensures that all resources requested by the workflow exist and can be reserved for the workflow execution.

2.2.4.2. Workflow Invocation

Once a workflow is invoked[14], the WMS manages the progression of the workflow invocation using a dedicated component: the controller (or engine)[138]. Based on scheduling plans of the workflow scheduler the controller is responsible for submitting the tasks to the right task executors, directing the data flow between tasks, providing mechanisms to improve the fault-tolerance of workflows executions, and acting on signals and events related to the workflow invocation.

In practice, and in the reference architecture in Figure 2.9, the workflow controller and workflow scheduler are often one and the same component, and in literature the conceptual difference is occasionally left out. Akin to the common wisdom in computer science of separating mechanism from policy, in a WMS the controller serves as the mechanism offering an API for the workflow scheduler—the policy—to use to implement specific strategies (see Section 2.3).

2.2.4.3. Workflow Scheduling

Workflow scheduling aims to answer the following question for workflow executions: *When and where should the tasks of the workflow be executed such that the functional and non-functional requirements are met as close to optimal as possible*? As part of hierarchy of schedulers (see Chapter 6), the workflow scheduler answers this question on a high-level [43]. it typically decides on which (clusters of) machines a task should be executed. Lower-level schedulers, such as cluster management scheduler, and kernel scheduler, are responsible for the fine-grained scheduling of the task on specific resources within a specific machine.

How exactly the scheduler is implementing can differ widely. Within the WMS, Yu et al. [141] identify three types of scheduling architectures: centralized, decentralized, and hierarchical. Centralized scheduling consists of a single scheduler responsible for scheduling all workflows, whereas with hierarchical the central scheduler defers the scheduling of (sub)workflows to lower-level workflow schedulers—for example, with lower-level schedulers belonging to an entity authorized to access a set of resources—and decentralized scheduling does away with the central scheduler all together.

[14] Throughout this thesis we use the terms *workflow invocation* and *workflow execution* interchangeably.

2.2.4.4. Task Execution

To execute tasks, a WMS needs a task execution runtime in which the tasks can be executed. This runtime is an abstraction of an arbitrary system—local or remote, machine or human—capable for executing a particular task [91]. This component can be managed by workflow engine itself, such as is the case with Luigi and Apache Airflow. The advantage of this approach is that the workflow engine has full control over the task execution, and reduces the complexity of the overall distributed deployment. On the other hand, the workflow engine can defer the execution of tasks to external systems—deferring in the infrastructure layer in Figure 2.9. For example, Pegasus defers the execution of tasks to DAGMan (and underneath that Condor)[50].

Similarly, workflow engines either tend to have a singular task type, or a more plugable notion of a task type. Systems, such as PyWren [79], have singular task type; they have single runtime for all tasks to execute them (e.g., a worker that can solely execute binaries). Others have a more fluid notion of a task type. For example, Luigi and Apache Airflow both have a concept of operators which describe the context in which a task should be executed; the Python operator executes the task using the Python interpreter, while the Spark operator executes the task on a Spark cluster.

2.2.4.5. Monitoring, Provenance, and Information Sharing

Represented in the user services layer in Figure 2.9, a key motivation behind handing of the execution of workflows to dedicated WMSs is that these provide extensive monitoring of workflows [141]. In the presentation layer (Figure 2.9), the metrics are typically provided to the user using a comprehensive dashboard, along with visualizations and charts [50]. This allows users to monitor the current (dynamic) state of the system and workflows, and investigate the historical workflows executions to root cause errors, degradations in performance, or the emergence of a specific result [91].

Besides the monitoring of metrics and logging, which are commonplace in all distributed systems, WMSs typically also records comprehensive provenance data [91]. Provenance data—that is, metadata related to the (intermediate) data inputs and outputs of tasks—allows the user to analyze and replay the execution of the workflow on a task-level.

Apart from providing users with an extensive overview to provide insight into the execution of workflows, the monitoring and provenance infrastructure can also serve as a source of information for other components in the WMS. The scheduler can use the data of historical workflow invocations to improve its prediction models for workflow and task performance [91]. Likewise, components responsible for autoscaling underlying resources could improve their performance by using this provenance data in their policies [68].

2.2.5. Quality of Service Metrics for Workflows

One can consider a wide range of metrics to evaluate the WMS, ranging from the usability of the user interfaces to the accurateness of the monitoring. Because these metrics in many cases overlap with metrics that are used for any distributed system we focus here on the Quality of Service (QoS) metrics that are relevant to the core responsibility of the workflow engine: the execution of workflows.

As identified by Yu et al. [141] the following metrics are evaluated on several levels of details within the WMS. At the highest level, these metrics are used to quantify the performance of the WMS overall—for this we look at the statistical properties of over all the workflow invocations. In more detail, we can use these metrics on a workflow-level to quantify the QoS. Finally, on on the most granular level, we can use these metrics to specify and evaluate the QoS of a task within a workflow; a specific task within a workflow may require high-performance or low-costs, while another may not.

1. **Performance**: The performance relates to the total time taken to execute a workflow, from start to completion. Factors that impact performance include network latency, data transfer, scheduling and controller overhead. A common metric for the performance is the *makespan*: the duration from the start of the execution of the first task to the completion of the last task of a workflow. Alternative to makespan, for WMSs with interactive workloads the *latency* (the time it takes between the user submitting a workflow and the user receiving the result [66]) can be used to evaluate the performance.

2. **Throughput**: The throughput is a measure of the number of workflow and task executions that a WMS is capable of processing in a specific time period, using a specific amount of resources.

Typically this is expressed using the ubiquitous metric of Queries-Per-Second (QPS), where the *query* represents as a workflow invocation.

3. **Costs**: The cost associated with the execution of workflows, including the cost for obtaining and managing the resources for processing the workflow and its tasks.

4. **Reliability**: The reliability is measured by the number of workflow invocations that fail to complete, relatively to the total number of workflow invocations. Within a workflow invocation, reliability can also be used as a measure of the number of tasks succeeded versus the total number of tasks completed. Typical metrics used for the reliability are the Mean Time To Recovery (MTTR), and Mean Time To Failure (MTTF) [66, 127].

5. **Security**: Security in WMSs in this work refers to ensuring that the only authorized actors can execute, control and view the results of specific workflows. Furthermore, here security also comprises the confidentiality of the execution of tasks; users should be assured that tasks run within contained and secure environments.

2.3. Control Theory in Distributed Systems

Along with the underlying Fission and Kubernetes systems, the workflow management system prototype (see Chapter 5) relies on reconciliation loops and control theory to progress workflow invocations to completion under unreliable circumstances. Accordingly, we investigate the state-of-the-art of control theory applied in distributed systems.

Control theory is an entire field of study on its own, which a small background section in this thesis cannot do justice. For this reason, we limit the scope to solely fit the purpose of this background section: control theory within the domain of distributed systems for the target audience of distributed system engineers and researchers. Therefore, in the remainder of this section we discuss control theory on three levels of detail:

1. In Section 2.3.1 we introduce the basics of control theory from the perspective of distributed system researchers and engineers.

2. We provide an overview of the pervasiveness of control theory in the distributed systems ecosystem in Section 2.3.2.

3. Section 2.3.3 provides an in-depth explanation of how controllers and reconciliation loops are defined and managed in Kubernetes.

2.3.1. A Brief Introduction to Control Theory

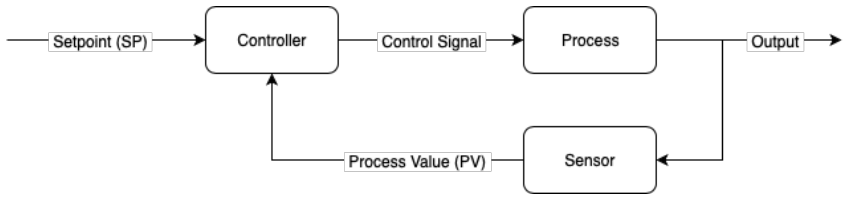
Control theory concerns itself with the modeling, analyzing, and control of dynamic systems [30, 89]; systems whose behavior changes over time, often in response to external influences. A controller observes the state of a system, compares it to the desired state of the system using a differential equation referred to as the *control law*. Based on the result of this control law the controller acts to (gradually) bring the state of the system towards the desired one, by providing inputs to the controlled system. Ideally, a controller responds to observations such that the difference between the *desired state* and the *actual state* is minimized. With a well-implemented controller, a system can be made resilient to both external influences and to variations in its internal state.

When discussing feedback loops, we are generally referring to closed feedback loop. Displayed in Figure 2.10b, a closed feedback loop depends on observations of the controlled system. The controller directly compares the observations to the desired state to the determine which actions to take. For example, a thermometer is a closed feedback loop if it bases its level of heating on the observed temperature.

In contrast, besides the closed feedback loop in Figure 2.10a, an open feedback (often referred to as feed-forward or non-feedback) loop does not base its control law on observations of the system [30]. For example, a thermometer is an open feedback loop if it bases its level of heating simply on a timer. Open feedback loops are typically less resilient than closed feedback loop, but are often favoured over closed feedback loops in non-mission-critical systems to reduce system complexity.

In control theory there are various ways to implement a control loop [30]. The simplest one is called an on/off controller, which, based on the error, makes a binary decision whether or not to perform an

(a) Open feedback control loop.



(b) Closed feedback control loop.

Figure 2.10: Open vs. closed feedback loops.

action. In contrast, a Proportional–Integral–Derivative (PID) controller is a proportional controller: it takes action that proportional to the error.

2.3.2. Controllers in Distributed Systems

As a part of the broader field of self-aware and self-adaptive computing systems [85], using control theory has become a popular approach to manage (distributed) computing resources. The use of control theory to model, analyze, and design self-aware or semi-autonomous systems to meet QoS requirements can be observed in various domains [44]: big data [54], cloud resource orchestration and autoscalers [71, 80], application performance optimization [82], and stream processing data systems [81]. To illustrate, with Borg, Omega, and Kubernetes, Google has pursued resource orchestration using different approaches, such as scheduling policies and scheduler architectures, primitives, and communication styles; yet all these systems are based on feedback control loops [42].

There are several reasons why control theory has been influential in the field of distributed systems [30].

1. **Resiliency**: In distributed systems there are various factors (unreliable networks, unreliable nodes, non-deterministic operations, etc.) which prevent distributed systems from reliably performing operations—leading to, among others, the infamous at-most-once or at-least-once conjecture [127]. In this context control loops at least provide a guarantee (assuming there are no permanent disruptions) that systems will converge to the desired state *eventually*. Because the field of distributed systems is moving towards higher and higher levels of autonomous systems, evaluated on non-functional requirements (such as availability, and reliability), higher-ordered control systems (possibly augmented with other, stochastic, machine learning, and portfolio-based approaches [71]) provide options to let systems strive for these non-functional targets (for example, in the form of a Service Level Agreement (SLA) or Service Level Objective (SLO)) without tediously implementing the rules to reach these goals explicitly.

2. **Separation of mechanism and policy**: The architecture used for control loops also leads to a clear separation of mechanism and policy, as the policy is explicitly defined as the control law. A system gains in observability and controllability by utilizing control theory. In contrast to more black-box approaches, all measurements received and actions taken by the controller can be logged. Combined with the control law this data can be used to perform extensive analysis of the behavior of the controller.

3. **Statelessness**: Feedback control loops do not require persistent state over the lifecycle of an observed system. As the control loop bases its decisions on the current state. A system with a control loop can be killed and restarted without any issue to the control loop. Restarted control loops that do depend on observation history—such as PID controllers—will only suffer in the starting phase while the control loop gathers the observations again [30].

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

Figure 2.11: An example of a definition of a controller in Kubernetes that is responsible for running 3 instances of a NGINX web server.

Although control theory has benefits when utilized in distributed systems, there are also inherent challenges to utilizing feedback control loops. First, controllers are complex components, especially if we take into account the effort needed to evaluate the behavior. Instead of blindly applying controllers to each problem, the various variations of controllers should be evaluated carefully [89]. Second, a poorly evaluated control law or lag in the inputs and outputs of the system can cause the system to oscillate; the controller takes excessive actions on small, or transient errors causing to increasingly larger errors [55]. To avoid oscillation, besides properly evaluating the system characteristics, one could use a more sophisticated (proportional) controllers, and implementing appropriate hysteresis (taking the recent history of measurements into account). Finally, the use control systems introduces more measurement noise to the overall system; the controllers take various small actions and compensating actions to eventually reaching the desired state, leading to less stable measurement patterns. To mitigate this issue, the system will need to carefully filter and process the sensor measurements (e.g., using Kalman Filters, or Fast-Fourier Transforms [41]).

2.3.3. Controllers in Kubernetes

As the last of multiple iterations of distributed resource orchestrators, the open-source project Kubernetes [15] has become the most important and influential system in the cloud computing domain [42]. In Kubernetes a controller is defined as ”a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state” [45]. The *apiserver* is an interface to a distributed key-value store, which by default is etcd[16], a RAFT-based implementation [104]. Aside from providing an interface to the cluster state, the apiserver is responsible for providing its consumers with information about resource usage, deployed artifacts, configuration, etc. Using the apiserver the controllers fetch the relevant state, and submit their desired actions again through the apiserver. With this centralized location for fetching and updating the cluster state, Kubernetes ensures a consistent view of the cluster state, and provides a clear abstraction for controllers to use.

Another advantage of this single interface to the cluster is that separate controllers can be created for each concern. This allows each controller to be relatively simple to understand, develop and upgrade. To illustrate, a subset of the default controllers in Kubernetes include:

[15] https://kubernetes.io/
[16] https://github.com/etcd-io/etcd

1. **Replication controller** for maintaining the correct number of replications of a pod (application) in the cluster.
2. **Node controller** for verifying with the cloud provider to determine if a unresponsive node has been deleted in the cloud.
3. **Service Controller** for creating, updating, and deleting load balancer configurations.
4. **Volume Controller** for creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes.
5. **Scheduler** (called `kube-scheduler`) in Kubernetes is also implemented as a controller. It watches for updates to resources, evaluates the ideal placement, and adds metadata to the resources that other controllers in turn use in their actions.

2.3.3.1. Example: Replication Controller

Next to system-related controllers, Kubernetes provides constructs for users to facilitate in building controllers, to encourage the use of the controller pattern everywhere. One of these components provided by Kubernetes is the *ReplicationController*, which ensures that a specified number of pod replicas are running at any one time [46].

For example, in the configuration file (Figure 2.11) the user defines the *desired state* of a set of containers. The user wants to have 3 replicas of a web server (in this case a plain NGINX server[17]) running at any time. The user submits this desired configuration to the apiserver, which in turn makes it discoverable by the Replication controller.

The replication controller retrieves this ReplicationController resource, when an event occurs that starts control loop. The controller compares the desired state as defined by the user (3 NGINX replicas) to the *actual state* of the cluster (0 NGINX replicas), and will deploy replicas as defined in the configuration until the desired state has been reached. Conversely, if the user later updates the ReplicationController configuration to only contain 1 replica of the web server, the controller will remove replicas until the desired number of replicas has been reached.

Other primitives, such as *Deployments*, *StatefulSets*, *Jobs*, *DaemonSets*, provide tools for users to implement different flavors control loops targeting other domains. Although these primitives serve widely varying use cases, they are all build using the same underlying primitives available to all components. Therefore, if the existing primitives do not support a certain use case, the user can choose to implement their own controller operating on custom resource types using the same tools as the existing controllers.

[17] `https://www.nginx.com`

3

Requirements Analysis for Function Composition in FaaS

Within the rapidly growing field of serverless computing, reliable composition or orchestration of cloud functions has been identified as one of the main challenges that hinders adoption of the paradigm [32, 133]. Yet, the community lacks a clear analysis of what a *useful* approach to function composition is. What are the use cases, workloads, characteristics, and constraints of serverless function composition?

Although previous and concurrent work has identified the challenge, provided initial analysis, and proposed approaches to function composition, these lack methodical analysis and convincing conclusions on the appropriateness of different approaches. Concurrent work by Baldini et al. [33] framed the function composition challenge into a trilemma: functions should be considered as black boxes; function composition should obey the substitution principle with respect to synchronous invocation; and invocations should not be double-billed. Despite a lack of thorough analysis of the domain, the authors conclude that workflows are a promising direction to investigate further for composition in FaaS.

Due to the lack of an existing, comprehensive analysis, in this section we provide a broad, systematic requirements analysis regarding the composition of functions in FaaS. To understand the characteristics of cloud function compositions, this section describes the current and prospective use cases (Section 3.1), common workloads (Section 3.3), and stakeholders (Section 3.2). From this usage domain—combined with the fundamental principles of FaaS platforms—the requirements (Section 3.4) and constraints (Section 3.5) for a function composition approach are formalized.

3.1. Composition-Enhanced and -Enabled Use Cases

For the problem of function composition it is important to understand what the (prospective) use cases are. The generic nature of FaaS and the subsequent composition leads to a various use cases, which we divide into two broad categories: *composition-enhanced use cases*, and *composition-enabled use cases*.

3.1.1. Composition-Enhanced Use Cases

The composition of functions can happen at arbitrary granularity, from implementing low-level fallback logic, to high-level workflows of business processes. For this reason, we posit that any current use case of FaaS is a use case for complex function compositions. Using function compositions for existing use cases can enhance performance and non-performance related aspects of the FaaS model.

FaaS and serverless computing in general are emerging technologies lacking a common consensus on definitions, interfaces, and—indeed—use cases. To get a clear picture (especially in light of our definitions in prior work [132]) of the use cases, we surveyed a variety of sources: (1) case studies and documentation of major FaaS providers [9, 11, 14, 19]; (2) existing surveys [32, 134]; and, (3) prominent industry-originating articles and white-papers [113, 139]. The listed use cases were deduplicated and grouped together based on workload characteristics. To validate each of these high-level use case, we searched for and mapped publicly reported, real-world serverless deployments to these use cases.

The current use cases of FaaS and serverless computing in general are: batch processing, stream processing, edge computing back-ends, and web applications:

3.1.1.1. Stream Processing

The initial use case of FaaS touted by AWS at the introduction of AWS Lambda [7] (arguably the oldest production-level FaaS platform [134]) was to use it as the *glue*: services emitted events, triggering a FaaS function, which in turn does simple processing and calling of external services. This kind of stream processing, or event processing, has remained one of the canonical examples of the domain where serverless has most impact, because of the convenience of creating fully-managed event processors that are billed based on the number of events. This use case can be found in most organizations: tying different cloud services together. Another emerging set of examples of this use case involve the processing of analytics [97].

3.1.1.2. Batch Processing

Compared to stream processing, batch processing involves processing less frequent, but larger chunks of data that need to be processed. Although there are clear challenges remaining to make FaaS effective for big data processing [132], batch processing on a smaller (data) scale is is being adopted as one of the use cases for FaaS and serverless computing. Examples of this asynchronous—often interval-based—use case are: the US' Financial Industry Regulatory Authority (FINRA) generating daily reports [13], daily re-balancing of stock portfolios at Financial Engines [12], or creating backups at Netflix [18].

3.1.1.3. Web Applications

Compared to the types of background processing, user-facing, interactive web applications have strict requirements with respect to latency and availability. Despite the cold starts (see Section 2.1), for most operations the performance FaaS is in line with existing technologies [139]. Examples of interactive web applications using FaaS: chat-bots [87], internet retailing [100], and read-heavy websites [67, 106]. Similarly, in the academia, initial serverless research by Hendrickson et al. [64] validated their approach using web application workloads.

3.1.1.4. Edge Computing Backend

Both mobile and Internet of Things (IoT) rely on cloud-based back-ends to perform requested services. A variation on the web applications use case, the emerging edge computing paradigm emphasis the need for computation locality. In mobile computing, users expect highly interactive *apps* (such as websites, messaging, and search) despite the high network variance and congestion. In IoT, sensors generate huge amounts of data, which need to be processed notwithstanding the (high) costs of data transfer. The stateless, easy-to-migrate, and cheap-to-deploy properties of FaaS makes it suitable technology to use in these fields. For example, the smart vacuum robots of iRobot use a serverless architecture to process analytics and manage the robots remotely [31], and Coca Cola is using FaaS to process vending machine payments [51]. Research by Akkus et al. [24] and Nastic et al.[97], reflect the interest within the academia for the edge computing use case.

3.1.2. Composition-Enabled Use Cases

Besides enhancing the expressiveness and usability of FaaS for its existing use cases, function compositions also enable new types of use cases for FaaS. These use cases all involve orchestrating complex processes or workflows over a longer period of time.

3.1.2.1. Long-Running Business Processes

Within organizations and systems, computational and business processes are complex, and constantly evolving. Running and monitoring these complex processes have given rise to an industry of business process systems, most notably leading to the emergence of the BPMN workflow standard (see background Section 2.2) Although, we could technically model most of these processes into self-contained FaaS functions, there are two problems with this approach. First, the functions are too constrained: they lack state management, and have a small maximum runtime. Second, these complex processes tend to have a lot in common, making writing a function dedicated to each process ineffective; we should be able to reuse smaller processes in larger processes. The Guardian employs serverless

workflows—with AWS Step Functions—to orchestrate complex business processes that involve various internal and external systems to create and delete subscriptions and implement fall-backs in case billing services are unavailable [16].

3.1.2.2. Continuous Integration and Deployment (CI and CD)

Although CI/CD can be viewed as a category of business processes, these processes have distinct characteristics: CI/CD workflows are often less logically complex (hence we refer to them as pipelines, not processes), and the tasks within these pipelines are far more resource-intensive (compiling artifacts; running performance tests). Fouladi et al. [56] speed up software builds by extracting compilation steps from Makefiles to a workflow of FaaS functions. Autodesk bootstraps development environments for its engineering teams using a complex orchestration of serverless components [8]. Popular, and arguably 'serverless', build tools, Jenkins[1] and Travis CI[2], contain functionality to separate the build and test process into different *stages* or *steps*—allowing the user to create basic, static workflows.

3.2. Stakeholders

Because the use cases for function composition comprise at least the use cases for FaaS, the stakeholders in FaaS function composition do not seem to differ from those for regular FaaS use cases. Based on the descriptions of the aforementioned use cases, FaaS composition comprises of at least the following six stakeholders:

1. **FaaS user**: The FaaS user is the actor using the functionality of the FaaS platform to solve business problems, creating applications using FaaS functions. With function composition available, it becomes part of the toolkit that the FaaS user can utilize to solve their use cases.

2. **End-user**: The end-user represents the users that rely on the (serverless) applications build by the *FaaS user* and operated by the *FaaS Operator*. In other words, the end-user is the stakeholder with the main stake in the successful execution of (their part of) the workload of the FaaS (composition) platform.

3. **FaaS operator**: Similar to a cloud operator, the FaaS operator is responsible for ensuring that the FaaS platform is available for FaaS users to develop their functions in. The operator is also responsible for ensuring that the deployed instances of FaaS functions and function compositions operate correctly and scale according to the demand. Although the FaaS operator is typically a separate, external party (e.g., a cloud provider), it can also be an internal party (e.g., a DevOps team within the organization).

4. **FaaS application QA**: Although most of the functional and operational monitoring is performed by the FaaS operator, Quality Assurance (QA) is still needed from the side of the *FaaS user*. This stakeholder is typically responsible for ensuring that the correct functions and compositions are deployed, using the correct procedures and security practices. If necessary, it is tasked with verifying that the FaaS operator meets the FaaS platform SLAs. Finally, the actor monitor the operation of the FaaS functions from a higher (business) point of view: are the serverless functions (and compositions) performing the correct operations?

5. **FaaS platform developer**: The FaaS platforms themselves—whether open-source or developed in-house—are designed and implemented by the FaaS platform developer. With the addition of function composition this stakeholder either needs to have a thorough understanding of distributed function composition to build the function composition system, or they need to be able to interface with an existing function composition system to integrate it seamlessly with the FaaS platform.

6. **Function repository operator**: Similar to the sharing of libraries within programming language communities, once functions or function compositions are written, it is often desirable to share these functions among different users. The function repository operator is responsible for facilitating the sharing of existing FaaS functions and function compositions. This repository of shared functions can be in-house, or it can be shared in larger, open-source communities (for example, AWS serverless application repository [1]).

[1] `https://jenkins.io`
[2] `https://travis-ci.org`

3.3. Workloads

Despite the emergence of well-documented use cases of FaaS and serverless computing in general, there is a lack of workload traces or characterizations for these serverless use cases. Due to this lack of publicly available workload details, we resort to multiple approaches to determine the workloads for function compositions: we synthesis general workload characteristics from the use cases and case studies listed in the previous section, which in some cases reveal characteristics in textual descriptions of their workloads (for example, FaaS functions at Financial Engines run 'daily' to re-balance portfolios [12]); some use cases of serverless computing directly overlap with an existing domain (for example, for web applications there are well-known workloads describing arrival patterns); and, in other cases we can draw parallels to existing technologies that focus on similar use cases (such as is the case with BPMN and business processes). Based on this initial analysis we find that the workloads for FaaS function compositions generally consist of: reasonably complex, dynamic workflows, containing lightweight tasks, which are executed frequently with bursty arrival patterns.

Frequent executions. With use cases, such as stream processing, web applications, and edge computing back-ends, the workload consists of many small events occurring and triggering (composed) function executions [28].

Bursty executions. Related to the frequent executions, use cases, such as web applications, are notorious for their bursty workloads. Next periodic variations (e.g., day-night cycles), viral content, and irregular events can trigger huge spikes in the workload [28]. Function composition for these use cases should anticipate similar irregular patterns, which leads to requirements of rapidly scalable executions of the function compositions.

Small tasks with long-tailed provisioning times. Regardless of the complexity of the function composition, fundamentally it exists out of the constrained FaaS functions (or, in workflow terminology, tasks). The benefit of this workload characteristic is that for system responsible for managing the composed functions, the individual choices where and how to schedule the tasks are less important—choices which for other types of workloads involving long-running tasks are far more impactful [118]. However, the downside is that the overhead of the function composition approach and the provisioning of the function has more impact at scale [48]. Moreover, in practice, these function provisioning times (see *cold starts* in Section 2.1) are present and can range from tens of milliseconds to multiple seconds [137].

Small-sized input and output data. From the use cases and case studies we find that it is common practice to keep inputs and outputs of the function (compositions) minimal. When handling large amounts of data, the function itself retrieves the data based on a provided identifier. This practice is also caused by the state-of-the-art FaaS platforms, who provide strict limits on the input and output sizes (for example, AWS Lambda has a request and response size limit of 6MB for synchronous and 126KB for asynchronous executions[3]). Because function compositions are constrained by the limits imposed on the underlying functions, we can infer that the input and output data of a function composition will also be in the range of MBs, rather than GBs or TBs. Research towards data-intensive serverless [116] so-far is upholding this workload characteristic, favoring more frequent function executions on smaller sets of data.

Frequently updated components. With many independently managed and deployed functions making up the composed functions, the function compositions themselves will update frequently. If we assume that the deployment interval of a FaaS function is roughly similar to that of a microservice—a week [101]—than a even function composition larger than a handful of functions will be updated daily. With compositions building upon other function compositions, we can expect the software-side of the workload for function composition to be high.

[3] `https://docs.aws.amazon.com/lambda/latest/dg/limits.html`

Complex compositions. The wide variety of domains, including complex business processes, the workloads of function compositions will be—in terms of complexity—between the simplicity of static workflows (no complex control logic) and the complexity of BPMN (many complex control flow constructs). Existing function composition systems, such as AWS Step Functions and Azure Logic Apps, provide conditional branches, error handling, and (dynamic) loop support. Given that those constructs are concepts available in nearly all general-purpose programming languages, we expect that the workloads will contain control flow constructs modeled after similar patterns in those languages.

3.4. User-Level Requirements

Based on the use cases (Section 3.1) and workload characteristics (Section 3.3), we can extract the requirements that a function composition approach should fulfill from a user perspective:

R1 Support complex control flow constructs. The approach should support common control flow constructs available in most general-purpose languages: (1) conditional branching, (2) recursion, (3) (dynamic) loops, and (4) error handling.

R2 Support long-running processes. To support complex business processes, the approach should support compositions waiting during an execution for specific events to happen before continuing.

R3 Minimize the composition overhead. The approach should minimize the overhead on the performance on the composition, while the composed function should at the same time not cost more than the sum of the costs of the individual functions.

R4 Reliable executions. To enable the execution of (critical) business processes, it is vital for the approach to execute compositions reliably. A user should be able to expect a function composition to be nearly as 'atomic' in execution, as a primitive function. Therefore, the function composition system should be resilient to at least common faults—such as intermittent network faults and restarts of the system during the execution of composed functions.

R5 Support versioning and upgrade strategies. With the frequent deployments of granular functions, the function composition approach should allow users to version their compositions and reference versions of the underlying functions. This functionality should should provide support for extensible upgrading strategies, ranging from compositions upgrading automatically on every new function version to business-critical compositions pinned to specific versions of functions.

R6 Elastic and scalable executions. Due to the burstiness of the workloads, the function composition approach should be able to scale with the functions. The scaling process should happen at least as fast as the scaling of the functions instances; the function composition approach should be elastic (using the definitions of the SPEC CLOUD WG [65]).

R7 Scalable in number of compositions. The granular nature of FaaS functions leads to users creating many function compositions. Both the system and the user should be capable of dealing with a large number of function compositions: the system should be able to scale with the number of compositions, and the user should be able to create these function compositions with minimal added overhead.

3.5. System-Level Constraints

Next to the requirements, the potential approaches to function composition should satisfy constraints posed by the current systems and ecosystems. Within this context, we view constraints as real-world, technical limits—which are outside of our limits or authority to change. For an approach to function composition, the following constraints are considered in this work:

C1 Use HTTP as the interface to functions. FaaS platforms support a variety of different *triggers* or *event sources* to start the execution of a function, from integrations with various message queue protocols to cloud platform-specific events. However, a generic approach to function composition should avoid using event sources only available in a specific FaaS platform.

For this reason, any approach should solely rely on HTTP as the interface to FaaS functions. Based on a review of the largest open-source and managed platforms [135], of all event sources,

HTTP is the only event source that is supported by all platforms. Despite a lack of a universally accepted interface in HTTP, there are several initiatives working on this, such as the CNCF Serverless WG [139], FaaSlang [69], and AWS with its SAM model [117]. Moreover, even though not strictly consistent across platforms the HTTP interface in FaaS platforms is generally similar, requiring only minimal effort to add new integrations.

C2 Treat functions as black-boxes. In the FaaS model (Section 2.1) functions are generic, function deployments are immutable, and function executions are atomic. Once a function starts executing, an external user or system cannot influence, modify or control the execution anymore; the FaaS model requires users to treat functions as black-boxes.

As argued by Baldini et al. [33], for a function composition approach the constraint should not be different. The approach should not *require* any additional functionality to control or influence active functions or function executions.

C3 Only expect functionality provided by all FaaS platforms. The wide diversity of FaaS platforms also leads to a wide diversity in the functionality offered by these platforms. For example, Google Cloud Functions [14] encourages users to use the *tmp* directory as a filesystem cache across function executions, while other platforms do not support this. FaaS providers have differing limits on the runtime of functions; Google Cloud Functions at the moment of writing allows 9 minutes of runtime, while Azure Functions and AWS Lambda allow for 15 minutes of runtime. Similarly, the performance and other non-performance aspects of FaaS platforms widely differ [137].

The function composition approach should be generic enough to be implementable in FaaS platforms; it should avoid using any functionality that is not present on all FaaS platforms. This also leaves open the opportunity to create cross-cloud and cross-platform workflows. However, as we point out in our prototype design (Chapter 5), the function composition approach can still utilize platform-specific functionality to improve the compositions—as long as that functionality is not required to be present.

C4 Adhere to the serverless development and operational workflow. Serverless computing and FaaS in particular propose a strict and opinionated development model. Any approach to generic function composition should fit well into this paradigm; it should minimize the extensions or changes to the development model. To foster composability, function compositions should adhere to the same interface as primitive functions—users should not have to be aware whether the function they are executing or reusing in a composition is a regular or a composed function.

3.6. Limitations

Despite that this requirements analysis of function composition within serverless computing is one of the first, and thorough in the field, limitations remain to this analysis.

The requirements analysis is based solely on publicly available sources, which could potentially ignore developments being made within the many proprietary serverless and FaaS platforms and providers. A recent example of this is the development of the Knative serverless platform [105] which has only been recently open-sourced after years of development internally at Google. The lack of insight into these proprietary ecosystems could result in workload characteristics, constraints, and use cases that are not discussed in public sources to be missing in this analysis. However, it is unlikely that major use cases are missing, because it is in the interest of the providers to promote publicize their use cases to attract users with similar use cases.

Related to the lack of insight in the closed-source serverless platforms, we lack quantitative data on the workloads of. The workload used in the previous work is either speculative (Hendrickson et al. assume a serverless web application workload similar to the Gmail web application [64] for OpenLambda, and Jiang et al. [78] and Spillner et al. [123] assume existing scientific workloads for their FaaS prototypes), purely synthetic [24, 79, 122], or constrained to a highly-specific use case (Fouladi et al. focus on makefile-[56] and video encoding-workloads [57]). The large, production-ready FaaS providers (AWS, Microsoft Azure, Google Cloud) which do have extensive knowledge about the workloads and use cases have (at the time of writing) yet to publish workload traces or characteristics. Given the privacy and competitiveness concerns it is unlikely that they will publish detailed traces or characteristics in the near-future.

Finally, the serverless ecosystem is rapidly evolving, with advances in both research and engineering[4]. For example, initial assumptions that FaaS would not be performant or cost-effective enough for web application workloads have been largely overturned–albeit with some workarounds [48]. Similarly, the large-scale and data-centric scientific workloads [24, 57] have only recently attracted the attention of researchers to use serverless computing. There are use cases and related workload characteristics that seem unlikely use cases of serverless computing: High-Performance Computing (HPC), with its hardware-tied optimizations and performance requirements; big data processing, with its emphasis on data-centric over compute-centric processing; and, mission-critical workloads, with its strict performance variability requirements. But with the ecosystems and models of serverless computing evolving rapidly, it can happen that these use cases will become realistic use cases of serverless computing. For this reason it is better to coin these use cases as *unlikely* rather than *impossible* use cases.

[4]In fact, we have had to rewrite parts of this thesis multiple times over the course of the year to include the latest advances.

4

Survey and Analysis of the State-of-the-Art

Following the requirements analysis (Chapter 3), in this section we aim to answer the question: *What is the current and related state-of-the-art in serverless function composition?* To understand why the state-of-the-art lacks proper function composition, we perform a systematic survey of the state-of-the-art in FaaS and its function composition approaches. This survey is qualitative; we consider the quantitative evaluation future work, delegated to the SPEC RG CLOUD [133].

Related work has not yet systematically surveyed and analyzed both the conceptual along with the technical components of serverless function composition. Baldini et al. [33] addressed the conceptual problem of function composition as a *serverless trilemma*; an approach (1) should not require introspection into serverless functions, (2) should not cause additional costs (double spending), and (3) should allow the substitution of functions by function compositions. However, the proposed approach—a workflow-like approach without a centralized workflow definition, scheduler and controller—lacks performance considerations, relies on the specific implementation of the OpenWhisk FaaS platform, and only acknowledges the potential of workflow scheduling in its related work. Other industry efforts, such as the CNCF serverless working group's whitepaper [139], survey the field of FaaS platforms thoroughly, but refrain from surveying the state-of-the-art in function composition (instead they pose the model of AWS Step Functions as the sole approach to function composition). Jiang et al. [78], Fouladi et al.[56], and Jonas et al. [79] pose the function composition issue from the academic, (scientific) workflows perspective. Their surveys do not include the technical state-of-the-art and do not consider generic function composition beyond their specific use case; their proposed function composition systems *leverage* the existing FaaS platforms to in their respective domains, rather than aiming to improve state of serverless computing.

In the remainder of this chapter we survey and analyze the state-of-the-art in four areas related to serverless function composition (see Table 4.1). Though differing in some aspects, most criteria that we use for the surveys are derived from the requirements analysis (Chapter 3).

1. For the FaaS layer supporting the function composition prototyping, in Section 4.1, we survey the existing *state-of-the-art FaaS platforms* to decide which platform to use for an initial prototype.

2. In Section 4.2 we survey and analyze the potential *approaches to serverless function composition*, from which we conclude that workflows is the prime candidate.

3. Following the conclusion that the prototype utilizes workflows, we analyze existing *Workflow Management Systems (WMSs)* which we can use as a basis for the workflow engine prototype (Section 4.3).

4. Section 4.4 describes our analysis of existing *workflow languages* within the context of serverless workflows.

Category	Topic	Description	Survey	Implication
Conceptual	Function composition approaches	4.2	4.2.6	workflow language (4.5), design (5)
	Workflow formalisms	2.2.2	[136]	workflow language (4.5)
	Serverless computing use cases	3.1	3.1[1]	requirements (3.4), constraints (3.5)
Systems	Workflow languages	2.2.4	4.4	workflow language (4.5)
	Workflow management systems	2.2.4	4.3	design (5)
	FaaS platforms	2.1.4	4.1	implementation (7)

Table 4.1: An overview of the surveys in this thesis and our prior work.

5. Based on the preceding surveys and the requirements analysis, in Section 4.5 we propose a new type of workflows: *serverless workflows*. We describe the characteristics of this type of workflow, and how it differs from existing workflow types.

This chapter relies on several prior publications. A prior evaluation of workflow formalisms [136] motivated the decision to survey solely DAG-based workflow languages (Section 4.4), and WMSs (Section 4.3). The survey on FaaS platforms (Section 4.1) influenced and was influenced by the work with SPEC RG CLOUD on a FaaS reference architecture [135].

4.1. Survey of FaaS Platforms

In this section we survey the state-of-the-art FaaS platform to find the most suitable platform to use as a basis for the prototype and experimental evaluation. The composition of FaaS functions builds on the function primitives, and model of the underlying FaaS platform (see Section 2.1), which made the decision which platform to use one of the key decisions in the process.

4.1.1. Requirements

The requirements for the FaaS platform we base for the most part on the requirements (Section 3.4) and constraints (Section 3.5). Overall, the FaaS platform that we will use to evaluate function composition approaches on should be representative, extensible, and be performance-focused. We consider the following criteria for the FaaS platform:

F1: Open-source Software and culture Although not part of the requirements or constraints as listed in Chapter 3, using an open-source and community is important both ethically and to augment the platform if needed.

F2: Thorough documentation Is the platform easy to understand? Is (extensive) documentation available on the functionality and internals?

F3: Well-defined, and extensible architecture Does the architecture of the FaaS platform promote extension through a stable, well-documented interface? Does it have align extensible, unopinionated architecture or does it enforce the use of complex stacks of technology for any extensions?

F4: Focus on Performance Does the platform focus on performance? Does the architecture and design decisions of platform inhibit or promote performance optimizations?

F5: Active community and wide-spread adoption Does the platform have an active community to collaborate with, interact with, and ensure long-term support?

F6: Representative Is the FaaS platform representative for other platforms? Does it follow the serverless model, and meet the constraints (Section 3.5)?

4.1.2. Methodology

This comparison is purely qualitative, because (1) no comprehensive benchmark for the performance currently exists, (2) the scope and time associated with this research is limited and best spend elsewhere, and (3) performance in all platforms will likely evolve in the coming years once the field shifts it

[1] An extended, more comprehensive survey is part of concurrent work with the SPEC RG CLOUD [135].

FaaS Platform	F1	F2	F3	F4	F5	F6
AWS Lambda	○	●	○	○	●	●
Azure Functions	○	●	○	○	◐	●
Google Cloud Functions	○	●	○	○	◐	●
OpenWhisk	◐	●	○	●	◐	●
OpenFaaS	●	◐	○	○	◐	◐
OpenLambda	●	○	●	○	○	◐
Kubeless	●	◐	◐	○	◐	●
Fission	●	◐	●	●	◐	●
Funktion	●	○	○	○	○	●

Table 4.2: A qualitative evaluation of main FaaS platforms (as of April 2017[5]) given the requirements of the FaaS platform underlying the workflow engine prototype. The symbols used in the table encode how well each FaaS platform satisfies a requirement: ●: Fully satisfied; ○: Not satisfied; ◐: Partially satisfied.

focus from getting the core functionality implemented to more of a focus on non-functional characteristics (including performance). For this reason requirement F4 (Focus on performance) will be evaluated by informal tests and on our judgement of the architecture and design decisions.

At the moment of evaluation (May 2017) we selected all main FaaS platforms to be considered as the platform underlying the prototype[2]. The main production-grade FaaS platforms: AWS Lambda, Azure Functions, and Google Cloud Function. Open-source FaaS platforms managing their own resource orchestration: IronFunctions, OpenWhisk, OpenFaaS, and OpenLambda. Open-source platforms leveraging Kubernetes for resource orchestration: Kubeless, Fission, Funktion.

4.1.3. Results

The results of the qualitative evaluation are visualized in Figure 4.2. Although their use is widespread, the hosted platforms (AWS Lambda, Azure Functions, Google Cloud Functions), are discounted because they are closed source—which inhibits gaining clear understanding of FaaS internals and performance. Several of the open-source projects lacked any activity or community adoption, which included OpenLambda[3] and Funktion[4]. OpenFaaS (at that time confusingly called 'faas') was too immature—lacking a well-defined, stable architecture, notion of performance, and documentation—to use as the basis of the prototype.

This left three options for the FaaS platform to consider: OpenWhisk, Kubeless, and Fission. All of these options are open-source. However, in terms of culture, in the case of OpenWhisk the governance and development seemed to be happening mostly within IBM rather than out in the open. Architecture-wise Kubeless and OpenWhisk both relied for a large part of their architecture on a message bus—Kafka—to handle the communication between the components; Fission relied on direct HTTP-based communication. Fission has a clearly stated focus on performance; Openwhisk had similar performance optimizations; Kubeless lacked any notion of performance. Finally, while Platform9's Fission and Bitnami's Kubeless are relatively similar in terms of development and community adoption, IBM's OpenWhisk has arguably the most development and community adoption. However, due to the comparably large marketing efforts by IBM it is unclear how much of this interest in OpenWhisk is generated by IBM itself, versus genuine community interest.

4.1.4. Conclusion

Based on this evaluation we found that the Fission FaaS platform fit the requirements best[6]. It is open-source with all development and design discussions happening out in the open—with the Fission team actively attempting to include the community. It has a well-defined, stable architecture with a clear focus

[2] https://docs.google.com/spreadsheets/d/1_UfFXnB_LeLe-rRqK__GKp8WdSUz3CDTNe1yz8eK3zA/edit

[3] Although arguably the earliest open-source (academic) FaaS platform it has since seen development only sporadically—being largely irrelevant outside of academia

[4] Officially abandoned by Red Hat in the summer of 2017 in favor of OpenWhisk

[5] This quickly evolving field has since seen the introduction of various new platforms (e.g., Google's Knative, Oracle's FN, VMWare's Riff), and major advances in existing FaaS platforms.

[6] Based on this conclusion, we contacted Platform9, which in turn led to a fruitful collaboration for the remainder of the thesis.

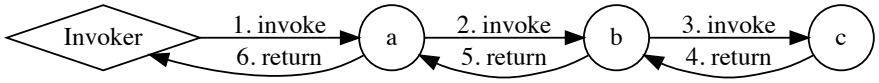


Figure 4.1: An example of function composition, consisting out of a chain of functions: a, b, and c. The numbers in the edge labels indicate the order of execution.

on performance, employing various optimizations to lower the function runtimes (such as a threadpool-like optimization, reusing function containers, and various caches). Finally, it has an active community of both users and developers.

4.1.5. Limitations

In this section we address the limitations of this qualitative survey of FaaS platforms. Specifically we identify three limitations: the timeliness of the survey, the qualitative nature of the criteria, and the subjective nature of the evaluation.

First, a limitation to this survey is its timeliness. The survey was performed at the start of this work, in the summer of 2017. With the popular serverless field evolving rapidly, this survey may fail to represent the newest (versions of) platforms. However, based on the motivation for this survey—finding a representative FaaS platform experiment on—we do not believe that there has been any shift that has made the selected platform (Fission) less representative.

Second, this survey does not include a quantitative component. A quantitative evaluation of the platforms would have been useful to identify which platforms are representative performance-wise. However, based on the motivation of this survey and the limited impact of the FaaS platform on this work, we believe that the minimal impact of a quantitative component would not have justified the effort.

Finally, a subset of the criteria are relatively subjective, such as the quality of the documentation and the extensibility of the architecture. We believe that this in part justified because of the scope of this survey, and due to its purpose being solely for the selection of the FaaS platform to use for the experimentation.

4.2. Survey of Serverless Function Composition Approaches

Orchestrating complex operations in a distributed environment is a non-trivial problem [127]. Various approaches have been proposed and are used in practice: message queues, peer-to-peer networks, and workflow management systems. In this context, the orchestration of FaaS functions can be regarded as a sub-problem, with the specific use cases, workloads, and characteristics described in Chapter 3. The goal of this survey is therefore to determine which approach to function composition—or function orchestration—fits the FaaS model comparatively best.

Within the context of this survey, we first collected all relevant approaches to function composition from current (serverless) use cases and approaches proposed in related fields (see Section 3.1). This resulted in the following five approaches to (serverless) function composition:

1. **Direct**: direct function composition consists of functions invoking other functions directly to create complex orchestrations (Section 4.2.1).

2. **Compiled**: compiled function composition relies on linking and compiling the source code of functions together into the composed function (Section 4.2.2).

3. **Coordinator**: coordinator-based function composition defers a large large or all orchestration logic to a user-defined orchestrator (Section 4.2.3).

4. **Event-Driven**: event-driven function composition consists of a event listeners triggering the execution of functions, which in turn produces events that can trigger other functions (Section 4.2.4).

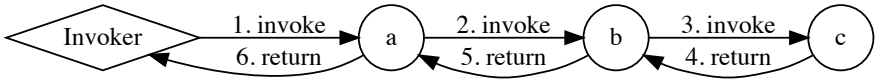


Figure 4.2: A direct composition of the chain of functions: a, b, and c. The numbers in the edge labels indicate the order of execution.

5. **Workflows**: workflow-based function composition represents functions are represented as interdependent tasks which a workflow engine executes in a dependency-satisfying order (Section 4.2.5).

In Section 4.2.6 we evaluate of these five function composition approaches. From this evaluation we conclude that the a workflow-approach is most promising to pursue as the basis of the function composition approach in this work.

To illustrate the similarities and differences between the different approaches, we map an example (Figure 4.1) to each. This example employs a directed graph to represent a chain of three functions that need to be executed in a specific order. The edges represent the control flow. This indicates the required order of execution, or the inverse dependency relation; function a needs to be executed before function b, which in turn needs to be executed before function c.

4.2.1. Direct Composition

In direct (or reflection-based [33]) composition function directly invoke the functions that it depends on. In this type of composition the functions form a dependency graph without any central component orchestrating the execution. As illustrated in Figure 4.2, the invoker executes the composition by invoking function c. Internally function c will invoke other functions, which in turn invoke other functions. This forms an implicit dependency graph. Each function will wait until its dependencies have completed, before completing themselves.

As this form of composition does not require any specialized component or functionality other than the ability for functions to invoke other functions, which is supported on most FaaS platforms. It is similar to how operations span multiple services in Service-Oriented Architecture (SOA) and microservice architectures—delegating communication and dependency logic to the services themselves. With the notion ”smart endpoints, dumb pipes”, the microservice paradigm promotes centralizing the orchestration complexity in the services (the ”endpoints”), leaving the middleware (the ”pipes”) as simple as possible [99].

4.2.2. Compiled Composition

Compiled composition is similar to how function composition is supported in programming languages. Given a description of the functions and their control flow relation, a new function is created that internally is composed of the three functions. To both the user and the FaaS platform this composed function is similar to any other FaaS function. In Figure 4.3 an example of this approach to composition is shown. The functions a, b, and c are concatenated in a new function z. When the invoker executes this function z, internally and atomically, it will execute the composition of functions; the independent functions a, b, and c themselves are not invoked.

This approach of compiling FaaS functions into new more complex FaaS functions has not seen any considerate industry adoption, because it breaks the black-box constraint (3.5[C2]) that has been a key part of the serverless computing paradigm so-far. However, it has been proposed and discussed conceptually in a few research and industry conferences [33].

4.2.3. Coordinator-Driven Composition

In coordinator-driven composition a function, called the *coordinator*, manages the correct execution of the dependency graph. This function is provided (in-part) by the user. It is responsible for providing the fault-tolerant, consistent, and correctly ordered execution of the functions. In the example (Figure 4.4)

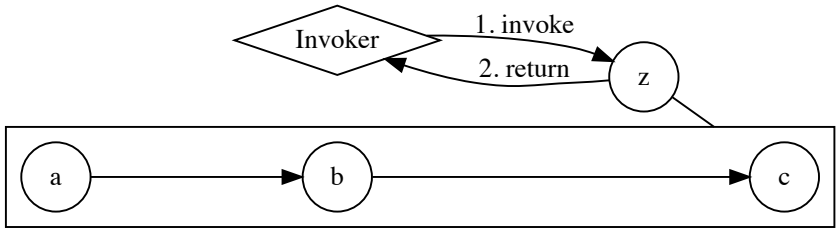


Figure 4.3: Compiled composition of the chain of functions: a, b, and c. Here function z internally contains the functions a, b, and c. The numbers in the edge labels indicate the order of execution.

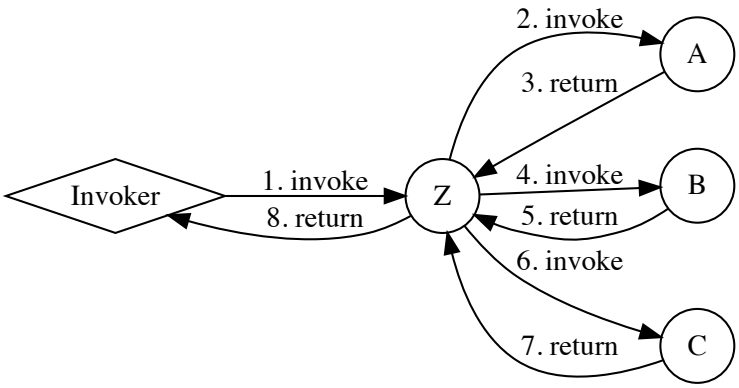


Figure 4.4: Coordinator-driven composition of the chain of functions: a, b, and c. Here coordinator function z orchestrates the chained execution of the functions a, b, and c. The numbers in the edge labels indicate the order of execution.

a coordinator function z is created containing an implicit dependency graph of the functions a, b, and c. The invoker executes the coordinator function z. The coordinator in turn invokes the functions a, b, and c according to its internal dependency graph.

The coordinator-driven approach is often proposed as an initial or ad-hoc solution to function composition, as illustrated by blog posts on introducing it to AWS Lambda[7] [8]. Although not strictly targeting FaaS, AWS Simple Workflow Service (SWF) requires users to upload a coordinator function, which responsible for keeping track and advancing through the FaaS functions.

4.2.4. Event-Driven Composition

Event-driven compositions make use of message queues, such as Kafka or RabbitMQ, to orchestrate complex executions. The message queue serves the role as the orchestrator by providing fault-tolerant, scalable and ordered message delivery. By listening and publishing messages—events—to specific topics, functions can be executed in the right order. The functions themselves remain decoupled from each other by solely using the message queue for the control-flow (and data-flow) communication. In Figure 4.5 the example of Figure 4.1 is mapped to the event-driven approach. The invoker, whether it is an actual user or middleware component, submits the function execution as an *invoke(a)* mes-

[7] https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/

[8] https://read.acloud.guru/some-lessons-learned-about-lambda-orchestration-1a8b72a33fd2

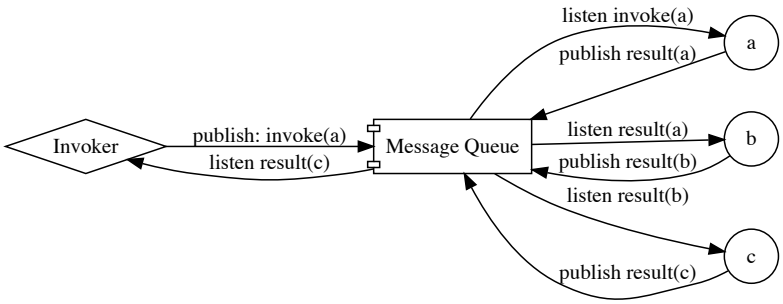


Figure 4.5: Event-driven composition of the chain of functions: a, b, and c. The functions publish and listen for specific messages (e.g., *invoke* and *result*) on a generic message queue. The execution starts with the invoker publishing the invoke message; the other functions are triggered when receiving the events.

sage to the message queue. Function a will be executed upon receiving this message, and emit its result as another message. This resulting message will in turn trigger the execution of other functions. Meanwhile, the invoker will wait for the message containing the result of the last function c.

The example in Figure 4.1 depicts a control-oriented event-driven approach, in which the messages resemble control flow constructs to explicitly invoke a specific function. Another variant on this type of approach is a data-oriented approach, where functions await specific data which they require in their execution. Functions operate on the data, emitting the resulting data back to the message queue. This variant, at the cost of potentially redundant computations, increases the decoupling even further.

Using an event-driven approach is gaining attraction, with the iRobots [31] and Autodesk [8] being notable examples of this pattern. There are several systems supporting this model, such as Microsoft's Azure Event Grid[9] and Serverless Event Gateway[10]. Alternatively, serverless deployments use a generic message queue, such as Kafka, RabbitMQ, or ZeroMQ, as an integral component of the infrastructure to support event-driven function composition.

4.2.5. Workflow-based Composition

Workflow-based orchestration is an established concept (see Section 2.2 for more background on workflows) and applied in different domains, such as (big) data processing [91] and e-science [49]. It relies on a system, the workflow management system (or workflow engine), to orchestrate the operations. The operation, process or, in this case, composed function, is explicitly described as a set of interdependent tasks using a workflow language. This workflow definition is parsed and evaluated by the workflow engine to determine which functions need to be executed next. The workflow engine ensures that the data is passed from the output of one function to the input of the next. In Figure 4.6 the execution of a workflow-based composed function is visualized. A workflow definition describing the sequence of a, b and c is uploaded to the workflow engine. Once an external invoker triggers the execution of the workflow, the workflow engine evaluates the workflow invocation to decide which function to execute next. Using this approach, Based on the dependencies between the functions defined in the workflow definition, the workflow engine first executes a, then b, and then c. Once all the tasks have completed, the workflow itself completes, returning the result to the user.

Workflow-based function composition has started to be adopted by the serverless computing community from both industry and research. In the industry, the major cloud providers have started offering their own variations of workflow management systems to facilitate function composition: Amazon was the first to introduce AWS Step Functions: a workflow engine that bases its model on finite-state machines. Microsoft offers a more traditional workflow system with Azure Logic Apps. and, recently, Google has started offering a managed version of Apache Airflow to allow users to compose functions [15]. A number of use cases have been published since the release of these systems, such as Coca Cola [51] using AWS Step Functions to orchestrate serverless payment workflows, and The

[9] `https://docs.microsoft.com/en-us/azure/event-grid/overview`
[10] `https://serverless.com/event-gateway/`

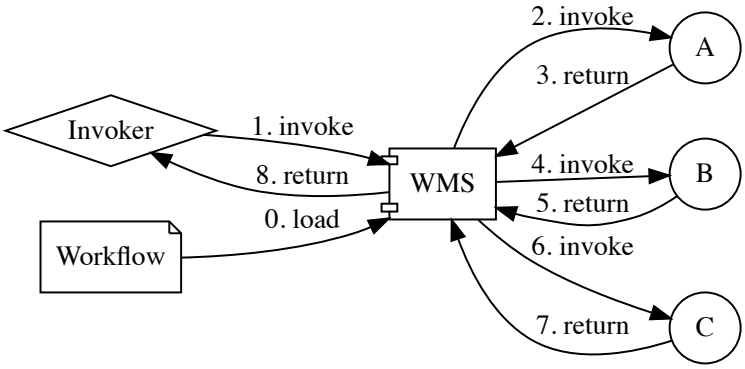


Figure 4.6: An example of a workflow-based composition of the chain of functions: a, b, and c. A central WMS reads a workflow definition and orchestrates the functions accordingly. The numbers in the edge labels indicate the order of execution.

Guardian using it in its subscriber management processes [16].

4.2.6. Qualitative Comparison of the Approaches

To evaluate the merits and demerits of the composition approaches for serverless function composition, we compare selected function composition approaches based on the requirements analysis of Chapter 3. Towards this evaluation we separate the the comparison criteria into two categories: criteria based on the *constraints* of the FaaS model (C1 to C4), and the *requirements* for a serverless function composition system (R1 to R7).

4.2.6.1. Results

In this section we present and discuss the results listed in Table 4.3. First, we describe the results from the perspective of the constraints. Then we analyze the approaches based on requirements.

Analysis of constraints. The function composition approaches should adhere to four constraints posed in Section 3.5: (1) they should be capable to expose the compositions using the HTTP interface akin to that of regular serverless functions; (2) they should not require introspection of the functions (functions should be black-boxes); (3) they should be generic enough to apply to any FaaS platform; and (4) they should not require the user to change their development workflow for serverless applications.

The results of this evaluation of constraints can be found in Figure 4.3. Direct and coordinator-based function composition use regular functions to orchestrate compositions, making the compositions themselves adhere fully to the FaaS model. The compiled function composition breaks function black-box constraint; it requires access to the source code of the functions to compile the functions into a new composed function. The event-driven function composition approach relies on message queues to orchestrate functions together. Although most FaaS platforms support one or more message queue implementations, these frequently differ from platform to platform, leading to non-portable and scattered function compositions. Workflows adhere to all constraints, except for the development complexity, as it is difficult to harmonize the writing of functions and defining of workflows.

Analysis of requirements. Besides the constraints inherent to the FaaS function (composition) model, we evaluated how the function composition approaches match the requirements posed in Section 3.4. For each approach we evaluate their (capability to) support the following dimensions: (1) their support for complex control flow constructs; (2) their support for long-running invocations; (3) the overhead (in performance and resources) that an approach requires; (4) the support for reliable invocations; (5) their support for versioning and upgrade strategies in light of frequently changing functions; and (6) their ability to scale with both the number of composed functions and invocations.

Approach	Constraints				Requirements						
	C1	C2	C3	C4	R1	R2	R3	R4	R5	R6	R7
Direct	●	●	●	●	●	○	○	○	●	●	●
Compiled	●	○	●	●	◐	○	●	●	○	○	●
Coordinator	●	●	●	●	●	○	○	○	●	●	●
Event-Driven	●	●	○	◐	○	●	●	●	○	◐	◐
Workflows	●	●	●	◐	●	●	◐	●	●	●	●

Table 4.3: Qualitative evaluation of the approaches based on the constraints (Section 3.5) and requirements (Section 3.4). The symbols used in the table encode how well each function composition approach satisfies a constraint or requirement: ●: Fully satisfied; ○: Not satisfied; ◐: Partially satisfied.

As visualized in Figure 4.3, akin to the comparison of the constraints the direct and coordinator approaches have similar results. Their use of regular functions to compose functions leads to multiple problems: they are constrained to the (tight) function runtime limits of the FaaS platforms, they are inefficient in minimizing overhead (leading to so-called 'double spends'[33]), and they lack fault-tolerance mechanisms to ensure reliability. The compiled approach does provide better fault-tolerance, because it runs the composition as an atomic unit of work, but is also constrained by the runtime limits of the platform and—due to the atomic execution—does allow the FaaS system to parallelize the execution. The event-driven function composition approach, by basing its approach on the subscription to events, constrains itself to simple control flow constructs—it is non-trivial to create event-driven compositions that contain loops or conditional branches. Furthermore, the lack of a central orchestrator makes it difficult to understand which functions depend on which functions at scale. Workflows have a well-established track of research on reliably orchestrating complex, long-running workflows at scale [91]. However, workflows, having a complex component stack and originally focused on large-scale, data-intensive workflows, can have a large performance overhead compared to other approaches.

4.2.6.2. Conclusion

Based on this evaluation of the function composition approaches, the most promising approach is workflow-based. It adheres to all constraints and dimensions posed in the requirements analysis. A major advantage of workflows is that they are well-proven in related fields that address distributed orchestration of processes/services/systems. However, *serverless workflows* are a new type of workflows, with especially the fast development cycle and the performance requirements differing from existing workflow domains. So, it is key to analyze what extend adaptations are needed in existing workflow systems and languages (which we analyze in Section 4.3 and Section 4.4) to make them apply to serverless workflows—which we formalize in Section 4.5.

4.2.6.3. Limitations

In this section we address the limitations of this survey of approaches to function composition. Specifically we identify three limitations: the scope, the qualitative nature of the criteria, and that the evaluation is aimed on all use cases.

First, the survey only accounts for five approaches. One could argue that there are many more options, especially variations or hybrid solutions of the evaluated approaches. Although, a more comprehensive study is most certainly part of future work, we argue, that for this initial survey these distinct approaches are diverse enough to capture most of the solution space to validate the decision for a workflow-based approach.

Second, the survey lacks a quantitative component. Although we could have spent effort in evaluating these approaches, for example, by using simulation, it seems unlikely that there will be surprising new findings—because the abstract approaches are intuitive. A full quantitative evaluation using real systems or full prototypes did not fit within the scope of this work.

Finally, in this survey we evaluated the approaches based on the complete set of use cases for function composition (see Section 3.1). The conclusions of this survey may have been different when considering the function composition approach for a specific use case. Although this is a valid concern, the intent of this study is to find a function composition approach for the set of generic serverless computing use cases.

Approach	Constraints				Requirements								
	C1	C2	C3	C4	R1	R2	R3	R4	R5	R6	R7	OSS	Docs
NodeRED	●	◐	◐	◐	●	◐	○	○	○	○	●	●	●
Openstack Mistral	◐	○	◐	○	◐	●	◐	●	○	●	●	●	●
AWS Step Functions	◐	●	●	●	◐	●	●	●	◐	●	●	○	○
Azure Logic Apps	●	◐	◐	●	●	●	●	●	◐	●	●	○	○
Pywren	●	○	●	○	○	●	●	○	○	●	●	●	◐
Apache Airflow	○	◐	○	○	◐	●	◐	●	○	◐	◐	●	●
Azkaban	◐	○	◐	◐	○	●	◐	●	○	◐	●	●	○
Luigi	○	○	◐	◐	◐	●	○	●	●	◐	●	●	●
Pegasus	◐	◐	○	◐	○	●	○	●	●	●	●	●	◐

Table 4.4: Qualitative evaluation of the state-of-the-art in workflow management systems with respect to the supportablity of serverless workflows. All of the systems in the evaluation use the workflow-based approach. The symbols used in the table encode how well each system satisfies a constraint or requirement: ●: Fully satisfied; ○: Not satisfied; ◐: Partially satisfied.

4.3. Survey of Workflow Management Systems

Because we concluded in Section 4.2 that a workflow-based approach is the most promising approach to function composition to pursue further, it is key to have a clear picture of the state-of-the-art in the workflow domain. The field of workflow orchestration is relatively well-established, containing substantial previous work and systems in both research and industry (see background Section 2.2 for more details on the field of workflows). Although, we identified use cases, requirements, and constraints unique to serverless computing (Chapter 3), existing research and systems in other domains could (partly) apply to this new type of workload. By analyzing the state-of-the-art, we gain valuable insight and established practices that benefit the design and evaluation of a workflow-based prototype for serverless function composition. The goal of this evaluation is therefore two-fold: (1) identify the state-of-the-art WMSs, which we explore using a domain-wide survey of systems in cloud computing; and, (2) determine which of these systems are most suitable to base the prototype on, which we achieve by qualitatively evaluating the WMSs.

4.3.1. Requirements

With this selection of state-of-the-art WMSs, to determine which systems are most suitable to base the prototype on, we systematically analyze the WMSs based on the requirements R1 to R7 (Section 3.4), and the constraints C1 to C4 (Section 3.5) of the requirements analysis (Chapter 3). In addition to these requirements and constraints from requirements analysis, to evaluate the quality of the systems specifically, we add two more criteria:

1. **OSS: Open-Source Software**: Is the system open-source? Is it easy to access, with a permissive license to use and adapt?

2. **Docs: Quality of the documentation**: does the project have high-quality, comprehensive documentation? Are the architecture and design choices clearly documented?

4.3.2. Methodology

Early on in the process of this work—surveying for interesting challenges to tackle in the systems side of distributed computing—we compiled a comprehensive list of state-of-the-art systems in multiple domains, from resource orchestration to build tools to workflow management systems[11]. As *state-of-the-art* (like *popular*) is an ambiguous term, systems are included based on a subjective evaluation of the following factors: adoption by industry, public exposure, development activity, use of state-of-the-art technologies, and indications in the respective communities that the system is state-of-the-art. Using this survey, we selected all the systems tagged as DAG-based workflow management systems. This resulted in a set of systems which were evaluated based on the requirements and constraints.

[11] https://docs.google.com/spreadsheets/d/1_UfFXnB_LeLe-rRqK__GKp8WdSUz3CDTNe1yz8eK3zA/edit

4.3.3. Results

In this section we describe and analyze the evaluation results displayed in Table 4.4. For each system we summarize the main capabilities, followed by a description of the pros and cons.

4.3.3.1. NodeRED

NodeRED[12] takes a visual approach to JavaScript-based function composition, which positions itself in-between workflows and dataflow programming (aptly using the term *flow* for its compositions). It focuses on simplicity, allowing its users (IoT developers and hobbyists) to deploy it anywhere and effortlessly add extensions. The execution model is kept simple too, with the execution of these flows taking place in within the, same, single process.

Though NodeRED focuses specifically on orchestration in the IoT domain, it adheres to most of the constraints: it has the option to expose all flows as HTTP services, and it requires limited information about the functions in the flows. By relying on functions for the control flow, it can express arbitrary complex control flow constructs.

The downside of NodeRED is its focus on simplicity. The system can only be deployed as a single process, inhibiting scalability. It runs flows inside the process, without a clear notion of fault-tolerance. Nor does it have a notion of versioning. Finally, its task model does not entirely match the constraints of the serverless model—for example, the flows contain GUI-related information.

4.3.3.2. OpenStack Mistral

OpenStack Mistral [5] is a workflow engine focused on automating common (DevOps) tasks within the OpenStack ecosystem, with tasks running *actions* similar to FaaS functions.

Despite being deeply nested into the OpenStack ecosystem, Mistral seems to (theoretically) generalize well beyond the OpenStack ecosystem; its execution and programming model do not assume the presence of other OpenStack services. It has a well-defined architecture, separating the workflow engine from the task executors, and from the API server, allowing it to scale well beyond a single node deployment.

However, Mistral does not fit the serverless model. Instead of relying on a FaaS platform for the execution of arbitrary functions, it has a notion of actions, which specify the execution of a task. Although these actions can be defined by the user, these actions are plugins within the system, and need to be defined using a strict interface. Nor is there the possibility to execute workflows synchronously over HTTP akin to the FaaS function model. Moreover, though Mistral provides built-in constructs for complex control logic, these constructs are not comprehensive. Finally, the performance overhead is not addressed, and does not seem a focus of the project.

4.3.3.3. AWS Step Functions

AWS Step Functions[13] is the most well-known, production-ready serverless WMS. It allows users express their workflows as state machines in the AWS states DSL. As a part of the AWS ecosystem, it provides deep integration with AWS Lambda and other AWS services.

AWS Step Functions focuses on being production-ready. It provides reliable and scalable executions of a large number of workflows—the default limit allows for the definition of $10,000$ workflows. Moreover, it satisfies the FaaS model, both in its pricing model, and in the explicit deference of any task execution logic to AWS Lambda or other (AWS) services.

However, AWS Step Functions is closed-source, without any documentation on its internal architecture. It also has a limited set of control flow constructs for users to express their workflows. Finally, it does not fully satisfy the constraints, as it is not possible to synchronously execute a step function.

4.3.3.4. Azure Logic Apps

Similar to AWS Step Functions, Azure Logic Apps[14] provides a closed-source, serverless WMS with deep integrations into other Azure cloud services. It provides a GUI and a comprehensive workflow language for creating and editing workflows.

Compared to AWS Step Functions, it provides the user with more and higher-level control flow constructs for common patterns (loops, conditionals, scopes). It is also one of the few analyzed WMSs

[12] https://nodered.org/
[13] https://aws.amazon.com/step-functions/
[14] https://azure.microsoft.com/en-us/services/logic-apps/

that allows for synchronous workflow executions. Finally, it can scale reasonably well, putting a (default) limit of 300,000 workflow invocations per second.

However, like AWS Step Functions, the system is closed source, preventing any insight into or adaptations of the architecture. It also violates the FaaS model in a small way, by expecting additional metadata (e.g., data bindings, connections) about the function—something that is only available for Azure functions.

4.3.3.5. PyWren

Pywren [79] is a serverless WMS originating from academia, without tight integration to a serverless provider—although at the moment of writing it only supports AWS. The main use case for the system is to speed up the execution of scientific workflows (e.g., Python notebooks).

The system does well what its intent is to do: execute a Python function in parallel to map a large data-set. For this it does not rely on functionality of a specific FaaS platform. The system scales well in both compositions (as they are managed and executed by the user) and executions (relying on the execution capacity of AWS Lambda).

However, PyWren does not aim to be a full workflow system. It has no notion of workflows beyond maps. It limits functions to Python functions with a specific signature, which it loads and runs on a single worker-function on AWS Lambda—violating the FaaS black-box constraints. Nor does it have a notion of workflow reliability (it assumes the orchestrator function will not fail), or function versioning.

4.3.3.6. Apache Airflow

Apache Airflow[15] is one of the most popular workflow management systems, and has recently been adopted by Google as a managed service as *Google Cloud Composer* [16]. For its data processing use case, the Python-based WMS focuses on deep integrations with various systems and providers (such as Hadoop, Spark, or AWS S3).

Key to its popularity is that the system has been battle-tested and is reliable for long-running workflows. With the right configuration, the system scales well in the number of workflows.

However, Airflow, despite its popularity, does not fit the serverless model. It has a unique developer workflow for developing and testing workflows using function-like plugins, called operators, which are managed by Airflow itself. Nor does it provide an API[17] for invoking workflows other than using an interval. Finally, as we find in the experimental evaluation of Airflow (see Section 8.2.5), airflow does not scale well in the number of executions.

4.3.3.7. Luigi

Luigi[18] provides a Python-based workflow orchestration targeting long-running, batch processes, using a scheduler and worker. Although initially Luigi could only be deployed as a single-process, unscalable deployment, in subsequent iterations has been extended with an (optional) centralized scheduler that can distribute tasks over multiple Luigi instances.

The benefits of Luigi are similar to that of Airflow; it has a large, active user and contributor community. It supports complex control flow logic through a feature called *dynamic dependencies* (allowing functions to depend on tasks at runtime).

However, Luigi has no notion of function versioning. Furthermore, the authors warn explicitly that Luigi is not focused on performance. Finally, like Airflow, Luigi assumes a lot of control over the task execution runtimes.

4.3.3.8. Azkaban

Azkaban [10] focuses on big data pipelines—specifically focusing on Hadoop—keeping the language purposely simple. The architecture tightly integrated, which results in workflows being executed on a single ”executor-server”.

Azkaban does well what its intent is to do: creating data pipelines within the Hadoop ecosystem. It has been engineered to production-scale, providing extensive authorization, authentication, and other

[15] `https://airflow.apache.org/integration.html`

[16] `https://cloud.google.com/composer/`

[17] though, at the moment of writing, an experimental API has been proposed to at least allow asynchronous workflow executions over HTTP

[18] `https://github.com/spotify/luigi`

governance options, and supporting SLA objectives.

However, similar to Airflow, it closely manages the lifecycle of the tasks breaking the FaaS black-box constraint. Next to breaking the constraints, Azkaban does not have any support for (complex) control structs that allow the definition of dynamic workflow. Finally, in contrast to the similar Airflow and Luigi projects, the documentation of Azakaban is lacking detail on usage and the architecture.

4.3.3.9. Pegasus

Pegasus [50], one of the most well-known, academically-originating WMS, it focuses on the large, highly-specific scientific workflows, which voids the need for complex control flow logic. Although the tasks are tightly coupled to data sources, the WMS takes care to only focus on workflow scheduling, deferring the task execution to external dedicated systems.

In contrast with most other WMSs, Pegasus has a well-documented, extensible, and well-structured architecture. The architecture consists of modular components (e.g., scheduler, engine, mapper), and defers task execution to pluggable, external systems. Furthermore, the structured approach to processing workflows is follows the workflow lifecycle in Section 2.2.3.

Despite the benefits, Pegasus is unlikely to handle serverless workflows satisfactory. It focuses on few, large-scale workflows, rather than small, frequently-executed workflows. It does not support synchronous execution. And, although it defers the task execution to external systems, it still manages a large part of the deployment and resource orchestration process.

4.3.4. Conclusion

As reflected in the results, there is no WMS that fulfills all the criteria associated for the serverless workflows. The WMSs that closely fit the serverless paradigm are proprietary and closed source (AWS Step Functions, Azure Logic Apps). Other, open-source, systems could be altered to fit serverless workflows, but this would require substantial reengineering efforts. Even with the reengineering efforts, often the fundamental model of these WMSs would not fit; functionality would need to be added, removed, or changed in an API-breaking way.

For these reasons we conclude that it is best to design and implement a new WMS as the prototype. This allows us to tailor the prototype specifically to serverless computing, FaaS, and serverless workflows, without the need to implement workarounds or fundamentally change the execution models or architecture of an existing WMS. However, as we will explain throughout the design (Chapter 5) and implementation (Chapter 7), we reuse concepts and components found in this evaluation of the state-of-the-art and in general WMSs.

4.3.5. Limitations

In this section we address the limitations of this qualitative survey of WMSs. We identify three limitations: the timeliness of the survey, the number of evaluated systems, and the depth of the evaluation.

First, the field of serverless workflow systems has, like the broader serverless computing field, evolved rapidly. At the moment of writing, many industry companies and FaaS platforms have started to evaluate and build various variations on serverless workflow systems. For example, concurrent with this work IBM released a prototype of a serverless workflow orchestrator of their own called IBM composer and the fn FaaS platform has added a composition mechanism with fn flow. Next to this development being an indication of the timeliness of this work, we also argue that this survey is useful for developers of newer workflow engines—to understand what already exists and how it relates to the serverless domain.

Secondly, there is large set of workflow and workflow-like systems that we did not evaluate in-depth. However, we selected this set of systems on a best effort approach to capture the key state-of-the-art systems. A full survey on workflow systems is beyond the scope of this work.

Finally, the survey does not go into depth when evaluating the selected systems. For the evaluation, we limited ourselves to basing the evaluation on the documentation, examples, and trying out the system in a basic setup. The results could have been more accurate if we invested more time in evaluating the systems based on realistic usage and understanding the source code. Again, we believe this is justified due to the scope of the project, and it is unlikely that the documentation differs widely from the source code.

Workflow Language	Format	FaaS functions	Complex Control Flows	Readability	Model fit
BPMN 2.0	XML	○	●	○	○
WS-BPEL	XML	○	●	○	○
YAWL	XML	○	●	○	○
WDL	Custom	○	○	●	◐
CWL	YAML, JSON	●	○	●	◐
Amazon States Language	JSON	●	◐	◐	●
Openstack Mistral Language	YAML	◐	◐	●	◐
Apache Airflow's Workflow format	Python	◐	●	●	◐
Azure Workflow Definition Language	JSON	●	●	◐	◐
Pegasus DAX	XML	◐	○	○	◐
Azkaban Language	Custom	◐	○	◐	◐
NodeRED Flow File	JSON	○	○	○	○
Luigi's Workflow Format	Python	○	●	●	◐
Amazon Simple Workflow Framework	Java	◐	●	○	○

Table 4.5: A qualitative evaluation of the state-of-the-art workflow languages. The symbols used in the table encode how well each workflow language satisfies a requirement: ●: Fully satisfied; ○: Not satisfied; ◐: Partially satisfied.

4.4. Analysis of Workflow Languages

As discussed in the background on workflows (Section 2.2), the design of the workflow language plays an important role in the WMS. It constrains the kind of computations the user can express, and a system needs to support. Furthermore, a well-defined workflow language (or a programming language in general [109] provides the user with a (intuitive) mental model how computations take place. And, can also function as a intermediate representation, to which applications written in higher-level languages can be mapped to.

The specific requirements, constraints, and use cases of serverless workflows (Chapter 3) make it unlikely that there is an existing workflow language can be plainly re-purposed. Therefore, the main goal of this section is to provide an analysis of what workflow languages fit *best*. On these closely-related languages we can base in turn the workflow language of the prototype on (see Section 5.2).

4.4.1. Requirements

Similar to our evaluation of the state-of-the-art in WMSs, we conduct this analysis based on specific criteria. These criteria reflect the characteristics of the *serverless workflows* (as defined in the requirements analysis in Chapter 3). We consider the following criteria:

1. **Supports expression of generic FaaS functions**: Does the language allow the expression of generic FaaS functions—tasks with a reference to a remotely defined and deployed, HTTP-based, black-box function?

2. **Supports notation of complex control flows**: Does it support the notations or syntax for users to express complex control flow constructs, such as loops, conditionals, and error handling.

3. **Readability**: What is the perceived complexity of the language? Can workflows be expressed in a concise and readable manner?

4. **Uses a well-established data format**: Does it use a well-established data format that makes it easy to understand and read for people, such as Python, YAML, JSON, or XML?

5. **Model fit**: Does the language fit the serverless workflow model, or does it have features that need to be removed or changed?

4.4.2. Methodology

We base the selection of the workflow languages on the systems found for the survey of WMSs (Section 4.3), and on a literature search for "workflow language". We analyze the workflow languages found, and well-established languages well-known in the workflows community (WS-BPEL, CWL, YAWL, etc.) in Table 4.5.

4.4.3. Results

In this section we describe and analyze the results listed in Table 4.5. We analyze the results grouped by their main domain and goal: BPMN-like workflow languages for modelling business processes; workflow languages originating from a specific WMS; and, ”universal” workflow languages which have attempted to standardize workflow definitions across multiple platforms.

4.4.3.1. Business-Process Workflow Languages

Workflow languages from the business processing domain, BPMN 2.0 [103], Yet Another Workflow Language (YAWL)[128], and WS-BPEL [124], support complex control flows at the cost of complexity [136]—complexity that is often unneeded [143]. Similarly, these business process-oriented workflow languages largely rely on visual editors for non-technical users to create workflows, making the workflow definitions themselves hidden and, in most cases, unreadable.

4.4.3.2. WMS-Specific Workflow Languages

A majority of the selected WMSs define their own languages, which often results in workflow languages that are tightly coupled to the execution model and technologies of the WMS. Azure's Workflow Definition Language consists of a expressive and readable format, but it contains various integration specific to their cloud platform. Similarly, NodeRED's Flow file contains data specifically for visualizing the flow in their visual editor. Luigi's format deeply integrates the workflow definition with the connectors and operators.

Luigi and Airflow opt for the use of a general-purpose programming language to define workflows. Both use Python files to define a Python-internal workflow data-structure. Although this makes these languages (with all the existing Python tooling) far more user-friendly and familiar to users, the available complexity of Python can lead to confusion among users, about what Python functionality is, and is not, supported. Moreover, the Airflow language makes clear assumption about the use case for the workflows, requiring the specification of timing-related information, such as the start time and interval.

4.4.3.3. General-Purpose Workflow Languages

Finally, there are multiple efforts to provide a standardized *universal* workflow languages, which include the Common Workflow Language (CWL) [27], Workflow Description Language (WDL) [70], and Yet Another Workflow Language (YAWL) [128]. However, despite the intentions these efforts have still introduced assumption about the workflow execution model and and WMS architecture, such as the existence of a unified filesystem, which makes directly reusing these languages difficult for a serverless use case.

4.4.4. Conclusion

The proposal and design of a unified serverless workflow language is a full-sized project on its own. Moreover, with the volatility and change occurring in the serverless domain it would be challenging to predict which use cases, functionality, and platforms will stay relevant in the coming years. Similarly, because none of the surveyed languages fits the serverless model, re-purposing an existing workflow language for serverless workflows would turn out into a difficult integration process; supporting certain and not other functionality in the language, extending the workflow language, or changing execution model assumptions.

Instead, we design a new, minimal workflow language that evolves alongside the workflow engine prototype to explore exactly if and how a generic serverless workflow language will look like. Nonetheless, we do incorporate into this language favorable aspects of the languages analyzed here. The language—described in the design (Section 5.2) and fully specified in the Appendix A—contains a purposely minimal syntax (similar to CWL, and WDL) to avoid introducing too many (presumptuous) assumptions about the workflow model. We assume the workflow language to be a low-level—yet readable—language to which higher-level programming language can be 'compiled', akin to Pegasus' DAX and the Azkaban's language. The workflow language is declarative using a data format, and we follow a syntax similar to that of OpenStack Mistral's language [5] and Azure's Workflow Definition Language.

4.5. Serverless Workflows

Based on the requirements analysis (Chapter 3) and the preceding surveys, we argue that with FaaS and serverless computing have introduced a new form of function composition or orchestration. With our conclusion of the high relevance of workflows to serverless computing (Section 4.2), we pose this new form of function composition as a novel type of workflow: serverless workflows.

Definition 4.5.1. Serverless Workflows are complex, dynamic workflows of FaaS functions or serverless services, which are frequently and irregularly executed.

Similar to other types of workflows, such as business-critical workflows, BPMN, and scientific workflows, we argue that serverless workflows have unique characteristics, or dimensions, that make them unique from other types of workflows. We describe serverless workflows in two ways: in Section 4.5.1 we present the dimensions of such workflows, where, in each dimension, we describe how the dimension resembles the constraints and requirements from Chapter 3; and, we contrast serverless workflows with existing classes of workflows in Section 4.5.2.

4.5.1. Dimensions

Serverless workflows can be characterized by the following dimensions:

D1 Minimal control over the task execution: A workflow engine orchestrating serverless workflows follows the general FaaS model (constraint 3.5 in Section 3.5), having minimal control over the task execution environment. The tasks themselves should be considered black-box functions; the workflow engine provides inputs and receives the output or error. It typically can only interact with FaaS functions or serverless services over the same interface as other serverless users, over an abstract protocol, such as HTTP.

D2 Frequent, irregular executions: Serverless workflows, like serverless functions in general, are used in a widely varying range of domains with differing workload characteristics, from supporting web applications to batch processing. As described in Section 3.3, this means that serverless workflows typically have more frequent executions, compared to traditional workflow domains. At the same time the arrival pattern of the requests for workflow executions, like serverless functions, have a more bursty workload profile, with sudden periods of frequent executions.

D3 Deadline-aware: With the real-time, business-critical use cases (see Section 3.1), such as web applications and edge computing, low-latency is core metric to evaluate serverless workflows. Latency is the duration between the user's request for the workflow invocation and the final response to the user. Although the performance of the workflow depends for a large extend on the performance of the underlying FaaS functions that are executed by the workflow tasks, a serverless workflow engine should minimize the performance overhead it adds to the overall execution of the workflow.

D4 FaaS-like Cost Model: Because the serverless workflows should not differ from serverless functions, a key characteristic of FaaS—the cost savings of not paying for idle resources—also holds true for workflows. Akin to regular functions, workflows should not incur additional costs during execution, nor should they incur costs when unused.

D5 Unknown or high runtime variability of tasks: Tasks (the functions executed in the workflow) within serverless workflows suffer from highly variable and difficult to predict runtimes. This has three reasons: (1) the performance of underlying FaaS functions is affected by different sources in their complex technological stack—such as the cold starts; (2) the tasks are more granular compared to traditional workflows, magnifying any existing performance variability; and, (3) functions are frequently updated, limiting the options to accurately profile or model the function's performance over time.

D6 Dynamic requirements Traditional workflows have static requirements for a given workflow. Not only do applications have more non-functional requirements that need to be considered and affect the workflow, such as latency, reliability, elasticity, they are also dynamic. During the lifetime of a complex workflow, the requirements of users are likely to change.

Workflow Type	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10
Scientific Workflows	○	○	○	○	○	○	○	○	○	○
Business Process Workflows	◐	○	○	○	◐	○	●	●	○	●
Web Service Workflows	●	●	◐	○	○	●	●	◐	○	●
Big Data Workflows	○	◐	○	○	○	○	○	○	◐	◐
Serverless Workflows	●	●	●	●	●	●	●	●	●	●

Table 4.6: A comparison of workflow types based on the dimensions of serverless workflows. The symbols used in the table encode how well each workflow type satisfies a dimension: ●: Fully satisfied; ○: Not satisfied; ◐: Partially satisfied.

D7 Multiple, independent workflow orchestrators: Although a workflow engine can manage its own serverless workflows to an extend, sub-flows occur often, uncontrollable by the main workflow. These sub-flows might be orchestrated by a different workflow engine residing at a different cloud provider, or even another instance in workflow engine deployment. This has implications for many aspects of the serverless workflow, such as the ability to monitor or visualize the workflows across multiple workflow engines. Moreover, this also leads to challenges and opportunities for investigating options for workflow engines to share contextual information about workflow to improve performance.

D8 Dynamic structure: Serverless workflows are used for dynamic, complex workloads in a range of use cases. In contrast to traditional workflows, where workflows are expressed as large static, purpose-build DAG, serverless use cases require serverless workflows to be more dynamic in nature. A serverless workflow typically should be able to replace or augment workflows that are otherwise expressed using general-purpose programming constructs, such as (unbounded) loops, error handling, and conditional branching. From a workflow perspective this means that serverless workflows require support for dynamic workflows; workflows that can alter their own structure at runtime to support more complex control flow constructs.

D9 Task Prioritization: Because serverless workflows focus mainly on latency; determining the correct response to the user as fast as possible, this leads to an implicit task priority. Any task on the *hot path*—that is, the minimal set of tasks required to be completed to return the desired response—should have a higher priority than tasks that are less timing-sensitive. Although this prioritization is implicit in any serverless workflow, a serverless workflow engine could improve the performance by allowing users to explicitly indicate priorities or other non-functional requirements for individual tasks.

D10 Granular tasks In contrast to other workflow domains, the tasks of serverless workflows to have smaller runtimes. As we found in the workload part of the survey of current serverless use cases in Section 3.3, this is in part due to FaaS platform restrictions; AWS Lambda has a maximum runtime of 5 minutes. Besides the technical limitations, the constrained, stateless nature of FaaS functions seems to nudge users to keep the runtime of their functions short.

4.5.2. Workflow Comparison

To validate that the combination of the dimensions in the previous section makes serverless workflows a distinct type of workflow, we map existing workflow domains onto these dimensions as shown in Table 4.6. For this we consider five common types of workflows: scientific workflows, business process workflows, web service workflows and big data workflows.

4.5.2.1. Scientific Workflows

Scientific workflows, as the name suggests, are commonly used in academia to automatically conduct large-scale, distributed experiments. Common examples of scientific workflows are LIGO (detection of gravitational waves) [61] and MONTAGE (image mosaic composition) [35].

These workflows are more or less the direct opposite of serverless workflows. In contrast to serverless workflows, scientific workflows are large, static workflows. These workflows are executed infrequently with huge amounts of data to process, where they focus mostly on throughput—performance overhead of the WMS is less important in light of the hour of day-long task execution durations. The

tasks themselves are typically long-running, and (nearly) identical across workflows, allowing systems to do offline and online task runtime prediction. Moreover, scientific workflows are purposely constructed for a highly-specific use case, meaning that they typically do not require dynamic requirements, structures or task priorities.

4.5.2.2. Business Process Workflows

Business-process workflows, with standards such as BPMN [103], are popular to model both offline and online business processes. Similar to serverless workflows, these workflows integrate services over standardized interfaces, deferring the task execution to dedicated subsystems.

However, business process workflows do not extend well beyond their main use case of business workflows. As we noted in the background on workflow formalisms, BPMN is a complex workflow formalism, focused solely on the domain of process modelling. These types of workflows tend to be executed less frequent, and lack a focus on the performance of individual workflow executions. Nor is there a notion of a serverless cost model in business process workflows.

4.5.2.3. Web Service Workflows

A precursor to the current serverless computing, web services and SOA also enabled the emergence of web service workflows [134]. These types of web service workflows are most similar to serverless workflows, focusing on composing the interaction between micro- and web services [88]. The use cases of these workflows consist are similar to that of those of serverless workflows: web applications, and business processes.

However, web service workflows lack a few dimensions that are key to serverless workflows: (1) there is no notion of a serverless cost model; (2) the model does not consider highly variable runtimes of tasks; (3) it does not consider frequently updated workflow versions, and (4) it does not contain a notion of dynamic requirements or workflow structure.

4.5.2.4. Big Data Workflows

Big data workflows relate to the distributed processing of big data workloads in systems, such as Hadoop and Spark [59]. Similar to scientific workflows, these large, static workflows are opposite to serverless workflows, with the only similarity being that that the tasks (queries, filters, maps) can be relatively small, due to the sheer size of tasks.

However, big data workflows lack all other aspects: there is no notion of dynamic structure or requirements; there is no notion of a serverless cost model; nor is there a black-box model for the task execution.

5

Design of Fission Workflows, a Serverless Workflow Management System

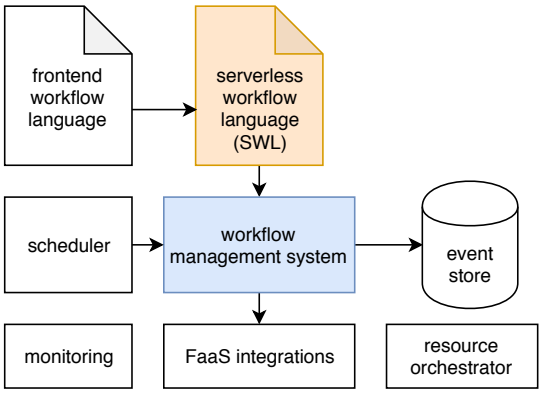


Figure 5.1: The aspects of Fission Workflows covered in this chapter: the Serverless Workflow Language (Section 5.2), and the architecture of the serverless workflow management system (Section 5.3).

In the serverless function composition survey (in Section 4.2) we found that a workflow approach to function composition is a promising direction to take. We further concluded in the survey that none of the existing workflow management engines or function composition systems meets the requirements and constraints defined in Chapter 3. Thus, in this section we present the design of Fission Workflows, a workflow-based approach to serverless function composition.

Fission Workflows aims to be an *extensible* serverless workflow management system. We focus, on minimizing the critical path of workflow invocations, and ensuring that the design consists where possible of easy to scale, stateless components where possible. The design focuses on fault-tolerance, achieved by utilizing control theory (see background in Section 2.3) and event sourcing [58]. The execution model provides a small, yet highly expressive, set of primitives that enable users to model their workflows. Finally, by defining clear interfaces, and by constraining the workflow engine to a limited set of responsibilities, the system is highly extensible, and in particular to allow users to add other FaaS platforms, add support for other workflow languages, and replace the workflow scheduler.

Originally intended as a research prototype, the scope of the workflow engine described in this chapter is limited to the functionality needed to validate the research questions and to provide a basis for experimental evaluation of functional and non-functional requirements. Therefore, the research prototype does not focus on providing functionality similar to state-of-the-art in workflow management systems, such as complex visualizations, extensive graphical user interfaces, multi-tenancy support, administrative interfaces. This project also does not explore functionality regarding multi-cloud, multi-FaaS

platform function composition. Finally, other functionality that is considered related, but not required, for this prototype: sophisticated (intermediate) data management, function and workflow registries, and alternative deployment environment and options. Section 9.2 discusses on these potential extensions and other future work.

This chapter relies on several prior publications. Both the serverless workflow language and workflow management system rely for their design on the concepts, constraints and definitions of serverless computing and FaaS functions, posed posed in previous work [132–134]. The serverless WMS further assumes that the underlying FaaS platform aligns with the FaaS reference architecture defined in concurrent work [135]. Finally, for Serverless Workflow Language (SWL) we rely on the DAG workflow formalism in part due to the prior work on workflow formalisms [136].

The remainder of this section provides a comprehensive overview of the design of Fission Workflows, and provides insight in the design decisions made:

- Section 5.2 describes the data and execution model of the *Serverless Workflow Language* (SWL).
- In Section 5.3 we provide an overview of design of the workflow management system, Fission Workflows.
- In Section 5.4 we analyze the designs of both the SWL, and the workflow management system satisfy the requirements (from Section 3.4) and constraints (from Section 3.5).

5.1. Design Principles

Based on the requirements (see Section 3.4) and constraints (see Section 3.5) of a satisfactory design, we approach the design process by defining a set of design principles, which align with the design vision of the AtLarge group [75]. These design principles are not meant to be strictly adhered too or analyzed (see Section 5.4). Instead, they serve to summarize the main ideas behind the design of both the workflow language and workflow engine. In the remainder this section, we briefly highlight each of these core design principles:

P1 View workflows as functions: Serverless workflows can be executed using the same interface as serverless functions. In its essence, the end-user does not have to be aware of whether the serverless function they are executing is a regular function or a serverless workflow. This ensures that we adhere to C4 (from the constraints in Section 3.5): no changes to the serverless development and operational workflow are needed when using serverless workflows. Next to the interface, the notion of workflows-as-functions also implies that the workflow model requires a similar execution model as serverless functions (see Section 2.1.2): workflows accept inputs at the start of the invocation; workflows either run to completion or return a failure; and, serverless workflows have an explicit output.

P2 Delegate task execution: Whereas existing workflow engines typically assume a large degree of control over the deployment and execution lifecycle of the tasks [135], in Fission Workflows we delegate as much task execution logic to the underlying FaaS platforms, adhering to the layered FaaS reference architecture (see Section 2.1.4). Following C2 (in Section 3.5), the tasks are treated as black-boxes. The workflow language and engine do not control any part of the deployment process. Similarly, during the runtime of the task the workflow engine does not expect control over execution beyond the functionality of a typical FaaS platform (see Section 2.1.4.2), which helps conform to C3.

P3 Focus on the execution model: Next to delegating the task execution and resource orchestration to dedicated subsystems (P2), the design targets the execution model, deferring the development model to (external) subsystems. Foreshadowing, the design of the serverless workflow language focuses solely on the execution model. Similarly, in the design of the workflow engine (Section 5.3.3), we focus solely on the components that are required for the execution of workflows; we defer functionality that defines how workflows are defined, or when the workflows need to be invoked, to external systems.

P4 Utilize event-driven architecture: Since serverless computing is highly dependent on event-driven architectures [134], the design emphasises the use of events. Key in this design is the

idea that workflows can be represented as a sequence of events (which include: tasks started, tasks finished, etc). In the execution model of SWL we use event types to define the transitions between states. In the WMS, using this event-based representation of a workflow we rely on event-sourcing [52, 58]—a technique in which we persist the event sequence instead of the (workflow) state, and reconstruct the state from these events.

5.2. The Programming Contract: A Serverless Workflow Language

In the requirements analysis (Chapter 3), we found that the characteristics of the workloads of serverless computing (and specifically of FaaS) differ distinctly from existing types of workloads. Based on this finding, we proposed (in Section 4.5) a new type of workflow: *serverless workflows*. In Section 4.4, we analyzed existing workflow languages to determine how well they fit this new workload type. We found that none of the existing workflow languages fits these serverless workflow accurately, which therefore requires a new serverless workflow language.

In this section we present a workflow language for this domain: Serverless Workflow Language (SWL) is our workflow language for modeling *serverless workflows*. In contrast to existing languages, it focuses on a intuitive, functional execution model which contains minimal constructs while being expressive enough for complex control flows. The data model provides comprehensive functionality for executing cloud functions.

We limit the scope of this part of our project providing workflow language capabilities to Fission Workflows to solely the executable workflow language—the workflow specification that the workflow engine parses to execute a workflow—rather than specifying a complete suite of (programming) languages and tools to express workflows. This distinction between *front-end* workflow languages and a *back-end*, executable workflow representation is similar to the approach taken by the Pegasus WMS with its DAX workflow language [50]. Although developer-oriented tooling and (programming) languages is a critical challenge for serverless workflows and serverless computing in general. [134] this is considered out of scope for this work. Instead we follow practices in existing workflow languages, implementing a declarative workflow language, called *SWL-YAML*, as an initial implementation of SWL in the implementation (Section 7.1.2). Besides the developer tooling and (programming) languages, we also consider non-functional concerns, such as multi-tenancy, authentication, and authorization, as future work.

In the remainder of this section we present SWL using two perspectives: using the data model (Section 5.2.1) to introduce the main (data) constructs in SWL, and using the execution model (Section 5.2.2) to provide an overview of the main (execution-related) concepts and lifecycles. Finally, in Section 5.2.3 we present dynamic tasks, a key extension to SWL to support complex control flow constructs.

5.2.1. SWL Data Model

In line with other workflow languages (see Section 2.2), SWL supports explicitly the notions of workflows and tasks. We choose to model each workflow as a DAG, because we found in prior work [136] that DAGs: (1) are most popular in the field of computer science, in comparison with Petri Nets and BPMN; (2) have the smallest number of constructs in our view; and (3) are expressive enough to express additional properties for specific tasks, as our analysis reveals.

Figure 5.2 visualizes the data model of SWL. The figure highlights in gray the two top-level data structures that users interact with: workflows and workflow invocations. We present each component in Figure 5.2, in turn.

A *workflow* (component M6 in Figure 5.2) consists of a sequence of interdependent *tasks* (M3) whose completion in the right order ensures a result or performs a composite procedure. To adhere to constraint C4 (see Section 3.5), the workflow contains, besides *tasks*, *inputs* (M5) that indicate expected or default parameters provided to the workflow upon execution, and an *output* (M5) to indicate which task output should be used as the workflow output. This enables workflows to adhere to the same API as serverless functions—which allows users to interact with workflows as if they are regular functions.

A *task* (component M3 in Figure 5.2) represents an atomic unit of execution within the workflow. Besides a unique identifier, it contains of three properties: a *function reference* (M2) that indicates which function or service to run, *inputs* (M5) to provide to the function; and *dependencies* (M1) that need to be satisfied before the task can run.

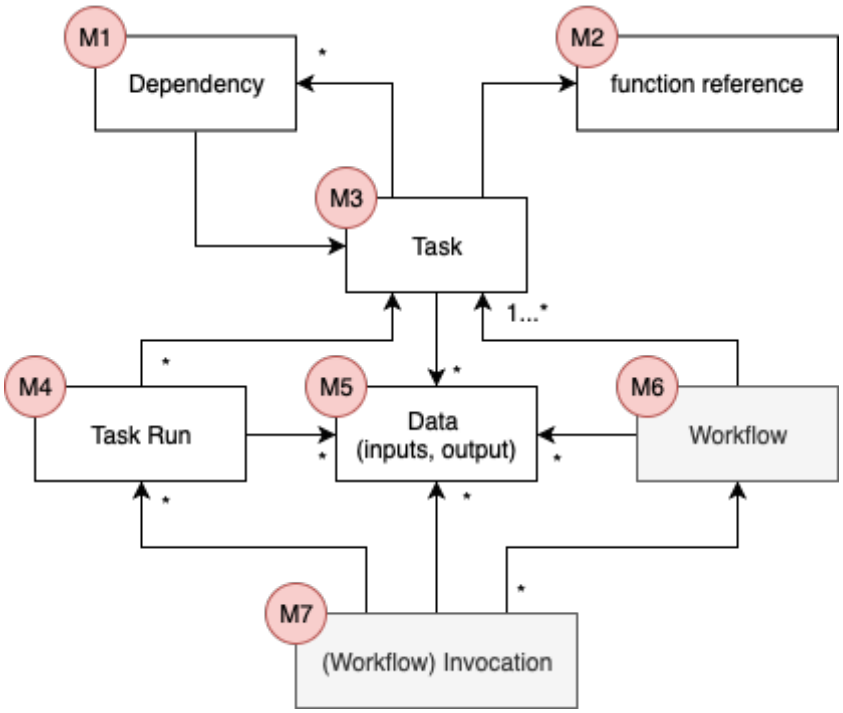


Figure 5.2: A class diagram of the SWL data model. The data structures in the gray boxes are top-level; users create, retrieve, and interact with these models.

A *dependency* (component M1 in Figure 5.2) signals a (control) dependency relationship between two tasks (M3). Constraints can be added to the dependency to indicate the type of the dependency. For example, besides the typical dependency present in DAGs where a task can only start when the task(s) it depends on has completed, the SWL model supports that the dependency could indicate that a task should only when another task has failed.

A *workflow invocation* (component M7 in Figure 5.2) is an execution instance of a workflow; it is the data structure responsible starting, tracking, and archiving the progress and result of the workflow invocation. It contains user-supplied parameters (*inputs*) and is directly tied to a single workflow definition. For each of the tasks of the workflow that need to be executed, it instantiates *task runs*. Because SWL adheres to the FaaS function model, a workflow invocation completes with an *output*.

A *task run* (component M4 in Figure 5.2) is to a task what a workflow invocation is to a workflow; it represents an execution instance of a task. The task run is responsible for tracking the execution of a single *task*. It specifies the *inputs* to be used for the execution, which it takes from the inputs of *task*—potentially augmented as is the case with expressions (see Section 7.1.2). In addition, a completed task run contains (a reference to) the output data.

5.2.2. SWL Execution Model

To allow workflows to be substituted by regular FaaS functions, and adhere to the other constraints listed in Section 3.5, the execution model of SWL follows the FaaS execution model (see background in Section 2.1). It focuses on simplicity, separates the software flow from the control and data flows, and supports options to extend the execution model using explicit descriptions of the lifecycle.

As depicted in Figure 5.3, SWL follows the FaaS model of separating the management and execution of functions lifecycles (see P3 in Section 5.1), by separating the definition of workflows from the invocation of workflows—see the serverless operational patterns in Section 2.1.4.2. For this reason, we describe in turn these two processes.

5.2.2.1. Execution Model for Workflow Definitions

As in the general workflow model, adding a new workflow to a WMS in SWL is more than a simple database operation (see Section 2.2.3). The *workflow definition* provided by the user will likely need to undergo a number of checks and transformations, before it is available for invocation (see Figure 5.3).

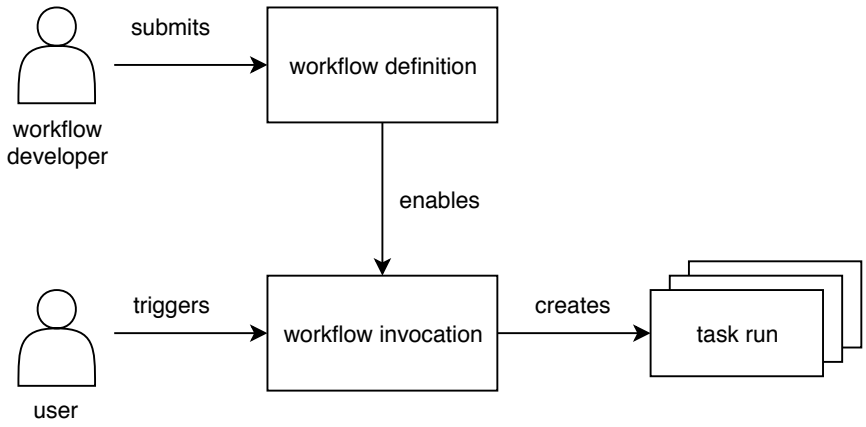


Figure 5.3: An overview of the execution model of SWL: the submission of workflows (Figure 5.4) is separated from the workflow invocations (Figure 5.5), which comprises task runs (Figure 5.6).

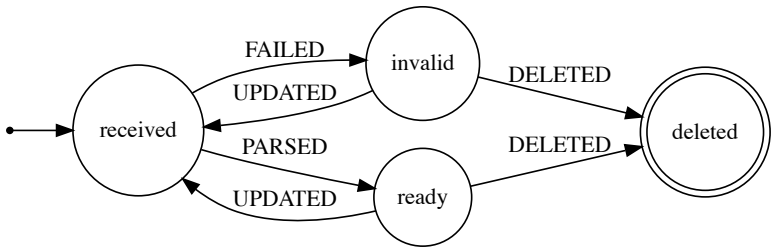


Figure 5.4: The lifecycle of a workflow definition: a workflow definition received by the workflow engine, after validating/parsing steps, is either ready or not (failed) for workflow invocation. It can be permanently deleted, or reevaluated when updated. The edges are annotated with the name of the event that triggers the transition.

The workflow is not guaranteed to pass these stages, which can lead to the workflow to be (temporary) invalid. Furthermore, these checks and transformations need to be rerun if the workflow is updated.

Based on this non-trivial execution model of workflow definitions, we define a Finite State Machine (FSM) and the mechanisms to enforce this execution model, as depicted in in Figure 5.4. A workflow received by the workflow engine is considered to be unavailable to invoke until it is in a `ready` state. To transition from a `received` state into a `ready` state, the workflow goes through a pipeline of checks, parsers, and resolvers. If, along this process, a stage fails, the workflow enters in an `invalid` state. Unless a workflow is `deleted`, the workflow can updated, changing the state to `received`, and triggering the pipeline again.

5.2.2.2. Execution Model for Workflow Invocations

We explicitly define the execution model for a *workflow invocation* for several reasons: (1) it highlights that workflows follow the execution model of serverless functions (`0.01`); (2) it emphasises the separation of the workflow definition and workflow invocation (P3); and, (3) it addresses what role events play in the execution model (P4).

To enforce the execution model, we define it using two FSMs. The workflow invocation FSM (see Figure 5.5), which describes the states and the valid transitions on the invocation-level. Then, for the execution of a task within the workflow invocation, we define a separate FSM (see Figure 5.6), which includes the valid task states and transitions related to the specific task.

Once a workflow definition is `ready`, the system user can trigger a *workflow invocation*, which results in an instance of the workflow. As depicted in Figure 5.5, the *workflow invocation* starts in the `queued` state. Once the workflow engine starts invocation, the state of the workflow-invocation changes from the queued state to the `in_pogress` state. While in the `in_progress` state, the

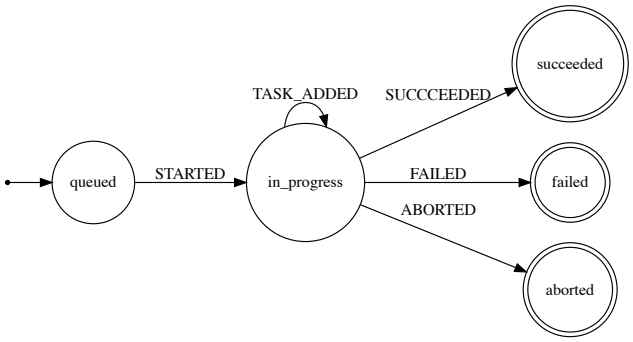


Figure 5.5: The lifecycle of a workflow execution: within a workflow, the task is started or skipped, and runs to completion resulting in a success or error. The edges are annotated with the name of the event that triggers the transition.

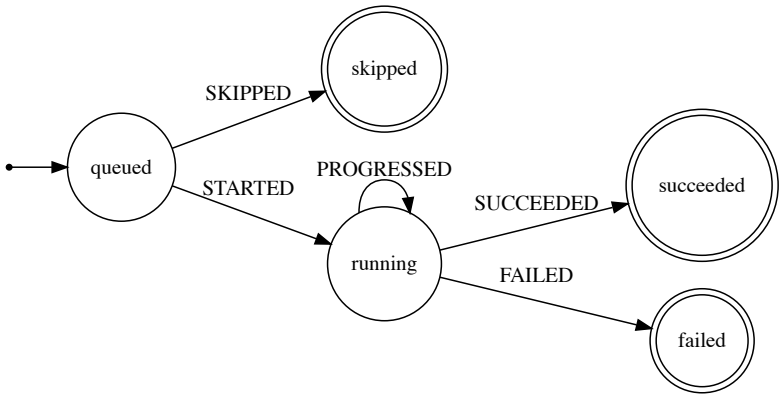


Figure 5.6: The lifecycle of the execution of a task: within a workflow, the task is started or skipped, and runs to completion resulting in a success or error. The edges are annotated with the name of the event that triggers the transition.

workflow invocation forks into multiple task executions. A workflow invocation runs to completion (`succeeded`, `failed`, or `canceled` by the user). To support dynamic workflows (D8 in Section 4.5), the lifecycle of the invocation allows for runtime adaptation of the workflow, by supporting the addition of a task while `in-progress` (see Section 5.2.3).

Within the workflow invocation, SWL models each execution of a task, a *task run*, as a finite state machine. Figure 5.6 depicts the states and events (transitions) of a *task run*. The task run will occur within the context of a workflow invocation, and requires the associated workflow to be in the ready state. Before the task can be started all its *inputs* need to be available—that is, any placeholders or data references need to resolved. Different from the task lifecycles in existing workflow languages (and especially BPMN), SWL has an intentionally simple execution model. Tasks are—like the FaaS functions—*atomic* units of work. They will only be executed *at-most-once*, and once started they run to completion. They have only three final states: `skipped`, `succeeded`, and `failed`.

5.2.3. Extension: Dynamic Workflows

Although the basic SWL model allows for the execution of traditional, static, DAG-based workflows, an important extension that can be added to SWL is to allow the expression of *dynamic workflows*, that is, workflows which can alter their own structure at runtime (D8 in Section 4.5). The support for these dynamic workflows allows the workflow developers to construct more complex control flow constructs, such as conditional branches, and bounded or unbounded loops.

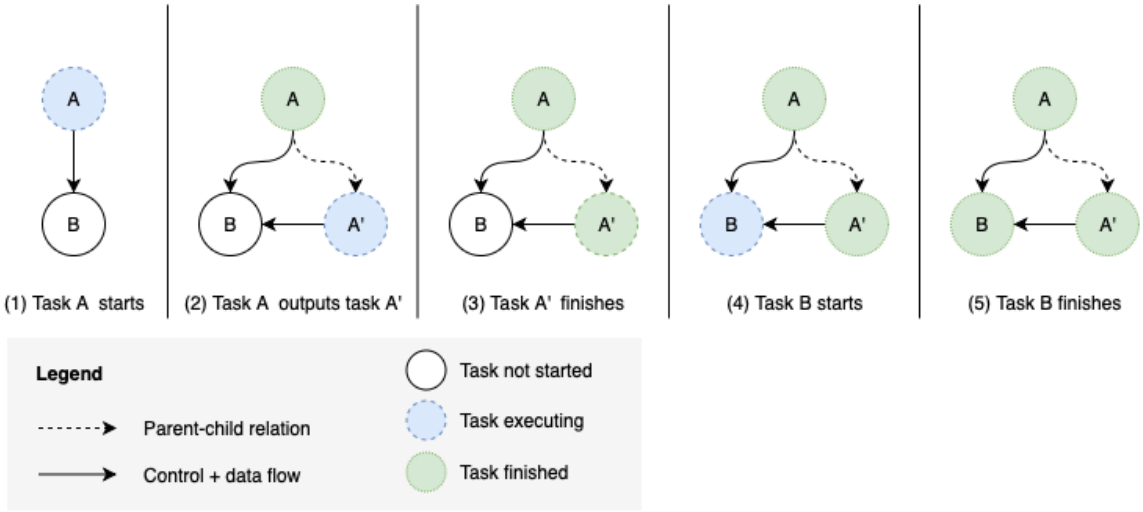


Figure 5.7: An example of the approach SWL takes to supporting dynamic workflows. Note that the control + data flow are depicted as a forward dependency; Task B depends on Task A.

.

As we found in the survey of WMS (Chapter 4), dynamic workflows have been proposed multiple times over the past decades, in separate workflow domains, and using different approaches. BPMN-compliant [138] or related workflow languages [140] provide an exhaustive set of high-level, interdefinable [143], constructs to model dynamic workflows; yet, it is this comprehensiveness that makes their execution engines difficult to develop and to validate, and that limits performance [110]. The scientific WMS FireWorks [76] proposes a execution model for generic dynamic workflows by providing a set of dynamic constructs called *FWActions*, which allow tasks to alter the workflow definition, cancel other tasks, exit the (sub)workflow, change intermediate data, and intercept and alter control flows. Other state-of-the-art WMSs, such as AWS Step Functions and Azure Logic Apps, often take a middle-ground design approach by supporting a subset of dynamic workflows and by providing the user with a small number of constructs, such as for-loops and if-statements (modeled closely to their equivalents in programming languages).

In contrast to existing approaches, SWL requires an approach to dynamic workflows that follows the model and characteristics of this serverless workflow domain. First, the approach should be simple—it should not complicate the existing execution and data model. Second, the approach should be generic; it should be flexible enough to support the diverse, and evolving, use cases and workloads of serverless workflows.

To support dynamic workflows in SWL that we introduce a minimal extension to the workflow language. We extend the execution and data model with a single operation: the output of a task can be—instead of plain data—a new task or workflow. This mechanism adheres to the principle of parsimony [40] and is similar to *fork-exec* in UNIX [109], where instead of executing the new process internally within the current process, it is added explicitly to the system as a new process.

The support for dynamic workflows in SWL works as follows: In a workflow, a task returns (instead of regular data) a task or workflow definition—called a flow. The workflow engine recognizes this flow and converts the flow to a complete workflow definition, which it then adds to the workflow repository. It then creates a *proxy task*, which contains all instructions for calling the dynamic workflow. The WMS adds this proxy task to the workflow invocation and attaches links it to the original task using a *parent-child* connection. This connection signals two things: (1) tasks that depend on parent task, need to wait for child task to finish too; (2) tasks depending on the data of parent task will instead receive the data of child task.

An example of this approach to support dynamic workflows in SWL is depicted in Figure 5.7. In stage 1, the initial workflow is depicted with task A and B forming a pipeline, in which task B depends on running task A to complete. In stage 2, task A completes and outputs a *flow* (that is, the task output is a task or workflow). This flow is converted to a workflow, and a proxy task that will execute this dynamic workflow is added to the current workflow invocation, as Task A'. In this example, this means

that Task B cannot be started until Task A' finishes. Therefore, task A' starts in stage 2 and finishes in stage 3. Now that both Task A and Task A' have completed, we can start Task B in stage 4. Because of the parent-child relation, if Task B required output data from Task A, it will receive the output data of Task A'. We complete this workflow in stage 5, when Task B has finished too.

5.3. Overview of the Fission Workflows Architecture

In this section we describe the architecture of Fission Workflows, the serverless workflow management system. We focus here on the main components core to the system, which are numbered and highlighted in Figure 5.8.

C1 The **user interface** is needed to provide users and consuming services tools and libraries to effectively interact with the *apiserver* (C2) of the WMS. The user interface component in this architecture is kept abstract on purpose; The tools used to interface with the WMS depend on the type of user. For a physical user, the user interface typically comprises a Command-Line Interface (CLI) or a Graphical User-Interface (GUI), to access and manipulate their workflows. For systems interfacing with the WMS, the tooling generally consists out of API programming libraries.

C2 The **apiserver** is responsible for exposing a clear, (public) interface to enable users and (external) systems to access and manipulate workflows and other data in the WMS—potentially through a *user interface* (C1). In the design of Fission Workflows, the apiserver is the core component in the architecture; even internal subsystems—such as the *controller* (C5)—have to use the apiserver to access or manipulate workflows. All apiserver commands that manipulate workflows are translated to events (see Section 5.3.2), which the apiserver appends to the *event store* (C3). Because executing functions affects a workflow and leads to new events, the apiserver also needs to be used to execute functions in one of the *runtime environments* (C7).

C3 The **event store** is the abstract data store responsible for persisting the events generated by the *apiserver* (C4) for workflow invocations. The persisting consists of linearizing and grouping append-only events based on the workflow invocation. The degree of persistence depends on the (configurable) event store implementation—from in-memory storage to persistence in SQL database. From these events the current state of workflows and workflow invocations are computed by and cached in *views* (C4).

C4 The **view** is the component responsible to abstract away the view-side the events, presenting users (such as the apiserver (C3) and the controller C5) with an easier-to-use current state of the WMS. It achieves this by turning the event sequences stored in the *event store* (C3) into the current state of workflows and workflow invocation; it sources the system state using the events. In the architecture the view component labeled explicitly as a cache to emphasize that, although it contains event-sourced state, it does contain vital state. If the view component crashes and recovers, no state is lost, because the view can rebuild its (cached) state from the events persisted in the *event store* (C3).

C5 The **controller** is the 'engine' in the workflow engine. It keeps track of current invocations, listens for events for these invocations, requests the *scheduler* (C6) extensively be done next for the workflow invocation, and executes these instructions of the scheduler for these workflow invocations. The instructions (e.g., start a task, abort the workflow invocation) typically require the controller to execute API actions of the *apiserver* (C2). As covered in the description of the apiserver (C2), these API actions typically lead to new events, which again trigger the controller—completing the control loop (see Chapter 2.3).

C6 The **scheduler** is needed to determine how to execute a workflow; which tasks need have to be executed when. The scheduler is a separate component from the controller to separate policy from mechanism, as has been common practice in WMSs [141] and other domains [43]. When the *controller* (C5) triggers the scheduler for a specific invocation, the scheduler evaluates the current state of invocation. Based on the current state, and optionally other sources of data (such as historical data), it creates a *scheduling plan*. This scheduling plan contains actions that need to be taken by the controller (C5), such as (re)invoking a function or aborting the workflow invocation all together. Chapter 6 further covers the scheduler design.

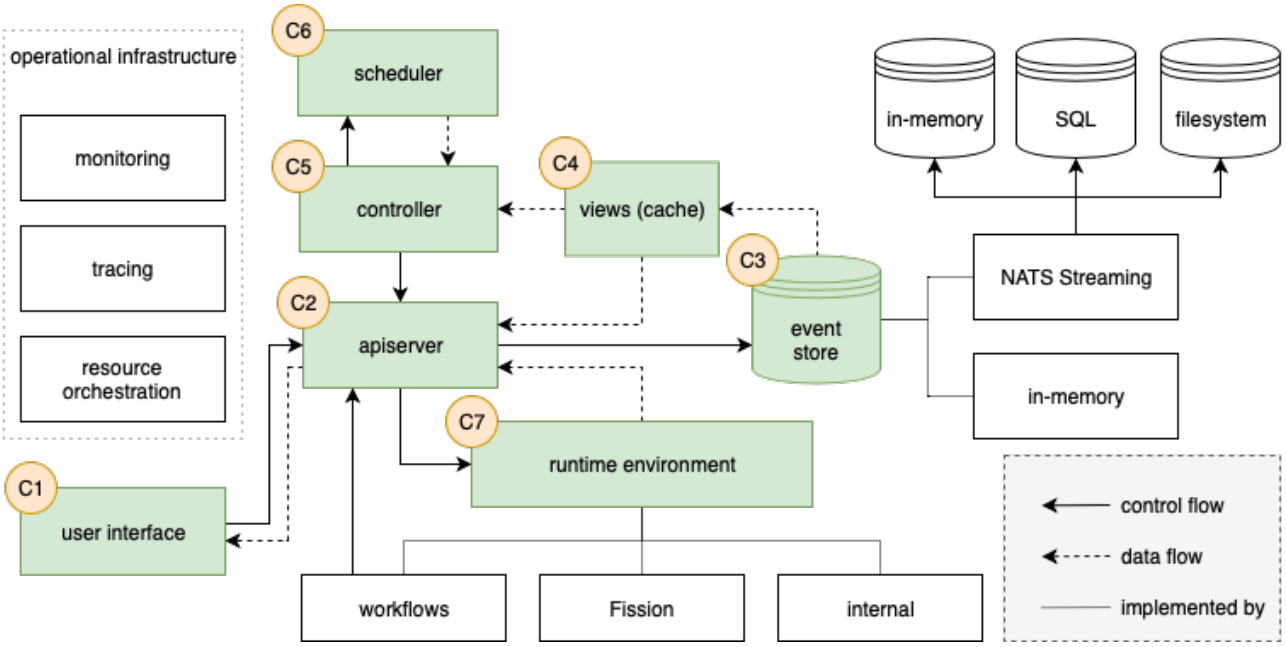


Figure 5.8: Overview of the Fission Workflows architecture. The numbered components core to the design of the WMS (highlighted in green) are discussed extensively in the text.

C7 The **runtime environment** is responsible for interacting with the function execution layer (see Section 2.1.4). Within the architecture, it is tasked with providing a abstract, consistent API for the different runtime environment implementations. Default implementations of the runtime environment are described further in Section 5.3.3.

In the remainder of this section we highlight five aspects key to the design of the workflow engine: resiliency measures (Section 5.3.1), configurable persistence of events (Section 5.3.2), FaaS platform integration (Section 5.3.3), the points of extensibility (Section 5.3.4), and scalability (Section 5.3.4).

5.3.1. Resiliency Measures

Users should rely on the WMS to correctly execute workflows. Yet, many external factors can influence the operation of a distributed system workflow engine, such as unavailable resources, unreliable networks, and service disruptions [127]. As explored in Section 2.3, control theory is a well-established approach to create reliable systems on top of these unreliable distributed resources.

The workflow engine for a large extend relies on control loops to improve the reliability. The controller in Figure 5.8 drives the workflow invocations forward using two approaches: event-driven and interval-based. In ideal situations—when networks are reliable, and resources available—the controller progresses workflow invocations event-driven; the controller triggers the execution of functions, which in turn lead to events that alter the state of workflow invocations, which in turn causes the controller to re-evaluate the invocation. This continues until a final state of the workflow invocation has been reached.

However, many things go wrong in this scheme: events can fail to arrive at the subscriber, a function might have become unresponsive, events might have failed to reach the controller, etc. To avoid these situations, the controller relies on interval-based checks. When no updates have been received for some time, on specific intervals the controller will re-evaluate the active workflow invocation, explicitly checking the caches, and data store for updates to the invocation. Depending on deployment-dependent policies (described in Section 7.2), the controller can decide on time-based actions, such as aborting the workflow after exceeding a specific upper-bound time.

5.3.2. Configurable Persistence of Events

An important responsibility of a workflow engine is to ensure reliability and persistence of data, even in the presence of faults [127]. The design of Fission Workflows addresses this in two ways: persisting

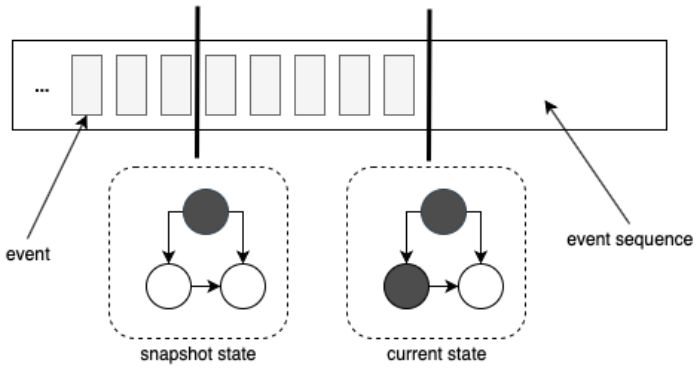


Figure 5.9: An example of an event-sourced workflow invocation. The events (grey boxes) are ordered from left to right, from old to new.

events (or operations) instead of state, and supporting multiple degrees of persistence.

Because execution and management of these remote functions is inherently event-driven, we exploit this property by modeling the workflow invocation models using *event-sourcing* (see P4 in Section 5.1), that is, an approach to state management where, instead of the current state itself, the operations (or *events*) to entities are stored [58]. Using the *event sequence* (or event log) we can (re)construct the state of entities by applying events in a chronological order. Figure 5.9 depicts an example of an event-sourced workflow invocation. Using the event sequence, the workflow engine can rebuild (or even rollback) its state to recover from errors, limiting the risk of permanent data corruption. This approach provides a further benefit to the system, by also allowing users to inspect the workflow invocations over time.

A disadvantage of using an event-sourcing approach is that it can become costly data storage-wise. Persisting all events over the lifespan of workflows—especially, large, and long-running workflows, can become more expensive than only storing the aggregate state of the workflow. There are several approaches to reduce the storage costs, such as snapshotting [52] or checkpointing [127] at specific points in time.

In this work we explore an alternative approach to reduce the data that is based on offering configurable persistence. The idea behind configurable persistence is based on the notion that not all workflow invocations equal: some workflow invocations are critical and need full persistence, whereas for other invocations often it does not matter if they get lost when faults occur. In the design of the WMS the user has extensive control over the persistence configuration: what kind of persistence model should be used (e.g., an in-memory, file-based, or SQL database)? is the WMS permitted to remove this invocation once it is completed? Etc.

This configurable persistence can not only reduce the data usage of the WMS, it allows users to make a trade-off between the degree of persistence and performance. As we explore in the evaluation (see Sections 8.2.2, and 8.2.3), the choice of event store implementation (and with that, the persistence model) impacts the fault tolerance and the performance of the workflow invocation.

5.3.3. FaaS Platform Integration

The workflow engine integrates with FaaS platforms using two interfaces: *runtime environments* and *resolvers*. The runtime environment is a consistent interface for FaaS platforms to integrate with. At the bare minimum, to be available as a runtime environment, the FaaS platform needs to support synchronous execution of a task. However, it can optionally support more functionality, such as prewarming of functions (see Chapter 6). This flexible interface allows FaaS platforms (and even non-FaaS systems) to be implemented as a runtime environment without the need to comply with a complex, potentially ill-fitting interface.

Besides the runtime, FaaS platforms need to implement a *resolver*. This component resolves *function references* to runtime-specific *function identifiers*, as described in the implementation of SWL (Section 7.1.2). Although in future work we envision the existence of FaaS-independent function resolvers, akin to service brokers and resource discovery mechanisms in cloud and grid computing [98], in the current design the resolver is part of a FaaS runtime environment.

The resolver and runtime environments are points of extensibility. They allow users to integrate with multiple FaaS platforms and providers. For example, for the prototype we focus on three distinct runtime environments (REs):

RE-1 Workflows runtime: To allow workflows to call other workflows, the workflows runtime acts as an identity function; it exposes the workflow engine as a resolver and runtime environment. This runtime is especially essential to support dynamic workflows, which rely on the execution of the dynamically-created workflows from the parent workflow.

RE-2 Internal runtime: Workflows often include simple functions—such as data selections and transformations, and other simple control flow manipulations. For such functions, the (network) overhead of converting the operation in to a full-fledged Fission function would dominate the runtime of the function. Thus, for these types of low-resource operations this runtime executes the functions within the workflow engine itself, using a lightweight FaaS implementation. This internal runtime also allows the workflow engine to come with a library of 'pre-supplied' functions—like AWS Step Functions and Azure Logic Apps have—without making functions 'built-in' or privileged.

RE-3 Fission runtime: Fission Workflows is designed to integrate with the Fission FaaS platform, by implementing Fission as a resolver and runtime environment. We further describe in the implementation (Section 7.1). The resolver resolves function references to Fission function identifiers, and the runtime converts task executions into HTTP requests to execute Fission functions using the HTTP interface. The HTTP response is converted back to the task output accepted by the Fission Workflows model.

5.3.4. Points of Extensibility

The UNIX principle of extensibility argues for explicitly extensible code that "aims to extend the lifespan and enhance the utility of the code the developer writes" [109]. For these reasons we explicitly define key points of extensibility in the design, where the user can augment or alter the default functionality of the system:

1. **Scheduler policies**: Scheduling is an inherently complex task, often involving various environment configurations, workloads, and (non-functional) requirements; a computationally hard problem for which schedulers frequently need to tuned to the specific workloads and use cases. For this reason, it is common practice in systems, including systems for workflow management, to provide options to alter or replace scheduler policies. As covered in Chapter 6, in this work we (by default) include three policies that implement this point of extensibility.

2. **Resolvers and runtime environments**: To keep a clear separation of concerns between the "workflow layer" and the "FaaS execution layer" (see the preliminary work on the FaaS reference architecture, in Section 2.1), the communication between the workflow engine and the function execution system is defined over a narrow interface. This allows users to integrate other function runtimes or even *types* of function runtimes (for example, integrations with the FaaS platforms of hosted clouds). The prototype contains three runtimes: Fission as the main FaaS, the workflow engine itself to allow workflows invoking (recursively) other workflows, and an internal runtime environment which executes lightweight functions locally within the workflow engine to minimize the impact of network latency.

3. **Internal functions**: In the internal runtime environment we avoid 'special functions': functions with privileged access to internal APIs. The internal functions have access to exactly the same functionality as regular, user-defined functions. This ensures that internal functions are also definable by users. Similarly, the internal functions provided with the system by default (see Chapter 7) utilize the same API as provided to the user.

4. **Event broker and storage**: Due to our design choice to use an event-sourcing approach to keep track of the state of the system entities, the choice of data broker (such as an event queue) and the persistent storage solution used to persist the brokered data, have a distinct impact on the performance. Depending on the specific workload, use case, and infrastructure, there exist alternative event broker and storage solutions that fit the system better. For example, in one situation a SQL database might be needed to provide full persistence, whereas in another situation the performance of an in-memory key-value store might is more desirable.

Component	category	Requirements R1	R2	R3	R4	R5	R6	R7	Constraints C1	C2	C3	C4
data model								✓		✓		✓
execution model	language			✓			✓		✓			
dynamic workflows		✓										✓
controller			✓	✓	✓							
scheduler				✓						✓	✓	
apiserver									✓			✓
runtime environment	system					✓					✓	
event store				✓	✓		✓	✓				
All	all	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 5.1: Analysis of Fission Workflows and the serverless workflow language (SWL) based on the requirements and constraints posed in Chapter 3.

5.3.5. Scalability

Although evaluating techniques and policies that can guarantee high scalability to our design is out of scope for this thesis, the design of the workflow engine does consider multi-node scalability.

In our design, a useful property to scaling the workflow engine itself is the fact that workflows executions are independent from each other. This allows us to shard the workload (of workflow invocations) over a set of workflow engines, using common load-balancing techniques and policies [108]. In turn, this has been a reliable method for the systems community to achieve good scalability with low management effort [21].

We design the components of the workflow engine to be deployed independently. Using event-sourcing (see Section 5.3.2) ensures that all components—other than the event store (C3 in Figure 5.3)—only have transient state. In other words, these components do not contain vital state, and can rebuild their state at any time by using the event store.

Although the event store is the only centralized, stateful component in the architecture, there are still some characteristics that make it more viable to scale than other regular databases. First, as addressed earlier, workflow invocations are independent, which eliminates the need to maintain strong consistency of events between workflow invocations—which allows for sharding of invocations (see [127]) . Second, the persistence model is append-only; events are only inserted, and never mutated. As explored by, among others, Beaver et al. [34], an append-only persistence model is generally more scalable than a persistence model that relies on state mutation.

5.4. Analysis

Table 5.1 provides an overview of the analysis of the workflow system and language aspects based on the constraints and requirements listed in the requirements analysis (Chapter 3). In Section 5.4.3, we evaluate the design of SWL based on the serverless workflow dimensions (see Section 4.5). Overall, we find the workflow system and the language design of Fission Workflows satisfy all constraints and requirements. In the remainder of this section we analyse how these constraints and requirements are met.

5.4.1. Requirements

The design meets the requirements (defined in Section 3.4) for a serverless function composition:

R1: Support complex control flow constructs. SWL explicitly allows for the expression of complex control flow constructs by relying on dynamic workflows. The ability to alter the control flow of the workflow by adding new tasks during the execution of workflows allows for the expression of arbitrary complex control flows.

R2: Support long-running processes. Within Fission Workflows we can rely on the controller to support long-running workflow invocations. It does not restrict the runtime for workflow invocation,

beyond limits that can be configured by the workflow engine operator.

R3: Minimize the composition overhead. The WMS relies on various functionality to minimize the performance overhead added to individual workflow invocations. In particular, the scheduler of Fission Workflows, focuses on being lightweight and avoiding the costs of creating full scheduling plans—opting instead for JIT scheduling. Another example of this focus on low overhead is the configurability of the persistence guarantees in the event store. By decreasing the persistence needed for events, users can reduce the overhead even further. Finally, the controller relies on subscriptions to events and high-frequency event loops to ensure that workflows are progressed as fast as possible, yet the overhead is only due to components that must receive event-updates.

R4: Reliable executions. The reliability of workflow invocations is also ensured by the controller and by the event-store implementation. The controller relies for reliability on a combination of event-sourced state and reconciliation loops to recover and complete workflow invocations after failures. The configurability of the event store also impacts the reliability: by opting for more strict persistence of events, the workflow engine increases the reliability of completing workflow executions.

R5: Support versioning and upgrade strategies. Though versioning upgrades are not explored in-depth in this thesis, versioning and further exploration in this space are already supported by the WMS through the resolvers in runtime environments. The explicit resolving of function references before workflow invocation, allows pluggable resolvers to decide which version of functions will be executed.

R6: Elastic and Scalable Executions. The language and the system both enable scalable workflow invocations. The workflow language deliberately avoids complex dependencies between workflows and workflows, ensuring workflow invocations that are independent. Thus, concurrent invocations can be scaled over multiple instances of the WMS. In the workflow system scalability is promoted by avoiding persisting state in all components, except for the event store. Similarly, the event store relies on the append-only nature of events (that further only need to be linearized from the perspective of the specific workflow invocation) to ensure scalability.

R7: Scalable in number of compositions. Besides being scalable in the number of workflow invocations, the system and language also take into account scalability in the number of unique workflow definitions. The language promotes this notion by separating the workflow definitions from their instances (i.e., workflow invocations). Similarly, the event store tracks workflows (i.e., workflow definitions) separately from workflow invocations, thus allowing different instances of the WMS to handle non-overlapping subsets of the workflows.

5.4.2. Constraints

Besides meeting the requirements, SWL and Fission Workflows satisfy all constraints posed in Section 3.5:

C1: Use HTTP as the interface to functions. The system and the language both ensure that the serverless workflows can adhere to the same interface of FaaS functions. The execution model of SWL enables this by constraining the behaviour of workflows to follow that of a serverless function: workflows are triggered by any event (e.g., HTTP requests), run until they succeed or fail, and can return a single output (which can be used as the HTTP response). Furthermore, in the WMS, the API server ensures that there is a consistent interface for controlling workflows, which supports the same operational patterns (see Section 2.1.4.2) as a common FaaS platform.

C2: Treat functions as black-boxes. The system and the language also meet the constraint of treat the FaaS functions as black-boxes. The data model does not require any configuration of FaaS functions beyond specifying the function reference and the payload. Nor does the workflow engine, in its FaaS platform-specific runtime environments, expect any introspection into the functions; it executes functions just as a any other FaaS user would.

C3: Only expect functionality provided by all FaaS platforms. The system is compatible with virtually all FaaS platforms, because it expects only a small API to be able to integrate with a FaaS platform. Although optional API features can be added, already any FaaS platform that allows for the synchronous or asynchronous execution of functions can be integrated with Fission Workflows. This also means that most other cloud services can be used as runtimes. This is exemplified by the implementation of an HTTP runtime environment, which allows for generic HTTP endpoints as tasks.

C4: Follow the serverless development workflow. The final constraint, is satisfied by both the language and system. SWL is designed as an executable workflow language—not as a complete workflow editor—to allow other, higher-level languages to be compiled/transformed to. Similar to FaaS functions, this allows for implementation of workflow definition approaches in various ways and languages, to which the SWL plays the role of virtual bottleneck of a conceptual hourglass. The API server also follows functionality similar to that of FaaS platforms, allowing developers to perform common operations (i.e., to create, list, delete, update, and test) their workflows similarly to developing FaaS functions.

5.4.3. Serverless Workflow Dimensions

Based on the requirements and constraints for a serverless function composition approach, in Section 4.5 we identified serverless workflows as a new type of workflows. In this section we analyze how the design of the serverless workflow management system and language handle the dimensions of serverless workflows.

D1: Minimal control over the task execution. Since we emphasised the deferring of task execution as a design principle (P2), both SWL and Fission Workflows require minimal control over the task execution in serverless workflows. SWL follows the serverless function model (see Section 2.1.2) for its tasks; it assumes that a task accepts data inputs, and returns either an output or an error. In the WMS, for the task execution we solely expect that we can execute the serverless function over HTTP, which is the case in virtually all FaaS platforms [135].

D2: Frequent, irregular executions. SWL and Fission Workflows anticipate frequent executions by separating the definition of a workflow from the execution. This separation allows the workflow executions to reference existing workflow definitions. The WMS can use frequent invocations to profile the workflow definitions—though comprehensive profiling is part of the future work.

D3: Deadline-aware. Although SWL supports setting a deadline on an invocation, this information can be used by the WMS to prioritize invocations with stricter deadlines—although we do not support this in the prototype.

Besides setting a deadline, the user has various options to prioritize the performance of the workflow over other characteristics. As described in Section 5.3.2, we can loosen the persistence model (e.g., only storing the events in-memory), to improve the performance. Or, we can use a specific runtime environment (See Section 5.3.3) that has a more favourable performance model.

D4: FaaS-like cost model. In the current design of the WMS, we do not have an explicit cost monitor to calculate the cost of a specific workflow invocation. However, there are several features that enable the addition of a cost monitor later: (1) invocations are individually tracked throughout the system; (2) all operations related to the invocation are stored as events; and, (3) components (including the controller and scheduler) log the time spend on invocations. An extension that implements cost monitor can aggregate this information to provide a cost-per-invocation.

D5: Unknown or high runtime variability of tasks. This dimension is not explicitly addressed in the current design. However, SWL nor the WMS make any assumptions regarding runtime variability. Similarly to D4, components expose information (e.g., task runtime) that can be used to profile the performance characteristics of tasks and runtime environments.

D6: Dynamic requirements. Dynamic requirements—requirements that change during a workflow invocation—are not explicitly addressed in the current design. However, the model does not impede dynamic requirements. The scheduler is stateless and evaluates workflow invocations solely on their current state. This means that a dynamically changed requirement of a workflow invocation will automatically be considered by the scheduler on the next evaluation.

D7: Multiple, independent workflow orchestrators. In both SWL and Fission Workflows, the workflow invocations are fully isolated from each other. Invocations cannot depend on other invocations in intricate ways or influence other invocations at runtime. This approach of independent workflow invocations enables the support for multiple workflow orchestrators—whether the orchestrator is from a third-party, or the orchestrator is another instance in a high-availability configuration (see Section 5.3.5).

D8: Dynamic structure. SWL supports workflows with a dynamic structure using the dynamic workflow extension (see Section 5.2.3). This extension, which enables tasks to output other workflows or tasks, supports the implementation of various dynamic control flow constructs, such as (un)bounded loops, conditional branches, etc.

D9: Task prioritization. Task prioritization is not part of the initial design of SWL or Fission Workflows. However, it does not appear that implementing support for task prioritization is impeded by the current design of neither the language or the system.

D10: Granular tasks. The WMS assumes granular tasks; tasks frequently running for only a second or less. Motivated by this dimension, the WMS utilizes a push-based communication model [127] for events (with a fallback to a pull-based communication model, in case of resource contention).

6

Design of an Architecture for Scheduling Serverless Workflows

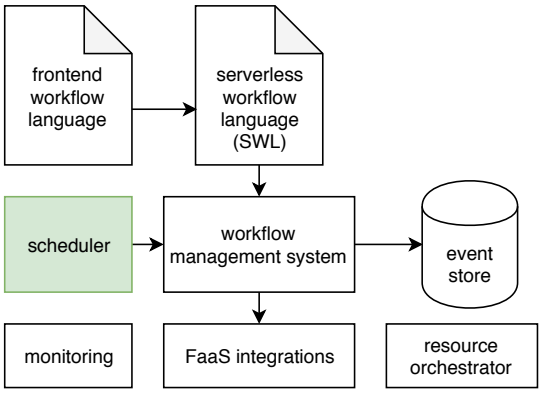


Figure 6.1: The aspect of Fission Workflows covered in this chapter: the design of the scheduler and the scheduling policies.

One of the main benefits of workflow-based function composition is the FaaS platform manages the operational logic automatically, on behalf of the user. By controlling the entire workflow, a workflow engine has a complete view of the state of both current and past workflow invocations. Thus, a workflow scheduler can use this knowledge, captured at runtime as operational data, to take decisions that improve performance and other non-functional aspects.

There exists a large corpus of related work on sophisticated scheduling policies in the related fields of autoscaling [68], and workflow scheduling [22]. Similarly to *serverless* workflow scheduling, these domain schedulers have a full view on the workflow, which allows them to implement sophisticated, data-driven scheduling policies. These include policies such as transferring data to a node, in the anticipation of the computation, to reduce or even avoid the delay of waiting for input data after the computation has started. However, the unique characteristics of serverless workflows (see Section 4.5) add both new opportunities and new constraints to scheduling.

The goal of this chapter is to do an initial exploration of serverless workflow scheduling. In Section 6.1, we introduce the different schedulers in the typical serverless architecture. We frame these schedulers as an abstract model, in which each scheduler is responsible for finding the optimal answer to a specific question. Based on the abstract model, we can now create a practical architecture for scheduling (in Section 6.2). To allow it to support a wide diversity of scheduling concerns, and in particular to decouple user- and system-level scheduling, we make the architecture *multi-level*. Then, we focus on the serverless workflow scheduler, with an exploration of the field of serverless scheduling policies, in Section 6.3 we define three basic scheduling policies: `horizon`, `prewarm-horizon`, and `prewarm-all`, which we implement in the Fission Workflows prototype (Section 6.4.1). Finally, to

Scheduler	Concern
Workflow scheduler	When to execute which function?
Function execution scheduler	On which function instance should the function be executed?
Function scheduler	Which configuration and how many resources to use for the function instance?
Resource orchestration scheduler	On which resources should the function instance be deployed?

Table 6.1: A hierarchical model to categorize schedulers based on the concerns they address. As we traverse schedulers top-down, the concerns go from high-level concerns, to more operational concerns.

analyze the proposed scheduling architecture, in Section 6.4 we map the Fission Workflows design to this scheduling architecture.

6.1. Abstract Model for Function Scheduling in FaaS

Scheduling in FaaS is an interesting topic, because it consists of multiple co-operating schedulers for deploying and executing the functions. As described in the background introduced in Section 2.1, in many cases the FaaS platform itself relies on an underlying resource orchestrator platform, such as Kubernetes, for the actual resource scheduling. Consequently, the function scheduler can defer resource management to the resource orchestrator, and thus focus solely on the function-specific aspects, such as which environment to run the function in and which parameters to use. Similarly, a workflow scheduler can focus on ensuring successful execution of the workflow, deferring the scheduling and the execution of tasks to a lower-level system (e.g., the Pegasus workflow engine defers task execution to the Condor DAGMan [50] or other similar systems).

In previous work [135], we have created a reference architecture for FaaS platforms (see Section 2.1.4). By extracting the scheduler components of this work we get an overview of the general scheduling architecture in a FaaS platform. Visualized in Figure 6.2, an approach to distinguish these schedulers conceptually and identify them in existing FaaS platforms is to consider which question each of these schedulers is responsible for answering. We have included in this reference architecture several scheduling components, one per layer. Together, they form an abstract scheduling model, whose details we present and analyze in the remainder of this section.

1. The **workflow scheduler** supports deadlines [22] and aims to meet timing constraints [115]; it also ensures the functional requirement that functions are executed only after their dependencies have completed successfully. In situations where resources are limited or there exist tasks with specific deadlines, the workflow scheduler faces more intricate decisions, and will have to decide which tasks to prioritize over others.

2. The **function scheduler** is concerned with mapping a function—a software artifact—to resources. This mapping is based on the requirements and constraints of the function, and on the types of available resources and services. Although trivial for simple functions, and for small, homogeneous setups, this multi-objective configuration problem quickly becomes complex and largely intractable to optimize online.

3. The **function execution scheduler** is responsible for mapping incoming (function-execution) requests to the function instances (see Section 2.1.4), akin to load balancing or scheduling in traditional web application architectures. This scheduler needs to make a decision based on two competing interests: the (non-functional) requirements of the request and the available function instances. In resource-constrained environments, the scheduler will have to decide how to prioritize the requested function executions.

4. The **resource (orchestration) scheduler** is responsible for mapping the functions and their resource demands to concrete resources. This level of scheduling, though highly relevant in the FaaS model, is already well-established and researched [43]. Although within the scope of this thesis we consider for this level a cluster resource scheduler, such as the Kubernetes scheduler, this level can also be expanded to an entire hierarchy of resource schedulers: from cluster-level to CPU-level to core-level.

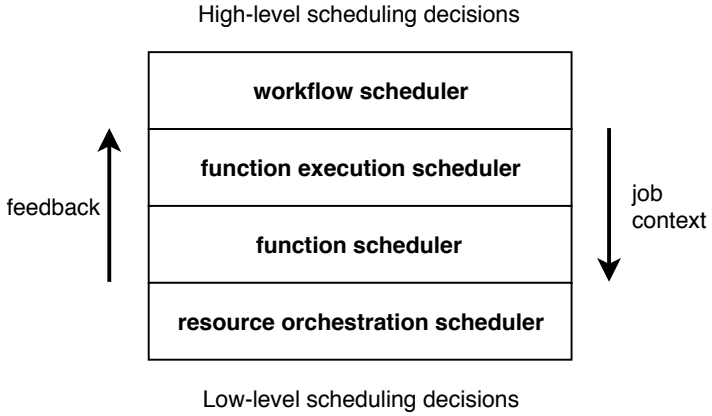


Figure 6.2: A multi-level scheduler architecture for serverless computing: contextual jobs flow down from higher-level schedulers (e.g., workflow scheduler) to lower-level schedulers (e.g., resource orchestration scheduler), which respond with feedback. As we traverse down the hierarchy of schedulers, the number of scheduling jobs increases, but the importance of the individual decisions decreases.

6.2. A Scheduling Architecture for Serverless Workflows

In this section we propose a scheduling architecture for the serverless scheduling model introduced in Section 6.1. The goal of this scheduling architecture is to from a framework and to propose terminology used to represent the interactions between the different schedulers in the model's hierarchy. The implication of this architecture is that in the remainder of this thesis, we can accurately define responsibilities inside and for the entire workflow scheduler in Fission Workflows.

Figure 6.2 depicts the architecture. This multi-level scheduling architecture is comprised of a hierarchy of schedulers involved in the process of executing serverless workflows. These schedulers communicate explicitly and implicitly using contextual jobs and feedback: schedulers submit jobs to lower-level schedulers, and provide feedback to higher-level schedulers on the impact of their decisions. By design, this architecture allows an arbitrary number of schedulers at each level.

To trigger a scheduler, a user submits a **job**; a data structure describing what the job submitter wants to be performed. The job is specific to the scheduler executing it, and thus may represent different abstractions depending on the level of the hierarchy where it is executed. For example, a function scheduler takes in as a job a function request, whereas the resource scheduler takes in requests to deploy specific VM images.

Along with the job, a user can submit **context** to a scheduler. Whereas the job has a strict defined structure, the context is more a more loosely defined data-structure. This optional context can contain all metadata related to the job, including correlation identifiers, timestamps, and user information. Schedulers are also free to add arbitrary metadata to the context they pass to the next scheduler. This allows schedulers to provide optional data to lower-order schedulers, which could use this information for their scheduling process, yet remain only loosely coupled to the higher-level schedulers. From a design perspective, the context allows higher-order schedulers to provide *hints* to lower-order schedulers—which can choose to use or ignore them.

Following the scheduling (and execution) of jobs, **feedback** flows back up the hierarchy of schedulers. In general, this feedback is implicit. The duration of the job submission, job completion, and potential errors are all implicit feedback. If systems support it, schedulers could also optionally choose to return explicit feedback. This could include scheduling metadata, system information, or additional metadata about the submitted job. As in the case of the job context, we design the feedback mechanism to be both flexible and non-mandatory, that is, the sending side can decide to add information and the receiving side can choose to ignore it.

A number of lower-level schedulers typically exist in the operational hierarchy that enables serverless tasks to execute, such as hypervisors or processor-level (hardware) schedulers. However, we consider these to be out of scope for this work; as is common in (large-scale) distributed systems, we consider the resource scheduler coordinating all processors inside a physical machine to be the lowest-level scheduler considered for inclusion in the resource management and scheduling system.

Finally, although the model does not show it, the systems will need to find ways to propagate the

context, along with the job, to underlying systems. The schedulers—being decoupled from each other—do not have the option to directly pass data about jobs and contexts to each other. Because the hierarchy of systems treats each system as a black-box, schedulers need to submit their jobs through the same process as any other process or user. Although this helps maintain clear decoupling between the layers—and it adheres to the principle of consistency [37]—it depends on the system if and how it can propagate the context. To mitigate this, the system integrator could choose to adapt the system or find ways to propagate the context through non-supporting systems (for example, by merging the context data into the data structure for the job itself).

As identified in the scheduling model (Section 6.1), the workflow scheduler is responsible for answer a single but possibly complex question: *when to execute which function?* All other concerns are deferred to lower-level schedulers in the scheduling architecture (Section 6.2).

In Fission Workflows, the scheduling of the workflows is divided up between three conceptual components:

1. The **controller** is responsible for checking workflow invariants, such as whether a workflow is valid or it has completed. By checking the invariants before querying the scheduler we avoid the overhead introduced by the scheduler, and guarantee the scheduler that the workflow is schedulable.
2. The **scheduler** provides a consistent interface over the scheduling policies.
3. The **scheduler policy** is the pluggable part of the scheduler, which implements the strategy that the scheduler takes towards answering the question of when to execute which function.

6.3. Scheduling Policies: `horizon`, `prewarm-horizon`, and `prewarm-all`

The execution model and the centralization of operational logic in the infrastructure opens new possibilities for more sophisticated, policy-based or even more complex scheduling. In this section, we define three scheduling policies for serverless workflow scheduling, `horizon`, `prewarm-horizon`, and `prewarm-all`. The investigation of other directions in scheduling policies falls outside the scope of this thesis and is included in the directions for future work in Section 9.2.

6.3.1. Prewarming Cold Functions

When a function, that needs to be executed, still needs to be deployed, the caller—in this case the WMS—will incur an additional delay to await the deployment of the function instance. This is the problem of *cold starts*. Previous work [133] has indicated that the cold start problem (see Section 2.1.4.2) is one of the key performance issues in the FaaS field, and in the encapsulating domain of serverless computing.

The main idea for scheduling serverless workflows explored in this work is that of *prewarming*: by prematurely deploying a function before the resources needed for executing the function are actually ready, the function execution time is solely determined by the running time. With the extended knowledge about the a serverless workflow, the scheduler, given correct policies, should be able to determine ahead of time when to expect function executions. In turn, this would allow the workflow engine and the underlying FaaS platforms and resource orchestrators to take all possible steps of preparation surrounding the function execution, and thus to prewarm the function.

The notion of prewarming is not novel. In various domains schedulers attempt to perform optimizations by preparing components for future executions. For example, in CPU scheduling, the CPU moves instructions and data from slow memory to faster internal memory just before the instruction is required—a process called *prefetching* [121, 126]. However, this work is, to the best of our knowledge, the first time prewarming is used explicitly for serverless *workflow* management—each of the policies we explain in the remainder of this part represents an alternative to driving the prewarming mechanism.

In the FaaS field several optimizations in the function scheduling have been devised to improve the cold start issue [130]. In many FaaS platforms, including OpenWhisk, OpenFaaS, and Fission, after the execution of a function, the function itself is kept alive for another couple of minutes before being cleaned up (garbage-collected). This reduces the cold start of executions in a series of function executions to only be present in the initial execution. Fission maintains a pool of generic ‘environments’

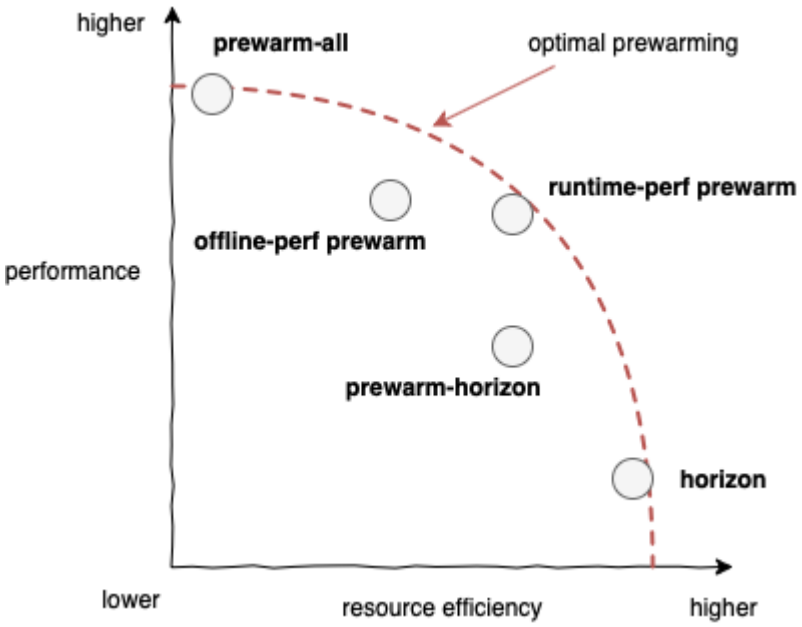


Figure 6.3: Visualization of the performance-cost trade-off achieved hypothetically by the design-space of prewarming policies—higher is better. The red, dashed curve indicates optimal policies, which provide most performance for a given resource efficiency. The locations of the scheduling policies in this trade-off are indicative. See the text for an explanation of each policy, and of the continuum.

(akin to the common practice of thread pooling on machines), which already have completed the generic part of the deployment process including aspects such as container deployment, network configuration, and HTTP server startup. When a new function is about to start its execution, the function source code gets injected into one of the generic environments (a process called specialization), reducing the cold-start time.

In the `prewarm-horizon` policy the main challenge is *when to prewarm functions to optimize the performance-cost trade-off?* By prewarming a function too late, the subsequent function-execution will still be delayed, because the warm-up process of the function has not completed yet, without delay for the start of the function. By prewarming a function too soon, the idle awaiting function waste unnecessary resources—making the execution of the function, and thereby the workflow, more expensive than necessary.

Figure 6.3 depicts this trade-off as a hypothetical continuous scenario, with various variations on the possible prewarm policy classified within the bounds of this trade-off. Theoretically, there exist scheduling policies that optimize this trade-off; they provide most performance for a given resource efficiency. However, in practice, the trade-off has been relatively unexplored—an exploration that is part of our future work.

Based on scheduling policies of related fields (such as autoscaling [44]), there are several possible prewarm policies. The `horizon` policy is most efficient in terms of resource usage; by not performing any prewarming fewer idle function instances will remain, awaiting executions. At the top, the `prewarm-all` policy will warm-up all functions in the workflow, avoiding the risk of cold-starts at the cost of (severely) increased resource usage for idle function instances. In between these two extremes of the design-space for prewarming policies, `prewarm-horizon` is a policy that warms up functions that solely depend on the functions in the current scheduling horizon (functions that are being scheduled and executed). We further populate the design space with policies that use a secondary mechanism to complement the capabilities of prewarming. The `offline-profiled prewarm` policy determines, based on offline profiling or assumptions about the function start- and run-times, at what points in time the functions need to be prewarmed to minimize the runtime and the resource usage. In contrast to this policy, the `runtime-profiled prewarm` policy performs the same calculation to determine when to prewarm functions, but uses runtime profiling of the functions instead, which we expect will make it more accurate and thus lead to a better performance-cost trade-off than its offline counterpart.

Input: AllTasks, CompletedTasks
Output: ExecTasks

```
1  OpenTasks ← AllTasks - CompletedTasks  ExecTasks ← [ ]
2  for t in OpenTasks do
3  |  h ← true  for d in t.dependencies do
4  |  |  if d in OpenTasks then
5  |  |  |  h ← false  break
6  |  |  end
7  |  end
8  |  if h is true then
9  |  |  ExecTasks.append(t)
10 |  end
11 end
12 return ExecTasks
```

algorithm 1: The `horizon` scheduling policy: based on the current progress (`CompletedTasks`), the scheduler determines the `OpenTasks`, and schedules the tasks that have all their dependencies met to be executed.

For the sake of the scope of this thesis, we fully define, and evaluate through real-world experiments, only three of the policies described before: `horizon`, `prewarm-horizon`, and `prewarm-all`. For each policy we describe the intuition behind it, describe the algorithm, and explain the policy using the example workflow depicted in Figure 4.1.

6.3.2. The `horizon` Policy

horizon is the scheduling policy that schedules the tasks on the scheduling horizon for execution. It does not prewarm any task.

The `horizon` policy is the least complex, and default, scheduling policy available in Fission Workflows. The policy can be described as a Just-In-Time (JIT) policy; it solely schedules the tasks that have all their dependencies met at the time when the scheduling decision takes place, rather than trying to schedule other tasks ahead of time. Despite the simplicity of the `horizon` policy, there is ample motivation to investigate it. First, it can be used as a baseline to compare other policies with. Second, since, to the best of our knowledge, no WMSs support prewarming of serverless functions, this also means that the state-of-the-art effectively uses this policy.

The pseudocode of the `horizon` policy is listed in Algorithm 1. The scheduler expects two inputs: `AllTasks`, a collection of all tasks in the workflow; and, `CompletedTasks`, a collection of the completed tasks in the workflow. The algorithm starts, on Line 1, with calculating the current open tasks—tasks that have not been executed yet. In Line 3 to 14 we loop through these open tasks. Within this loop, we check for each of the open tasks, whether it only depends on tasks that have completed (see Lines 4 to 10). If the task only depends on tasks that have completed, it is added to the `ExecTasks` collection on Line 12. This results in a collection of tasks to execute in `ExecTasks`.

Figure 6.4 depicts an example execution of the 3-task workflow using the `horizon` policy. When the workflow is invoked, the workflow is loaded, and the `horizon`-based scheduler evaluates the workflow for the first time. The scheduler will request and await the execution of task A in step 2. Once task A completes in step 3, the scheduler will re-evaluate the workflow, and execute task B in step 4. In turn, once task B completes in step 5, the scheduler will re-evaluate the workflow, and execute task C in step 6. Finally, the scheduler will decide that the workflow is completed once task C is completed.

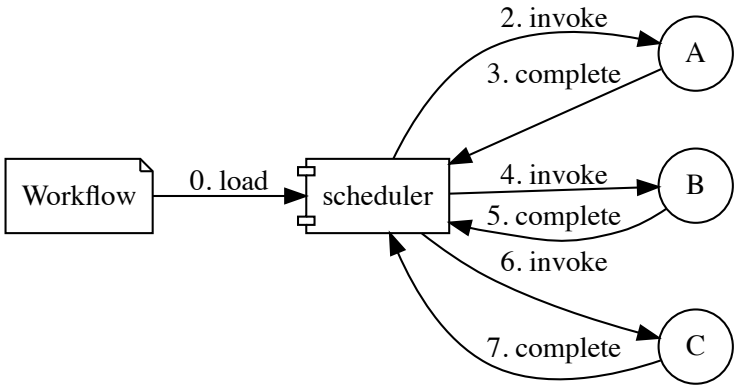


Figure 6.4: Example of a scheduler with a horizon policy. Given a chained composition of functions, $A \rightarrow B \rightarrow C$, the scheduler invokes a function (starting with function A) and continues to the next once the dependency has completed.

6.3.3. The prewarm-all Policy

prewarm-all is the scheduling policy that schedules the tasks on the scheduling horizon for execution, and it will prewarm all functions of the workflow on every scheduler evaluation.

The prewarm-all policy is one most aggressive prewarming scheduling policies. The idea behind it is to, on every scheduler evaluation, prewarm all open tasks that are not scheduled to be executed. Effectively, when using this policy the scheduler is optimizing for performance, taking the lower resource efficiency for granted. This policy works under the assumption that prewarming is idempotent—prewarming an already prewarmed function does not impact the system.

The pseudocode of the prewarm-all policy is listed in Algorithm 2. The scheduler expects two inputs: AllTasks, a collection of all tasks in the workflow; and, CompletedTasks, a collection of the completed tasks in the workflow. Similar to the horizon policy, the algorithm starts, on Line 1, with calculating the current open tasks—tasks that have not been executed yet. On Line 3 to 14 we loop through these open tasks. Within this loop, we check for each of the open tasks, whether it only depends on tasks that have completed (see Lines 4 to 10). If the task only depends on tasks that have completed, it is added to the ExecTasks collection on Line 12. Then, on Line 15 we calculate all open tasks that are not marked for task execution, and add these to PrewarmTasks. The result of this algorithm is a set of tasks to execute (ExecTasks) and a set of tasks to prewarm (PrewarmTasks).

Figure 6.5 depicts an example execution of the 3-task workflow using the prewarm-all policy. When the workflow is invoked, the scheduler evaluates the workflow for the first time. In step 2, the scheduler decides to execute task A. At the same time, it will also prewarm both task B and C. Once task A completes, the scheduler will re-evaluate, and decide to execute task B. Again, in step 4, it will prewarm all other open tasks—in this case just task C. In step 5, task B completes and triggers a re-evaluation. The scheduler will invoke the last task C in step 6, and complete the workflow after the task completes in step 7.

6.3.4. The prewarm-horizon Policy

prewarm-horizon is the scheduling policy which aims to prewarm all the functions that are *up next*, that is, the functions that willbe executed after the active tasks have completed.

The prewarm-horizon policy (with the pseudocode listed in Algorithm 3) aims to provide a middle-

Input: AllTasks, CompletedTasks
Output: ExecTasks, PrewarmTasks

```
1  OpenTasks ← AllTasks - CompletedTasks  ExecTasks ← [ ]
   // Execute tasks on the scheduling horizon
2  for t in OpenTasks do
3  |  h ← true  for d in t.dependencies do
4  |  |  if d in OpenTasks then
5  |  |  |  h ← false  break
6  |  |  end
7  |  end
8  |  if h is true then
9  |  |  ExecTasks.append(t)
10 |  end
11 end
   // Prewarm all open tasks not on horizon
12 PrewarmTasks ← OpenTasks - tasks
13 return (ExecTasks, PrewarmTasks)
```

algorithm 2: The `prewarm-all` scheduling policy: based on the current progress (`CompletedTasks`), the scheduler determines the `OpenTasks`, schedules the tasks that have all their dependencies met to be executed (`ExecTasks`), and prewarms all other open tasks (`PrewarmTasks`).

ground between the two previous policies. On every scheduler evaluation it will attempt to prewarm all the functions that are *up next*, that is, the functions that are expected to be executed after the current functions have completed. The example in Figure 6.6 highlights the difference between this and the `prewarm-all` policy; in this case the third task C in the pipeline is only prewarmed once task B is being executed.

The pseudocode of the `prewarm-horizon` policy is listed in Algorithm 3. The scheduler expects two inputs: `AllTasks`, a collection of all tasks in the workflow; and, `CompletedTasks`, a collection of the completed tasks in the workflow. Similar to the `horizon` policy, the algorithm starts, on Line 1, with calculating the current open tasks—tasks that have not been executed yet. On Line 3 to 14 we loop through these open tasks. Within this loop, we check for each of the open tasks, whether it only depends on tasks that have completed (see Lines 4 to 10). If the task only depends on tasks that have completed, it is added to the `ExecTasks` collection on Line 12. Then, on Line 15 we calculate all open tasks that are not part of the executing tasks, `NonExecOpenTasks`. On Lines 17 to 28 we loop through the `NonExecOpenTasks`, and check for each if the task depends solely on tasks that are either on completed or are part of `ExecTasks`. If this is the case, the task is added to `PrewarmTasks`. The result of this algorithm is a set of tasks to execute (`ExecTasks`) and a set of tasks to prewarm (`PrewarmTasks`).

Figure 6.6 depicts an example execution of the 3-task workflow using the `prewarm-horizon` policy. At step 1 the workflow is started and triggers the scheduler to evaluate the workflow. The scheduler decides to, in step 2, invoke task A. Additionally, it prewarms task B, because it will be executed right after task A completes. Once task A completes, the scheduler re-evaluates the workflow. The scheduler decides to execute task B and prewarm task C in step 4. In step 5, task B completes. The scheduler again evaluates the workflow, and decides to execute task C. In step 7, task C completes and the WMS marks the workflow as completed.

6.4. Analysis

In this section we reflect back on the requirements posed in the requirements analysis (see Chapter 3). We verify the serverless scheduling model by mapping the Fission + Fission Workflows stack to the model.

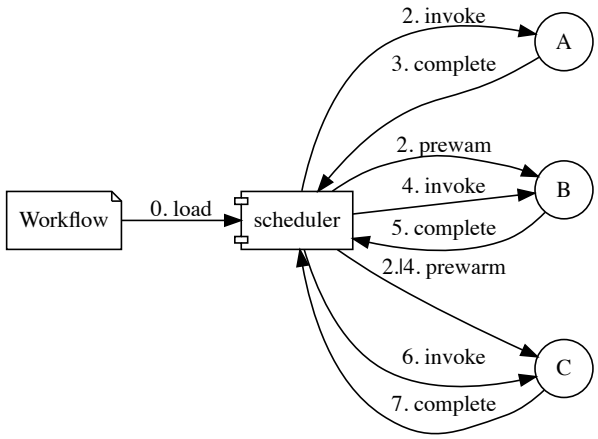


Figure 6.5: Example of a scheduler with a `prewarm-all` policy: given a chained composition of $A \rightarrow B \rightarrow C$, the scheduler invokes a task (starting with task A) and prewarming the next task in the chain (starting with task B), continues until all tasks have completed.

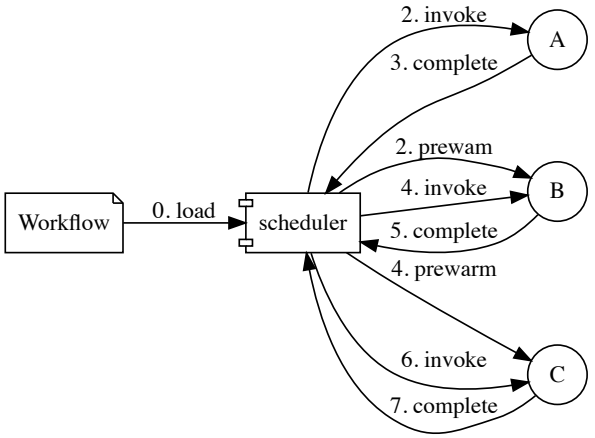


Figure 6.6: Example of a scheduler with a `prewarm-horizon` policy. Given a chained composition of functions, $A \rightarrow B \rightarrow C$, the scheduler invokes a function (starting with function A) and prewarms the next function in the chain (starting with function B), continues until all functions have completed.

Input: AllTasks, CompletedTasks
Output: ExecTasks, PrewarmTasks

```
1  OpenTasks ← AllTasks - CompletedTasks  ExecTasks ← [ ]
   // Execute tasks on the scheduling horizon
2  for t in OpenTasks do
3  |  h ← true  for d in t.dependencies do
4  |  |  if d in OpenTasks then
5  |  |  |  h ← false  break
6  |  |  end
7  |  end
8  |  if h is true then
9  |  |  ExecTasks.append(t)
10 |  end
11 end
   // Prewarm open tasks on the prewarm horizon
12 NonExecOpenTasks ← OpenTasks - ExecTasks  PrewarmTasks ← [ ]  for Pt in NonExecOpenTasks
    do
13 |  Ph ← true  for Pd in Pt.dependencies do
14 |  |  if Pd not in ExecTasks and Pd not in CompletedTasks then
15 |  |  |  Ph ← false  break
16 |  |  end
17 |  end
18 |  if Ph is true then
19 |  |  PrewarmTasks.append(Pt)
20 |  end
21 end
22 return (ExecTasks, PrewarmTasks)
```

algorithm 3: The `prewarm-horizon` scheduling policy: based on the current progress (`CompletedTasks`), the scheduler determines the `OpenTasks`, schedules the tasks that have all their dependencies met to be executed (`ExecTasks`), and prewarms all open tasks that will be executed once the current functions complete (`PrewarmTasks`).

6.4.1. Mapping of Fission and Fission Workflows to the Scheduling Model

To evaluate the scheduling model we map Fission Workflows to the multi-level scheduling architecture described in Section 6.2. As shown in the mapping in Figure 6.2, at the lowest level of scheduling the Kubernetes resource scheduler takes care of deciding where to place the function instances. It achieves this through its *kube-scheduler* component. In the level above that, the Fission function scheduler decides if and how to deploy the function source code to the deployable artifact. The component in Fission responsible for this is called the *executor* [1]). In the next layer, the Fission execution scheduler decides how to map the execution request to the right function, triggering the function scheduler if no function exists. It achieves this through its `router` component. At the highest level the *workflow scheduler* in Fission Workflows, based on the state of a workflow invocation, decides which functions need to be executed.

Implementing the `horizon` policy is straightforward. It only requires a working Fission Workflows on a Fission on a Kubernetes deployment. By default, the Fission Workflows prototype employs this `horizon` policy when scheduling [2].

Implementing the `prewarm-horizon` and `prewarm-all` policies require the FaaS platform to support prewarming functions. We augmented Fission to support prewarming, by exposing an originally hidden API call that requests the deployment of a function. Following the notion of optional hints, the controller attempts to prewarm the functions, but ignores these scheduling instructions if the FaaS

[1] `https://github.com/fission/fission/tree/master/pkg/executor`
[2] `https://github.com/fission/fission-workflows/blob/master/cmd/fission-workflows-bundle/main.go`

	Kubernetes	Fission	Fission Workflows
Workflow Scheduler			scheduler
Function Execution Scheduler		router	
Function Scheduler		executor	
Resource Scheduler	kube-scheduler		

Table 6.2: Mapping of the Fission and Fission Workflows scheduling-related components to the abstract scheduling model of Section 6.1, and thus to our design of a multi-level scheduling architecture in Section 6.2.

platform does not support prewarming, as is the case with the HTTP function runtime environment in Fission Workflows. Unfortunately, major FaaS platforms, such as AWS Lambda, Google Cloud Functions, and Azure functions, do not support prewarming yet.

6.4.2. Limitations

Given the scope of this thesis, in this work we limited the exploration of prewarming-related policies solely to three prewarming-based scheduling policies: `horizon`, `prewarm-horizon`, and `prewarm-all`. The choice for these prewarming-related policies is that, although they give a substantial improvement from function executions, these policies does not require any profiling or data models to be in place. In our experience, this lack of a need for existing data and predictability, also makes it a policy that engineers would appreciate and be more likely to adopt to implement in production-level deployments. Offline and runtime-profiled prewarm policies seem promising directions, but given the lack of representative serverless workflows and the lack of predictability of these policies, motivated by the scientific method we can only leave them for future work.

The variations of the warm-up policies discussed so far ignore the potentially dynamic structure of the serverless workflows. Without addressing this limitation, the warm-up policy will not evaluate dynamic tasks (tasks outputted by other tasks), because these are not present in the static workflow that the policies discussed so far use as input to make their decisions. This will lead to dynamic workflows being scheduled less accurately than static, creating the undesirable incentive for users to avoid dynamisms in their workflows as much as possible. We identify two approaches to mitigate this performance issue: workflow hints and runtime profiling. A mechanism could be implemented to allow tasks to provide hints to the workflow engine of what to expect in the workflow—which tasks, what characteristics for each task or in general, etc. For example, a task statically outputting another task, or a conditional task outputting either of two tasks, could provide a hint to the workflow engine that they will or might output a certain task. Another tool we could use is runtime profiling. The likelihood of serverless workflows, to be frequently executed, allows the workflow engine to collect relatively quickly many data points on the runtime of tasks and the probability of each task to output other tasks, dynamically. Similarly to branch prediction in CPUs [121] and other forms of optimistic caching [25], our serverless workflow scheduler could use this data to predict the need to prewarm dynamic tasks.

7

Implementation of the Fission Workflows *Product*

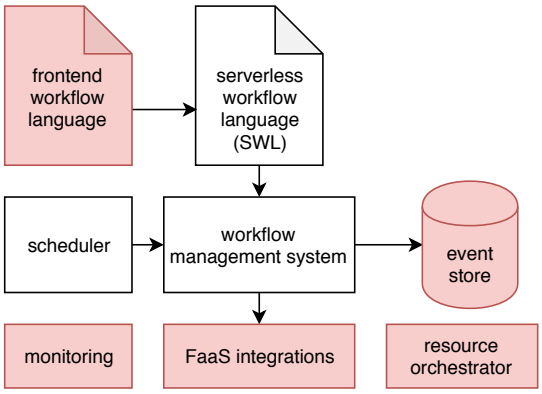


Figure 7.1: The aspects of Fission Workflows covered in this chapter: the SWL-YAML reference implementation of SWL (Section 7.1.2), the event store implementations (Section 7.1.3.4), the FaaS runtime implementations (Section 7.1.1), and the other operational concerns, such as monitoring and resource management (Section 7.1.3).

This chapter addresses the implementation of both the Fission Workflows *prototype* and *product*. The systems community has long agreed that implementations are useful to test a theoretical claim that a particular design is fit for its purpose. The implementation helps reveal real-world issues with the design, and helps convince industry of the practical usefulness of the proposed design. The implementation is commonly a prototype, a minimal, functional implementation of the proposed design to use in experimentation. Based on these established practices, we implemented the Fission Workflows *prototype*.

However, there are limitations to the benefits of relying on the prototype to validate a design. Prototypes are implemented for experimentation—to validate whether the design *can* work in practice. The prototype does not reveal real-world issues outside of its specific, target use case or user. Nor, does it reveal issues that can occur from long-term, production usage. Most importantly, given that the prototype is generally not implemented to be adopted by multiple, real-world users for production use cases, it does not reveal whether the design solves the target problem satisfactory—whether real-world users perceive the design to be valuable enough to be willing to pay the inevitable costs (such as, integration, adoption, and operational costs) to use it.

To overcome these limitations, we choose to implement the Fission Workflows *product* after the implementation of the initial prototype. The key difference between the prototype and product is the implementation of all concerns surrounding the functional core of the design. Whereas the prototype focuses on implementing the key functionality of the design, the product concerns itself with making this functional core performant, stable, usable, monitorable, and understandable. This requires implementing (user) interfaces, non-essential features, extensive tests, support tools, and integrations

with other existing systems. Moreover, besides the software, the product needs extensive, well-written documentation, understandable examples, and effective governance and support processes.

By implementing and releasing the product, real-world users were able to adopt and evaluate Fission Workflows for their use cases. To foreshadow the remainder of this chapter, based on the initial interest of users in the product we believe that the design of a serverless WMS helps solve the problem of serverless function composition. Moreover, this user interest in turn lead to valuable feedback from both potential users and actual users—which helped shape next iterations of the product, as well as the future roadmap.

Both the prototype and product of the serverless workflow engine were implemented in collaboration with Platform9[1], a company which has been a major stakeholder in the serverless computing ecosystem as the founder and main contributor of the Fission FaaS platform. Within the scope of this work, Platform9—besides providing logistical support—participated in design discussions, led the marketing efforts, and performed code reviews. The majority of the design, implementation, and evaluation of Fission Workflows was done by the author.

In the remainder of this chapter we provide an overview of the progression, from a prototype to a full-fledged product. We describe the implementation of the initial prototype in Section 7.1, which includes the reasoning behind the choices made for various backing technology. This is followed by a description, in Section 7.2, of the process; from the initial implementation of the prototype towards the Fission Workflows product. We conclude with a reflection on this process highlighted in a number of lessons learned in Section 7.4.

7.1. Implementation of the Prototype

Although not commonly acknowledged, implementation requires many important (design) choices, not architectural or policy-oriented as in Chapters 5 and 6, respectively, but to the realization of the working system and with high-impact especially to its non-functional properties. Therefore, in this section we describe the implementation of the Fission Workflows prototype, covering the design choices made for the workflow engine. We discuss the relevant implementation decisions made in relation to the Fission FaaS platform in Section 7.1.1, the high-level decisions made for the workflow language in Section 7.1.2, and details about the implementation of the prototype in Section 7.1.3.

7.1.1. FaaS Platform: Fission

As we determined in the survey of the state-of-the-art FaaS platforms (Section 4.1), Fission is an appropriate FaaS platform to base our prototype on. Although Fission was comparatively the best option, we still needed to make changes to the platform. Most notably, we made changes to Fission to (1) allow workflows to conform with the Fission function API, (2) automate the parsing of workflows, and (3) become able to pass Fission metadata with each function invocation.

A key feature of the workflow engine is that workflows should conform to the same interface as regular Fission functions. In other words, the user should be able to invoke a workflow without the need to use invocation semantics different from invoking any other Fission function. As shown in Figure 7.2, to ensure, or even to enforce, this requirement for a common API for both functions and workflows, we added a proxy to the workflow engine that made it conform to the exact Fission environment. However, as internally there are differences between regular environments and this workflow environment—in contrast to regular environments which can each only execute a single function, workflow invocations are multiplexed onto a single workflow engine, that must support the multiple functions in the workflow and their dependencies—we needed to extend Fission. To resolve this, we added new parameters to Fission to allow users to specify how environments should be handled: Can it reuse the same environment for multiple functions? Should it clean up the environment if it has not received any invocations recently? Etc.

Fission Workflows distinguishes between the format used by users to define workflows and the machine-format that it uses internally. Somewhere along the development process, the user-level format needs to be converted to the machine-level format. In the initial version of the workflow engine, this conversion had to be done by the users themselves. This additional, easy-to-forget, step caused confusion among users, and thus threatened to limit the broad adoption of the Fission product. After Fission introduced builders—a feature that allows modifying functions uploaded by users before they

[1] `https://platform9.com`

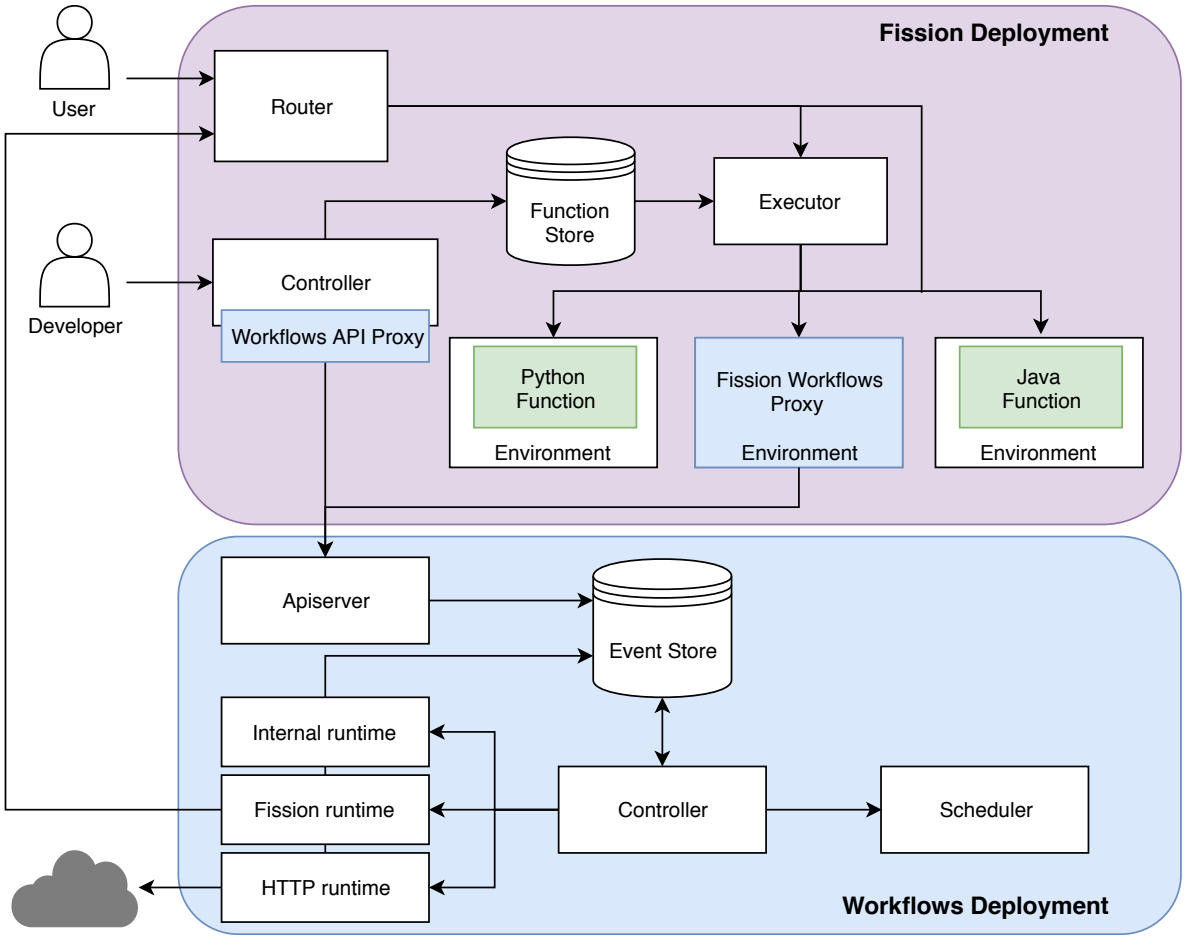


Figure 7.2: A deployment diagram highlighting how Fission Workflows (bottom) integrates with Fission (top). Similar to other user-defined (green) functions, the workflow engine (blue) is exposed in Fission as an environment, and proxies the requests to the workflow engine deployment. The remainder of the workflow engine API is exposed over a proxy in the Fission controller.

become deployable—the workflow parsing was moved into a builder, removing this step from the set of user concerns, while keeping the parsing of workflows an extensible process.

Although a FaaS platform should not expose too many internal details to the functions—to avoid tempting the users to create functions that depend intricately on internal details—the workflow engine did require to have a way to determine which Fission function maps to which workflow. To resolve this issue, we added Fission metadata headers to each function execution, which included the unique Fission function identifier. This allowed the workflow engine to consistently map these unique Fission function identifiers to internal Workflow identifiers.

There were multiple alternative approaches to integrate Fission and Fission workflows that we considered and discarded. First, we considered fully developing the workflow support into Fission, which would add workflows as a feature in Fission. We did not pursue this approach because it would make it impossible to generalize the serverless workflow engine beyond Fission. Furthermore, complete integration between the WMS and the FaaS would make it more difficult to keep the logical and technical separation of the Function Management Layer and the Workflow Orchestration Layer of the FaaS reference architecture described in Section 2.1.4. Second, we considered a more tighter integration Fission and Fission Workflows, in which we would bypass the need for a *proxy environment* and execute workflows directly from the Fission router (see Figure 7.2). Similarly, instead of implementing the workflows CLI as a separate CLI, it would be integrated directly into the Fission CLI. Although this approach would have likely simplified the integration effort, it would require modifying the Fission core components to add support for Fission Workflows. In contrast, the current approach to add support for Fission Workflows does not *require* modifying Fission components—which emphasises the portability of Fission Workflows to utilize other (potentially closed-source) FaaS platforms.

7.1.2. SWL-YAML: a Reference Implementation of SWL

In Section 4.4 we analyzed the current, popular workflow languages and found that none of these existing workflow languages fit the requirements (Chapter 3) of serverless workflows. Based on this observation, in the design of the workflow management system we proposed SWL (Section 5.2), a new (abstract) workflow execution and data model specifically targeting the serverless workflow domain. In this section, we summarize the design decisions made during the implementation of a concrete workflow language based on the SWL. A full description and specification can be found in the Appendix A,

As addressed in the design of the serverless workflow language (Section 5.2) SWL is a non-ambiguous, executable workflow definition. Similar to the DAX format used in Pegasus [50], SWL is not meant to be used directly by users to define workflows—it is as an Intermediate Representation (IR) of a workflow. Therefore, users are encouraged to express their workflows in higher-level (programming) languages that are compiled down to this SWL-based workflow definition. Because a comprehensive study of workflow languages, usability-wise, is beyond the scope of this thesis, we implemented a workflow language that follows the current practices where possible, leaving the exploration of novel, better notations to future work.

We implemented a reference implementation of SWL: a declarative workflow language called SWL-YAML. Similar to the OpenStack Mistral Language [5], the Amazon States Language [2], and the Azure Logic Apps Language [3], SWL-YAML uses a human-readable data serialization language, YAML, to allow users to define workflows. To encourage the user to define the workflow declaratively as a single, large data structure, we decided to keep SWL-YAML close to the SWL model—which also helped reduce the effort spent on the workflow language.

Figure 1 presents an example of a full workflow definition written in SWL-YAML. The first task (*GenerateFortune*) triggers the execution of a serverless function called *fortune*, which returns a famous saying. The subsequent task (*WhaleWithFortune*), which requires *GenerateFortune* to be completed, takes its output, and wraps it in ASCII art in the *whalesay* function. This resulting ASCII art-wrapped saying is set to be the output of the overall workflow.

7.1.2.1. Workflow Expressions

To match the developer's workflow as close as possible, SWL-YAML supports *workflow expressions*, which are expressions that the user can employ to reference and manipulate data within the workflow definition. This allows users to perform common operations (such as, transferring data from the output of one to another task, or performing trivial data transformations) without requiring users to create new functions for each new operation.

```
apiVersion: 1
output: WhaleWithFortune
tasks:
  GenerateFortune:
    run: fortune

  WhaleWithFortune:
    run: whalesay
    inputs:
      body: "{ output('GenerateFortune') }"
    requires:
    - GenerateFortune
```

Listing 1: A canonical serverless workflow example expressed in SWL-YAML. See the text for the description of the example.

Conceptually, workflow expressions are an extension of the lifecycle of workflow invocations (see Section 5.3.4), making them an optional, interchangeable, component to the overall system. Implementation-wise, in SWL-YAML the initial implementation uses a JavaScript interpreter to resolve the expressions. During the compilation of the SWL-YAML workflow to SWL, the parser recognizes expressions using a specific pre- and post-ambles ({ and }, respectively) and marks them as being expressions. During the workflow invocation, a scope with all workflow metadata is dynamically injected into the JavaScript interpreter and stored in the global variable "$". The JavaScript interpreter then evaluates the expressions, replacing the input field containing the expression with the result. For example, to fetch the output of the task called "GenerateFortune", the user adds to the task inputs:

```
{ $.Tasks.GenerateFortune.Output }
```

Along with the scope of the workflow, utility functions are also added to the JavaScript interpreter, allowing users to employ shortcuts to simplify the referencing[2]. For example, the $task(id)$ function is a shortcut for $\$.Tasks[id]$ and $output(id)$ is a shortcut for $\$.Tasks[id].Output$. So the previous example can also be written as:

```
{ task("GenerateFortune").Output }
```

Or as:

```
{ output("GenerateFortune") }
```

Finally, users have access to the standard library of JavaScript to manipulate the data directly. For example, to retrieve the length of the output instead of the full output the user can use the $length$ JavaScript property:

```
{ output("GenerateFortune").length }
```

Workflow expressions provide users with a concise, yet expressive, tool to control the data flow. However, the use case for these types of expressions is explicitly limited to data referencing and trivial data transformations. We consider it, along with dynamic workflows, as an experimental feature. We have considered but leave explicitly for future work advanced functionality related to the workflow expressions—such as sandboxing, time-outs, and resource fairness.

7.1.2.2. Function Reference

Another key aspect of SWL-YAML is the structure of function references. Although SWL specifies a function reference as part of its data model (see Section 5.2.1), the exact implementation of this reference is left to the implementation. Figure 7.3 depicts the components of a function reference in SWL-YAML, which follows the Universal Resource Identifier (URI) syntax [129] for flexible and for users familiar naming. Before the workflow invocation starts, the function reference is resolved to a *function identifier*—a non-ambiguous reference of a specific version of a specific function on a specific FaaS platform.

[2]See the SWL-YAML documentation in Appendix A for a full overview of the implemented shortcut functions.

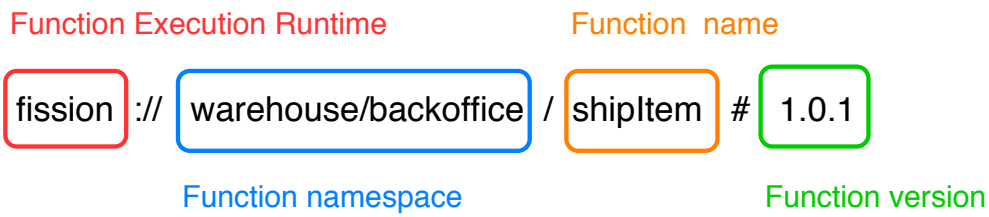


Figure 7.3: The structure of a function reference in SWL-YAML.

7.1.3. Other Implementation-Level Decisions

There were few constraints from Platform9 regarding the implementation of the prototype, except for the (few) technologies that were already widely in use throughout the company. We describe these constraints, and our experience working to meet them, in the remainder of this section.

7.1.3.1. Resource Orchestrator

As Fission is built to cater to the cloud-native community, it is focused on Kubernetes[3] with a corresponding reliance on container technology (Docker[4]). With Fission as the main technology supporting the workflow engine, and with Kubernetes steadily gaining more market-share as a key container orchestrator, we built the workflow engine with the same technologies.

7.1.3.2. Programming Language

Similarly to the decision on Kubernetes and Docker, we chose Go[5] as the programming language. The simplicity and open-source nature of the language, has lead to wide-spread adoption among projects in the distributed computing field (for example, Kubernetes, Docker, and Fission are all written in Go), which creates a snowballing effect of an abundance of well-established libraries and tools for the language. Furthermore, the language focuses on low-level, network-heavy applications—fitting the use case of a workflow engine.

7.1.3.3. Communication

For the communication, we decided to use Protobuf[6] (binary) message serialization format and gRPC[7] to both save time implementing clients/servers, and saving network bandwidth (which we expected to be the main bottleneck) with the compressed messages. Although there are many other options besides Protobuf that offer similar functionality, the popularity and pre-existing knowledge lead to the decision of incorporating them. We consider exploring other options in this space to be future work.

7.1.3.4. Event Storage and Distribution

For the event storage we evaluated two options. To optimise for performance (trading-in persistence) we implemented a lightweight, in-memory event store within the workflow engine itself. This both allows for the evaluation of what the minimal latency is that we incur with the event store, and it provides users with an option to deploy and test out Fission Workflows without needing to setup an external database.

For a more persistent event store, we decided to use NATS Streaming[8]; an open-source, high-performance message broker that supports persistent streaming, with a variety of data storage backends. As *messages* in message brokers can easily be used to fit the notion of *events*, we deferred the storage-related logic to this message broker. Along with its lightweight nature and close alignment with the Kubernetes ecosystem[9], we opted for NATS over other options, such as Kafka or RabbitMQ. Furthermore, the Fission team at Platform9 already had expertise with NATS and close relations to its development team, allowing us to get quick feedback and request features if needed.

[3] `https://kubernetes.io/`
[4] `https://www.docker.com/`
[5] `https://golang.org/`
[6] `https://github.com/google/protobuf`
[7] `https://grpc.io/`
[8] `https://nats.io/`
[9] Since the start of this work, NATS has been adopted as one of the open-source projects of the CNCF, the organization which also maintains Kubernetes.

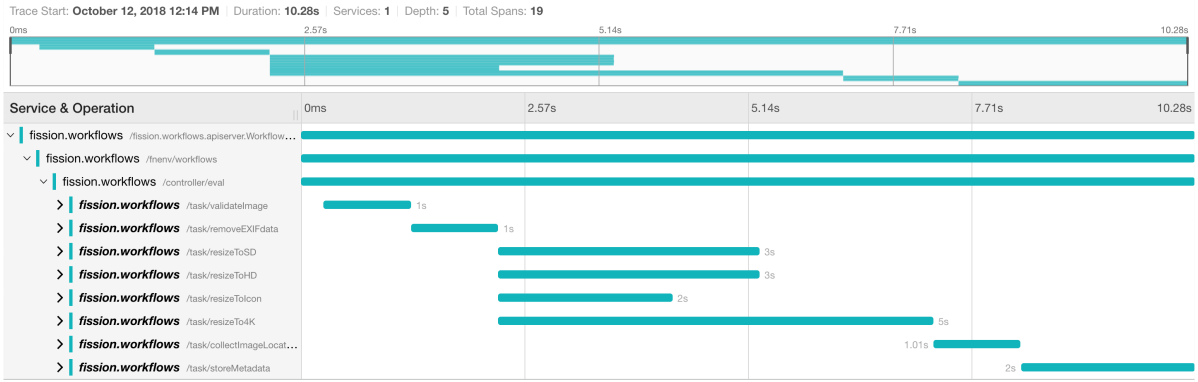


Figure 7.4: A trace of an invocation of a fictional image resizing workflow: the image provided to the workflow is first validated and processed (stripping EXIF metadata), after which the image is resized in parallel for four different formats (icon, SD, HD, and 4K), after which the locations of the resized images are stored in a metadata store.

Figure 7.5: A screenshot of one of the monitoring dashboards using the metrics exposed by the workflow engine, collected by Prometheus, and visualized using Grafana.

7.1.3.5. Monitoring

Because monitoring and provenance are key aspects of using a workflow management system (see background Section 2.2), Fission Workflows offers multiple ways to monitor and debug workflows. As shown in Figure 7.5, Fission Workflows integrates with the Prometheus time-series monitoring engine[10] to expose to the user a variety of metrics—from metrics measuring the performance of internal components to user-level performance concerns. Additionally, the WMS also provides comprehensive distributed tracing by implementing the OpenTracing API[11] using the Jaeger distributed tracing system[12]. An example of this tracing support is depicted in Figure 7.4. Finally, the Fission and Kubernetes ecosystems, which support the workflow engine, offer additional tools to help inspect workflows: Fission provides (distributed) log aggregation using Fluentd[13] and InfluxDB[14], and Kubernetes provides ways to query and inspect workflow definitions and metadata [4].

7.2. Process for Implementing Fission Workflows

Following an initial 3 weeks of designing the prototype and evaluation of the underlying components, we implemented the prototype over a period of 6 weeks. In this time, we developed the prototype using an agile methodology, in which (starting from the minimal functionality) we added functionality incrementally. Along the process, more than once we found that assumption were invalid, or we failed to address issues in the initial design, which led to reflection on and occasional augmentations of the design. With uncompromising, professional deadlines on the development of the prototype, the productization of the prototype started early on in the process.

Differently from what happens with the typical prototype, the productization process had a strong focus on usability and user experience from the start: (1) explicitly describing user experience, use cases, user stories, and step-by-step examples in the initial design; (2) creating examples and writing extensive documentation for components as soon as these matured enough to be tested by real users; (3) incorporating the usability of the system into the (weekly) progress meetings; and (4) making the installation of the system as straightforward as possible. Following the release and initial publicity, the emphasis on usability increased, and our productization process with it, by: actively evaluating the usability, incorporating early user feedback, and creating supporting documentation (such as writing blog posts, giving talks, and creating more complex, realistic examples).

We have taken, since the first prototype, a principled approach to the implementation of Fission Workflows. The code-base follows the opinionated conventions and practices of Go[15]. Overlapping with Go conventions, we followed principles from the UNIX philosophy [109]. Together these conventions lead to the code-base having mostly small, single-purpose packages, which aim to be small and composable. For this reason, the platform is now used in the AtLarge research group as a vehicle to support advanced research projects and other MSc. theses.

Another key factor, besides usability, what differs between a prototype and a product is the focus on *stability*. Users expect a productized system—even when it is in beta or alpha release stages—to be stable, at least for simple use cases. If examples provided along with the system do not work, users will stop exploring the system. However, foreshadowing the lessons learned (Section 7.4), we have to warn against creating extensive test infrastructure from the start, as this requires a major effort to get right. Especially for a frequently-changing prototype this might not yet be worth the investment.

To develop Fission Workflows, we tried to strike a balance between implementation and testing. Getting the testing infrastructure running was a prime concern; we implemented continuous integration (CI) and continuous deployment (CD) right from the start. Github[16], on top of Git, was used as the version control system (VCS) to keep track of the code base. For CI we relied on Travis CI[17], a hosted CI platform with close integration options for our VCS. As the system matured this we repeatedly improved the initial CI setup, by incorporating more extensive static code analysis, more thorough build checks, and better end-to-end test functionality.

[10] `https://github.com/prometheus/prometheus`
[11] OpenTracing is a effort of the CNCF to provide vendor-agnostic APIs for distributed tracing: `https://opentracing.io/`
[12] `https://github.com/jaegertracing/jaeger`
[13] `https://github.com/fluent/fluentd`
[14] `https://github.com/influxdata/influxdb`
[15] `https://golang.org/doc/effective_go.html`
[16] `https://github.com/fission/fission-workflows`
[17] `https://travis-ci.com/`

For CD, Fission Workflows contains automation for both contributors and users. The repository contains Makefiles[18] and shell scripts to automate the build and release process, allowing contributors to build artifacts (e.g., binaries, Docker images, changelogs) with a single command. For users, we provide tooling to make the installation of Fission Workflows as easy as possible. For example, we provide sensible defaults for configurations and integrate with the popular Kubernetes package manager Helm[19]. This reduces the installation of the workflow system on top of a Fission deployment to a single command. Setting up the CD infrastructure required considerable time and expertise, but we estimate that the effort was well spent.

The thoroughness of testing of the system was ramped up as the prototype moved towards becoming a product. During the initial prototype, we implemented tests once components stabilized in functionality and implementation. First, we utilized small functional tests to verify that component was functioning correctly for the main behavior. Second, as the component stabilized in implementation unit tests were added. Finally, as we integrated the component into the workflow engine, integration tests were added.

Ensuring that a system is stable and usable, also means that documentation and examples should reflect the actual system, and avoid (as happens with many systems) describing an out-of-date version of the system. Due to the complexity of the infrastructure—the testing infrastructure has deploy and manage, besides the workflow engine itself, Kubernetes, Fission, and message queue deployments—we added end-to-end testing after the workflow system started to stabilize. Besides the expected end-to-end tests, all self-contained examples in the repository are utilized in the end-to-end testing; ensuring that the examples are up-to-date. Our current documentation and examples are accessible at `https://github.com/fission/fission-workflows/tree/master/Docs` and `https://github.com/fission/fission-workflows/tree/master/examples`, respectively.

7.3. Real-World Impact

Throughout the public transition from the prototype to the increasingly production-ready product, Fission Workflows has garnered real-world attention and impact. In Appendix B we provide an overview of the talks, news articles, blogs, and podcasts that either center around or make a substantial mention of Fission Workflows.

Companies in the US, Europe, and Asia are—at the moment of writing—evaluating Fission Workflows internally as part of their serverless strategy. The domains of these users range from data-intensive video processing, to business process management systems in the healthcare and insurance industry, to machine learning. Besides these larger users, both the GitHub issue tracker[20] and the Slack community communication platform with 600+ users[21], occasionally receive questions, bug reports, and feature requests regarding the workflow engine.

A percentage of these (potential) users have also started to get actively involved in the further development of the Workflows product. The involvement of these external contributors occurs in various aspects of the project: users taking part in discussions on the roadmap and prioritizing of specific functionality; users sharing feedback and experience reports on their experiences using the product; and users contributing directly to the code-base.

7.4. Lessons Learned

The uncommon trajectory of this research—from prototype to publicly released and used system within months—has also resulted in various insights of what to expect, what practices to encourage, and what practices to avoid. In this section we share the lessons we have learned in the process of researching, designing, implementing, and evaluating the Fission Workflows prototype.

7.4.1. Start data collection as early as possible

The collection of data for (performance) evaluation is often an under-appreciated aspect of a research project. However, especially with a time-constrained project—such as a thesis—data collection should be a key priority from the start. Deferring data collection for later is not only risky time- and effort-wise,

[18] `ftp://ftp.gnu.org/old-gnu/Manuals/make-3.79.1/html_chapter/make_2.html`
[19] `https://helm.sh`
[20] `https://github.com/fission/fission-workflows/issues`
[21] `http://slack.fission.io/`

it also prevents gaining early insights that rely on the real-world data. Ideally, either the data is already present at the start of the project, or you should be working closely with an industry partner on a *specific problem*.

With Fission Workflows, we stumbled on this issue in several ways. On the one hand, we worked with companies from the financial and insurance industry, where data collection involved various regulation, and required many regulatory hurdles, sign offs and stakeholders—to the point that the data collection was dropped. On the other hand, we started the process of collecting data, and especially workload traces, only after the initial public release of the product. And, in one particularly unlucky case the Californian fires of 2018 delayed significantly this process, because the company that was interested in providing traces had to shift its priorities to restoring their own operations.

7.4.2. Focus on a specific niche or use case

Related to starting data collection early, it is helpful to focus on a specific niche or use case for a project early on. First, it helps to narrow the scope of the system, and to focus the design on concrete (and thus crucial) features. Second, it justifies and promotes the system, by being able to showcase its advantages for a concrete use case. Third, it attracts customers with similar use cases or problems, and thereby increases the chances of getting early users, feedback, and more data—this is the *virtuous cycle* of transforming a prototype to a product in the current cloud ecosystem.

Although Fission Workflows was and is intended as an approach to generic function composition, in our view not focusing on a specific use case early impeded the progress early on. Instead, we started generic, causing the process to involve much speculation about what users want and how the system would be used. In hindsight, the rapid convergence of user wants seems to indicate it would have been better to focus initially on a specific use case or problem. And, generalize only after we validated the system for the use case.

7.4.3. Marketing and communication take up time

A key differentiator between research and industry projects seems to be the importance and the approach to marketing and communication. Different from many academic research projects, it is common for industry projects to not have a specific end-goal; once the prototype is released, the process becomes an ongoing effort of improving marketing and communication, to justify and promote the project to attract both user interest and—in the case of open-source software—attention from the developer community. This effort, which includes writing blogs, recording screen-casts, talking to potential users, supporting existing users, and presenting at conferences, takes up valuable time and energy—and yet are vital to the continuity of the project. We see these lessons from the industry being increasingly more adopted by the research community; for example, by the AtLarge team.

The amount of time and effort required for the marketing and communication part of Fission Workflows too was surprising—at least to the inexperienced, academic part of the team. At times, most of the time-budget allocated to the project was spent on: writing blog post material, creating presentations, polishing documentation, and improving the examples and demos. For example, in anticipation of the first public release in October 2017, 3 weeks of preparing the communication and PR material preceded it.

7.4.4. Every dependency adds complexity, technically and organizationally

An—appropriately—dominant opinion and practice in computer science is to avoid reinventing the wheel: only implement functionality that you can not reuse from another system of library. However, in this project we also learned that there is a certain trade-off made when adding a library or system as a dependency. There is no question that well-proven, production-ready systems are better to simply reuse. Yet, not much software is that *ready-to-use* in practice—even though all marketing and documentation suggest, and in some cases even boast, that it is. This can lead to the situation, where the time spent waiting for dependencies to solve issues important to your project may even exceed the time spent developing it.

While implementing the prototype, we learned about this trade-off the hard way, by depending early on promising, but still evolving, systems. Fission itself was less robust and feature-rich than it is at the moment of writing, requiring functionality to be added along the way to support the workflow engine. Next to Fission, we chose to depend for the prototype on NATS message queue, for its performance characteristics. As we learned along the way, the NATS system was less feature-rich and less stable

than Kafka. These dependencies slowed down the development of the workflow engine, as the time it took to request and discuss changes, to implement changes, and to wait for the next release was far longer than engineering the features of the prototype itself. Although Fission was vital to the prototype, NATS was less so, and could have been replaced by a more robust—albeit less performant—alternative until it was mature enough in both functionality and performance.

7.4.5. Start end-to-end testing of distributed systems early, but not too early

End-to-end testing of distributed systems is notoriously complex. Part of this complexity comes from the stack of middleware systems on which state-of-the-art systems rely on to offer globally-distributed, reliable services–a true ecosystem [74]. Unit and integration testing of components within a single-node context already helps improve the robustness of the system. However, as noted by Yuan et al. [142], there exists a large category of bugs that only manifest themselves in the distributed settings of systems. Even by implementing a few end-to-end tests—for example, tests mimicking user-level input and errors—can help detect a majority of these bugs.

However, as we noticed while implementing the Fission Workflows prototype, setting up and maintaining a comprehensive end-to-end testing environment is time-consuming—especially at the start of a project when the code is volatile. With Fission Workflows, we knew this from start, as a consequence from the engineering effort that went into the Fission FaaS. Therefore, initially we focused on testing components individually, and only near the release of the workflow engine (when the prototype matured) we started building comprehensive end-to-end testing infrastructure—and we indeed found subtle bugs as described by Yuan et al. [142].

7.4.6. Useful examples and demos areas important as documentation

In most (research) projects, examples are added to the project repository as an after-thought; it is that one example that the developer has been using to test the system, or the example that is reused in presentations to explain the functioning of the system. More often than not the examples are outdated, or are simply not useful. Helpful examples serve as use cases, showcases of functionality, and documentation to the user—which are often consulted more than documentation itself, as the GitHub statistics of the Fission and Fission Workflows suggest.

The prototype has emphasized the use of examples as a way of documentation from the start. Newly introduced features were accompanied by a canonical example to show the functionality. And, indeed, most questions received were either related or specifically about one of the examples. Therefore, Fission Workflows contains a wide variety of examples ranging from trivial examples that show a specific part of functionality, to examples for specific use cases, to large, realistic examples that serve as demos. To avoid outdated examples, all examples are run and verified directly in the continuous testing infrastructure.

7.4.7. Getting users is the best and worst thing that can happen to a prototype

Fission Workflows was released to the public in an early stage (in September 2017). Along with exposure from TechCrunch and other news outlets (see Appendix B) over the following months, this project generated interest from various types of users—ranging from hobbyists, to other academics, to large organizations. This lead to various meetings with interested organizations and managing support for deployments of the prototype—and bug reports!

The feedback of these early users is invaluable to detect mistakes (both in execution and design), and to get a better understanding of how user are interacting and want to interact with the system. This helps validate the initial assumptions, and also (potentially) reshapes roadmap to maximize the benefits to the users. Most importantly, it validates the demand for a solution to the problem—in this case that of serverless function composition.

On the other hand, user adopting the system slows down the development. First, it is no longer possible to break the API, as there are now are actual users that will be affected by these changes. Second, small bugs and other issues in edge cases, that are largely ignored in prototypes, need to be addressed—and in some cases prioritised. Finally, writing comprehensive documentation, providing examples, and offering support to users all take up time that could otherwise be spend on development and evaluation.

7.4.8. Usability should be one of the first-class concerns; even of research projects

Related to the lessons learned about having useful examples, documentation, and testing, the usability of a system is key to adoption and continued interest. Even a project that is not intended to be used outside of a small (internal) group of users, the usability should be considered a primary concern. Similar to the principles underlying well-established ecosystems (such as UNIX [109]), focusing on usability, simplicity, and the avoidance of complex ”magic” functionality results in software that does not become obsolete once the original developers stop working on it. For research specifically, focusing on usability is a way to stand out as research project, and to have a project's software picked up and reused by future researchers.

For Fission Workflows the usability aspect has been both a strength and weakness. Usability was a key aspects that was—and still is—focused on throughout the process. For example, it led to the emphasis on keeping the mental (execution) model of the system simple, and keeping interface similar to that of well-known WMS and the regular FaaS interface have been core pillars. On the other hand, the complexity of a workflow engine, combined with the limited resources of this thesis, inevitably seeped through to the user—impacting usability of the initial prototype. However, by regularly evaluating and focusing on specific enhancements that improve the usability, the usability is one of the aspects that has consistently improved since the initial release of the product.

7.4.9. Event-sourcing and control theory are key techniques for practical, distributed systems

Event-sourcing [52] and control theory (see background Section 2.3) are increasingly being adopted by distributed systems engineers. event-sourcing, recording operations (or events) to store and replay the state, allows systems to both preserve the time-related information, restore state in case of errors, and promote event-driven architectures [58]. Control theory, utilizing reconciliation loops and controllers, improves the resiliency of distributed systems by minimizing the reliance on fragile, time-dependent sequences of operations (see Section 2.3)

Fission Workflows relies on both techniques. With the event-driven nature of serverless functions and workflows, the event-sourcing approach fits the model of the WMS. Following well-established practices of controllers in both Kubernetes and Fission, the workflow engine also relies on a controller to continuously monitor the workflow invocations, increasing the resiliency of the workflow invocations encountering failures. However, both technologies lack well-established open-source tools and libraries, making the initial development of systems using these concepts challenging.

7.4.10. Using another project in the system adds free publicity

For a system, especially a prototype, you are regularly faced with a common consideration: Do we want to depend on an external project or should we just implement it myself? Common practice advises you to avoid implementing components yourself wherever an alternative is present. On the other hand, especially for prototypes, it is often more time and effort effective to simply implement the few features you need yourself; understanding and integrating bloated dependencies for a small feature—which also turn out to be redundant or the dependency might turn out not fitting the functional and non-functional requirements—could turn out to be a waste of valuable time. In between these two extremes there is a large gray area, in which it is not entirely clear (beforehand) which of these two approaches is more favourable.

One aspect that should be taken into consideration is that choosing to use another project in the system or prototype will, with effort, lead to other people developing an interest in the succeeding of the project. Organizations will typically be enthusiastic about other people and projects relying on their projects. Especially for smaller, early-stage, and active projects the usage of their project provides them with feedback, and an use case to provide to potential users.

For Fission Workflows specifically this worked out well. The early, explicit choice of using the popular Fission FaaS platform (see Section 4.1), allowed this prototype to have, next to a stable FaaS layer, intensive backing in terms of resources, support, and public exposure. Choosing for an ad-hoc solution or a fork of another academic project (e.g., OpenLambda), would most likely not have resulted in this amount of impact. Additionally, the choice to use the, albeit more immature, NATS message queue over Kafka, resulted in close collaboration with the authors (see Section 7.1.3 and exposure regarding this system in terms of blog-posts [6].

8

Experimental Evaluation through the Fission Workflows Prototype

To evaluate the workflow-approach to function composition, we conduct real-world experiments with the Fission Workflows prototype. In Chapters 5 through 7, we describe the design and implementation of this prototype. Following the framework proposed by Jain [77], we systematically approach the experimental setup in Section 8.1. Using the setup, we then conduct the experiments and report the results in Section 8.2. Last in this chapter, we discuss the implications and limitations of the results, in Section 8.3.

8.1. Experimental Setup

The goal of this experimental evaluation is to validate the prototype based on the initial goal of this work: to find an approach to compose serverless functions reliably and with low performance-overhead. We aim to assess the Fission Workflows prototype the following criteria common for WMSs (see Section 2.2.5):

1. **Performance overhead**: We focus on the performance overhead because of requirement R3 (see Section 3.4), which states that the function composition approach should minimize performance overhead. In the evaluation we aim to answer questions such as how does our workflow-based approach to function composition compare to other forms of composition (see Section 4.2) in terms of performance overhead? How does it compare to existing workflow engines (see Section 4.3))?

2. **Reliability**: In R4 we state that a suitable function composition approach should have reliable executions, that is, the system should be able to execute workflows successfully in the face of an unreliable environment (e.g., network disruptions, and machine crashes). Is our workflow-based approach to function compositions reliable in the face of transient and fail-stop failures? How do the resiliency measures of the design (see Section 5.3.1) and the configurable persistence (see Section 5.3.2) affect the reliability?

3. **Scalability**: In the requirements R6 and R7 (see Section 3.4) we determined that a satisfactory function composition approach needs to be scalable in both the number of workflow definitions and workflow executions. Therefore, we evaluate questions, such as how does the serverless workflow engine scale in the face of growing adoption of serverless computing for increasingly larger (production) workloads? What type of resource is the bottleneck at scale?

4. **Cost**: To put the previous criteria into perspective, we evaluate the cost of workflow executions using the prototype. This should give us an indication whether the system is achieving good results in the other criteria, but at unrealistic high cost. What is the impact of the approach to function composition on the cost model? Is the cost model in line with existing workflow engines?

Note that, we do not evaluate *security* in this performance evaluation, despite it being an important metric in the evaluation of a WMS, as described in Section 2.2.5. The reason for this is that there is, to the best of our knowledge, no existing, established approach or framework for evaluating security in (serverless) WMSs. Although we are constructing a new setup for the other criteria precisely because there is no existing framework too, we deferred the evaluation of security to future work to limit the scope of this evaluation.

In the remainder of this section we describe the various aspects of the experimental setup. We first identify the system boundaries (see Section 8.1.1) and design the experiment infrastructure. In Section 8.1.2, we further describe both the synthetic and the trace-based workloads used in experiments. We conclude this section with an overview of the experiments (Table 8.4) and their corresponding design in Section 8.1.3. In Appendix C we provide further detail on the exact configurations used in the experiments.

8.1.1. System Boundaries

Although seemingly self-understood, good systems experimentation defines carefully the system boundaries. This is because systems are often nested collections of subsystems and are deeply interconnected with other systems. Therefore, setting the boundaries enables the experiments to work on a well-defined target.

Figure 8.1 visualizes these system boundaries for this performance study. The key component of the system-under-test is the workflow engine—the Fission Workflows prototype, as described in Chapter 5. We consider other components that facilitate composition of functions to be relevant for specific aspects of the study. Although the workflow engine relies on a container orchestrator (Kubernetes) and a FaaS platform (Fission), these are not considered prime targets for the evaluation—these components will be utilized (or avoided) consistently, to control for their impact on the evaluation.

8.1.1.1. Workload Driver

All controlled experiments need to control their input workload. This is done in this work through a Workload Driver. Though the workload driver is in part specific to the experiment, at its core these rely on (customized) state-of-the-art HTTP workload generators: Hey, Vegeta, and Wrk.

For the experiments that use the realistic Chronos workload trace (see Section 8.1.2.2), we wrote a custom workload driver logic using the HTTP library in Go. The reasoning behind this is that the workload trace is small enough—it has a mean QPS of 5.4—to be able to be replayed using the default HTTP library.

Hey[1] is a small, versatile HTTP workload driver. It supports HTTP2, and allows for synthetic workloads using a concurrency parameter. We use Hey in the scheduling and stress-testing experiments designed to evaluate the scalability of the prototype (see Section 8.1.3 and 8.1.3).

Vegeta[2] is a comprehensive HTTP workload driving platform. In contrast to Hey, it records every request and response, and is able to generate interactive graphs of the results. Furthermore, it allows users to specify a QPS target, which Vegeta will aim to adhere to. However, of the three workload drivers, based on our experience, Vegeta seems to be the least performant. We use Vegeta in the fault tolerance experiments (see Section 8.1.3) to create a synthetic QPS-based workload.

Wrk[3] is a high-performance HTTP workload driver based on HTTP 1.1. During the performance evaluation of the SimFaaS FaaS platform emulator (see Section 8.1.1.4), we found that the other workload drivers—Hey and Vegeta—were inefficient in their CPU utilization. Wrk was used instead, since it focuses on workloads of highly-concurrent HTTP connections. The downside of Wrk is that it—at the moment of writing—does not support HTTP 2, the lack of which can increase the performance overhead. Therefore, we only use Wrk as the workload driver in the the highly concurrent SimFaaS experiments (see Section 8.1.3).

We extended and contributed to these tools in various ways. We contributed to Hey in several ways: adding support for rate-limiting, and improving the request logging. We also extended Vegeta to also support custom trace-based workload generation—though this did not make it in the upstream version of Vegeta.

[1] `https://github.com/rakyll/hey`
[2] `https://github.com/tsenart/vegeta`
[3] `https://github.com/wg/wrk`

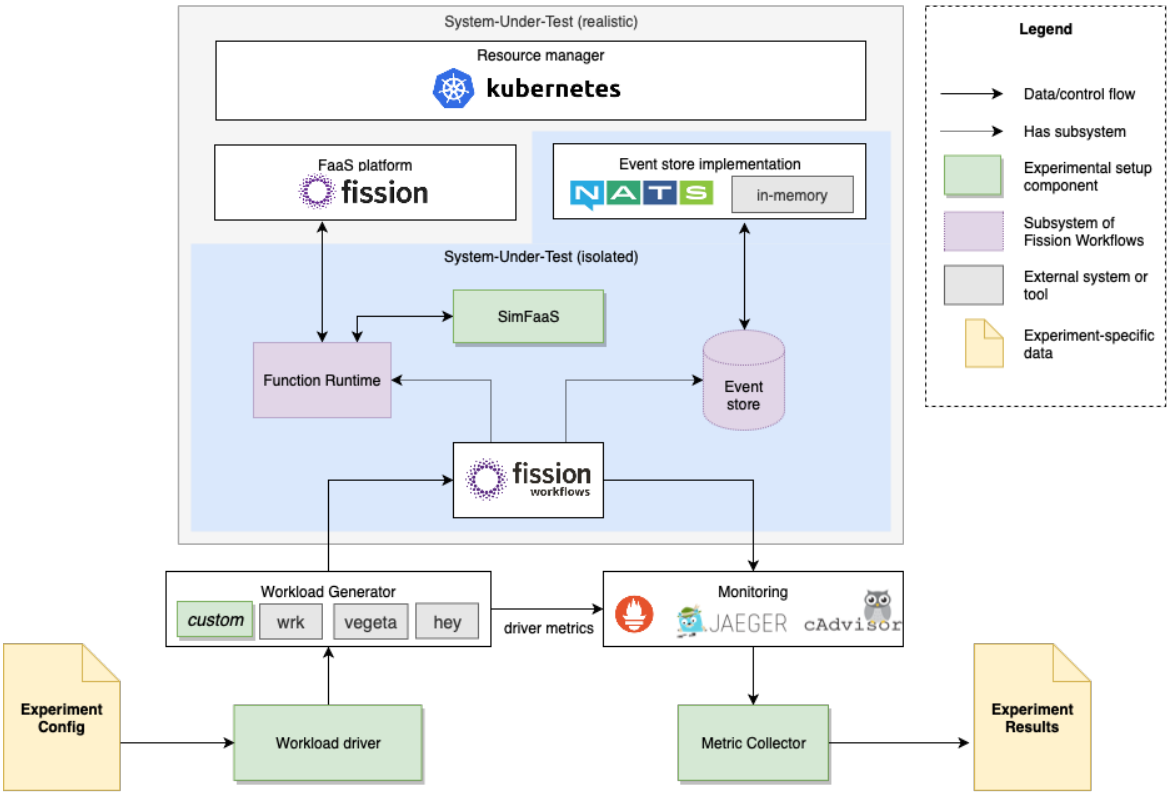


Figure 8.1: An overview of the experimental setup. The green boxes represent the components serving as the experiment infrastructure, the purple boxes representing the internal components responsible for communication across the system boundary with backing services, and yellow to indicate experiment-specific data.

Machine ID	CPU specification	Memory	Disk	Network	Cost (per hour)
H1	4 x Intel Xeon (Skylake)	16 GB	160 GB SSD	1 Gbit/s Ethernet	0.12 EUR
H2	8 x Intel Xeon (Skylake)	32 GB	240 GB SSD	1 Gbit/s Ethernet	0.06 EUR
H3	16 x Intel Xeon (Skylake)	64 GB	360 GB SSD	1 Gbit/s Ethernet	0.24 EUR
G1	1 x Intel Xeon (Skylake, virtualized)	3.75 GB	10 GB SSD	1 Gbit/s Ethernet	0.04 EUR

Table 8.1: An overview of the types of hardware used in the experiments.

Setup	Driver Machine(s)	SUT Machine(s)	Components/Services
isolated	1 x H2	1 x H1	SimFaaS, Workflows, NATS
isolated-XL	1 x H3	1 x H2	SimFaaS
GC	1 x G1	4 x G1	Google Cloud Composer, Google Cloud Functions
realistic	1 x G1	4 x G1	Fission, Workflows, NATS, Prometheus, Kubernetes
AWS	1 x H2	-	AWS Step Functions, AWS Lambda

Table 8.2: An overview of the setups used in the experiments. For machine details, see Table 8.1.

8.1.1.2. Metric Collector

For the metric collection we rely on both state-of-the-art metric collectors and customized collectors. Because both the workflow engine and the supporting systems—Fission, SimFaaS, Kubernetes—expose Prometheus-compatible metrics, we rely particularly on the Prometheus monitoring system[4], which incorporates state-of-the-art monitoring technology. Similarly, we rely on cAdvisor[5] to report machine- and container-level metrics in a Prometheus-compatible format.

8.1.1.3. Infrastructure Setup

To experiment with Fission Workflows, we deploy it on machines with relevant hardware (virtual?) properties. The collection of machines, together with the software deployed on it, represent the System Under Test (SUT). Table 8.1 lists all the types of VMs used in the experiments. All the machines were rented either from Hetzner Cloud[6] (H1, H2, and H3), or from Google Cloud (G1). We choose these two clouds, because, to the best of our knowledge, these clouds are realistic representatives of current cloud providers. Google Cloud represents the main public cloud providers, such as Azure, AWS, and IBM cloud. Hetzner represents the smaller cloud (infrastructure) providers, such as OVH, Rackspace, and DigitalOcean. Furthermore, within these two clouds, we rely on regular, average-sized machines to represent a realistic cloud user infrastructure.

Using these machine types, we define the set of infrastructure setups summarized in Table 8.2. In each infrastructure setup, we select a *driver machine*, that is, the node which will be responsible for generating the workload for the SUT, and one or more *SUT machines*, that is, the nodes which the SUT can utilize to handle the workload. Moreover, the infrastructure setup also describes the components (see Figure 8.1) of the SUT to be deployed and the services that we setup for each experiment.

Each infrastructure setup involve multiple machines, and in some cases machines and services across multiple clouds. To mitigate latency issues not related to the goals of our experiments, we deploy the machines as close to each other as possible. Specifically, we deploy the machines in the same datacenter if they are from the same cloud provider (i.e., for the setups `isolated`, `isolated-XL`, `GC`, `realistic`). Otherwise, we deploy the machines physically as close as possible (e.g., for AWS we deploy the machine and the services both in datacenters in the west Germany availability zone).

8.1.1.4. SimFaaS: a FaaS Platform Emulator

The goal of SimFaaS is to enable controlled yet realistic experiments, by emulating all relevant Fission interfaces but by also making the main operations stable—much like a production-ready Fission platform would be expected to operate. We implemented an initial version of SimFaaS[7] that consists of a

[4] `https://prometheus.io`
[5] `https://www.github.com/google/cadvisor`
[6] `https://www.hetzner.com/cloud`
[7] The SimFaaS project is separate from Fission Workflows, and can be found here: `https://github.com/erwinvaneyk/simfaas`

Workload	Type	Duration	Arrival Pattern	Task runtime	Workflow count	Tasks/workflow
chronos	Trace	11 m	5.4 QPS	8.1 s (median)	1	3
chronos-unique-fns	Trace	11 m	5.4 QPS	8.1 s (median)	3402	3
simfaas-stress(N)	Synthetic	5 m	N concurrent	1 s	-	-
wf-parallel(N)	Synthetic	5 m	1 concurrent	1 s	1	N (parallel)
wf-serial(N)	Synthetic	5 m	1 concurrent	0.1 s	1	N (serial)
static-qps(N)	Synthetic	5 m	N QPS	1 s	1	3

Table 8.3: The workloads used in the experiments.

single HTTP-accessible function that sleeps for a specific duration based on parameters in the HTTP request. SimFaaS adheres to the exact Fission API, which allows us to use it instead of a full-fledged Fission deployment. This enables us to minimize the external factors in focused, synthetic experiments, while relying on a full Kubernetes and Fission deployment in realistic experiments.

To ensure that SimFaaS provides minimal overhead to the emulations of functions and does not consume a critical amount of resources, we evaluate its performance characteristics in Section 8.1.3. Foreshadowing, we find that SimFaaS has minimal overhead, and thus can be used as a transparent FaaS emulator in the experimental setup when assessing the performance of Fission Workflows.

8.1.2. Workloads

e use across all experiments one of two types of workloads, as follows. For experiments concerned with evaluating the impact on performance of the workflow engine under varying workload parameters, we use a comprehensive suite of *synthetic workloads*—generated specifically to the target evaluation on a specific change in workload parameters. For experiments concerned with evaluating the performance of the workflow engine under realistic circumstances, we use the *real-life workload traces* obtained from a workflow system in an industrial IoT setting. We detail each of the workloads in turn in the following.

8.1.2.1. Synthetic Workloads

For most experiments in which we evaluate the performance of the workflow engine, and how it is impacted by changes in workload characteristics, we use synthetic workloads. This is because synthetic traces allow us fine-grained control over the main workload aspects that we believe can cause bottlenecks in the system. We fully understand the importance of realistic and real-world traces for experimental system evaluation, but such traces are currently not readily available for serverless work, and there are also theoretical considerations regarding their ability to be used in joint experimental and analytical studies.

For the generation of the workflow structures we created *fission-workflows-generator*, which is a tool that generates workflows. The tool can be controlled using a number of parameters, including (1) the number tasks in the workflows; (2) the number of workflows needed; (3) the degree of dependency between tasks; (4) the runtime of the functions within the tasks; (5) the size of the data being transferred in and out of the workflow; (6) the size of the data being transferred between functions. Because the evaluation of the performance of the FaaS platform is out of the scope of this study, the tool uses a single custom function to emulate function run time and data transfer; this is similar to previous approaches to benchmarking, e.g., to the approach taken in Grenchmark [72].

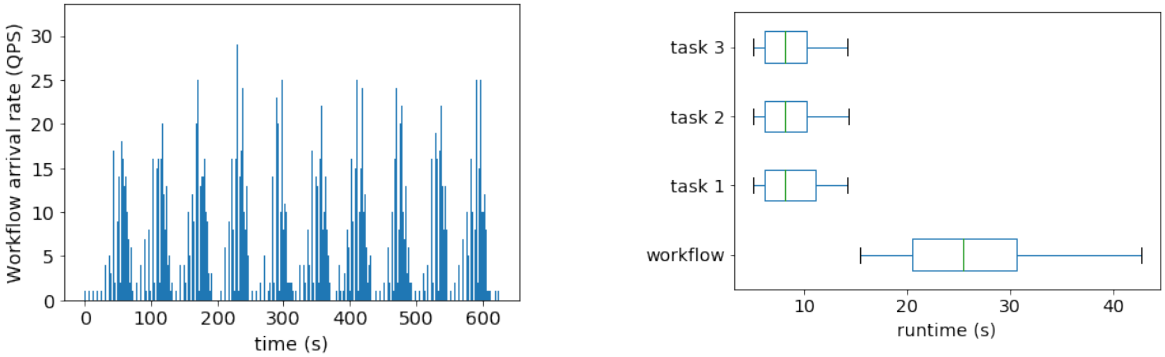
We generate the arrival pattern of workflow invocations (of queries) using the HTTP workload generators. As described in Section 8.1.1.1), these tools can generate synthetic workloads based on throughput (QPS) and concurrency targets.

8.1.2.2. Real-life Workloads

Given the immaturity of the serverless cloud model, no serverless workflow traces or workload characteristics have been released for community use. To validate the system using real-world data we choose to use the Chronos workload [92]—workload traces from a workflow deployment in an industrial IoT setting. Although these traces are *not* explicitly serverless workflows, they share characteristics with serverless workflows as defined in Section 4.5. We consider this workload representative for a serverless workflow for the following reasons:

1. As shown in the requirements analysis (Chapter 3), the IoT domain is one of main use cases for

Figure 8.2: The structure of the workflows in the Chronos workload trace. A sensor measurement is taken, passed through a filter, and then written to a database.



(a) The arrival pattern of the workflow invocations.

(b) The runtime characteristics of the workflow and tasks.

Figure 8.3: Characteristics of the Chronos workload trace. Plot (a) represents a more detailed analysis of the workload characterized in [92].

serverless workflows.

2. The tasks comprising the workflows in Chronos are responsible for reading (sensor) data, filtering data, and writing data to a database. As prior work argues [139], these kind of responsibilities are common use cases for serverless functions.

3. The structure of the workflow (see Figure 8.2), although not dynamic or complex, is based on a small number of tasks with few interconnections, more a pass-through sequence, which resembles the granular nature of serverless workflows.

4. The arrival pattern of the workflow invocations (Figure 8.3a) resembles that of serverless workflows; the workflow are invoked frequently and the arrival patterns varies over time.

5. The task runtimes, as represented in Figure 8.3b, fall within the range of serverless function runtimes, with the tasks executing for a couple of seconds. Correspondingly, the recorded task runtimes are measured in milliseconds.

6. Though the trace is collected over a relatively small time window (see the trace summary in Table 8.3b), it is long enough to show clear patterns in the runtimes and in the arrival patterns. This allows us to create longer and larger traces—we extrapolate longer traces by joining multiple instances of this window.

   This presents a trade-off in our experiments: the extrapolation limits the realism of the experiments, but does enable us to conduct longer-running measurements that limit the effects of spurious system instability. An investigation of this trade-off falls outside the scope of this work, and could lead to new methodological guidelines for analyzing the performance of serverless platforms.

With the increasing interest of both industry and academia in serverless computing, we expect workload traces specific to serverless workflows to emerge over to the following years. More extensive experimentation using such traces to evaluate and compare the performance of new and existing serverless workflow management systems is therefore promising future work.

§	Category	Experiment	Setup	Workload	Event Store
8.2.1	A. SimFaaS	concurrency stress test	isolated-XL	simfaas-stress(0, 100k, 500)	-
8.2.2		fail-stop	isolated	static-qps(10)	in-memory, NATS
8.2.2	B. Fault Tolerance	transient net fail: FaaS	isolated	static-qps(10)	in-memory
8.2.2		transient net fail: event store	isolated	static-qps(10)	NATS
8.2.3		event store implementations	isolated	chronos	in-memory, NATS
8.2.3		workflow ingestion	isolated	static-qps(0, 30k, 1000)	in-memory
8.2.3	C. Scalability	workflow throughput	isolated	static-qps(0, 1000, 10)	in-memory
8.2.3		workflow length	isolated	wf-serial(0, 900, 100)	in-memory
8.2.3		workflow parallelism	isolated	wf-parallel(0, 1000, 100)	in-memory
8.2.4	D. Scheduling	policy comparison	isolated	chronos-unique-fns	in-memory
8.2.5	E. Real-world	state-of-the-art comparison	realistic, AWS, GC	chronos	in-memory, NATS

Table 8.4: An overview of all experiments in this thesis. For setup details, see Table 8.2. For workload details, see Table 8.3.

8.1.3. Design of Experiments

We aim to conduct a diverse set of experiments, an in particular investigate the scalability and fault-tolerance of Fission Workflows through a set of experiments each. Table 8.4 summarizes all experiments in this thesis. In the remainder of this section we describe the general goal, method, and metrics for each of six categories of experiments.

A. SimFaaS experiments

The goal of the SimFaaS experiments is to evaluate the performance impact of the SimFaaS FaaS emulator. Because the use of it is meant to minimize the impact of the FaaS platform on the results of the workflow experiments, we use these experiments to evaluate if the tool scales enough for the expected workflow workloads, and in particular if it does not create contention in either of the resources. For this we use the `isolated` infrastructure setup, using a synthetic workload to drive an increasingly more concurrent workload to the SimFaaS deployment. During the experiment, we track resource usage of both the driver and worker node to ensure that the driver is not the bottleneck in the experiment.

B. Fault-Tolerance experiments

Because one of the key advantages of a workflow orchestrator over the other function composition approaches (see Chapter 4) is that provides stronger resiliency in the face of failures, the goal of the fault-tolerance experiments is to analyse if these theoretical benefits hold true for the real-world prototype. We evaluate a set of three types of failures:

1. **Fail-stop**: the workflow engine crashes and restarts. We evaluate two variations: (1) a bundled engine, where all components are run within the same process, and (2) a distributed engine where the components are separate processes and only the engine (or controller) process crashes. To simulate the crash we kill and restart the process at specific points in time.
2. **Transient FaaS network failure**: in this scenario we emulate an intermittent network failure between the workflow engine and the FaaS platform. To simulate the network unavailability, we pause and resume the FaaS platform at specific points in time.
3. **Transient event store failure**: here we emulate network unavailability between the workflow engine and the backing event store. To simulate this, we pause and resume the event store at specific points in time.

To observe the impact of failures on a workload, we use a synthetic workload (see Section 8.1.2) comprised of a simple workload of a workflow with 3, sequential, 1-second tasks with a predictable arrival pattern of 10 QPS. Although we track all metrics for all SUT components and the two machines, we focus specifically on the following metrics: the number of workflows successfully submitted, the number of workflows failed to be submitted, the number of workflows successfully completed, and the number of workflows that failed.

C. Performance experiments
The performance experiments aim to evaluate the performance of specific components and workload characteristics through stress testing. As summarized in Table 8.4, we evaluate the following components and workload characteristics:

1. **Event storage**: to evaluate and compare the performance of the event store implementations: in-memory (*mem*), NATS Streaming (*nats*), NATS Streaming with filesystem persistence (*nats-file*), and NATS with SQL database persistence (*nats-db*). We run the Chronos trace (see Table 8.3 and associated text) for each of the event stores and compare the results.

2. **Workflow ingestion**: to evaluate the performance of the ingestion of workflow invocations, and to verify that it does not become a bottleneck in other experiments. We use the `static-qps` synthetic workload to find the maximum throughput achievable, running iterations of the experiment from 0 to 30,000 QPS, incrementing the QPS by 1000 for each iteration.

3. **Workflow overhead**: to analysis the general overhead introduced by the workflow engine, we run a synthetic workload of executions of a 1-task workflow. This allows us to compare the performance of this workflow and the performance of the (SimFaaS) function (see Section 8.1.3) at a comparable level of load. We use the `static-qps` synthetic workload, running iterations of the experiment from 0 to 1,000 QPS, incrementing the QPS by 10 for each iteration.

4. **Workflow length**: to gain insight in how the workflow engine handles larger workflow lengths, we run a synthetic workload in which we increase the length of the workflow at every step of the experiment. We use the `static-qps` synthetic workload, running iterations of the experiment from 0 to 900 QPS, incrementing the QPS by 100 for each iteration.

5. **Workflow parallelism**: to evaluate how the workflow engine handles increasingly larger parallel workloads, we run a synthetic workload in which we increase the size of a fully-parallel workflow at every step of the experiment. We use the `static-qps` synthetic workload, running iterations of the experiment from 0 to 1,000 QPS, incrementing the QPS by 100 for each iteration.

To eliminate the external influence on the results of these experiments, we perform each experiment in the `isolated` infrastructure setup, using the `in-memory` event store, and using the SimFaaS FaaS emulator to minimize the impact of dependencies.

For each of the experiments, we aim specifically to track five metrics that reveal impact of the workloads on the performance and the resource consumption: (1) duration of the workflow invocations in milliseconds, measured from the arrival of the workflow invocation at the workflow engine until the workflow invocation has completed; (2) CPU load, defined as the CPU utilization by the SUT and measured on a 5 second interval through Prometheus; (3) Memory usage, defined as bytes occupied and measured on a 5 second interval through Prometheus; (4) data transmitted over the network in bytes per second; and, (5) data received over the network in bytes per second.

D. Scheduling experiments
The goal of the scheduling experiments is to provide an initial analysis of the policies designed in Section 6.3. We are specifically interested in the impact of these policies on both the performance and the resource consumption of Fission Workflows. We evaluate the following policies: `horizon`, `prewarm-all`, and `prewarm-horizon`. To emulate the various levels of cold start (problem discussed in Section 6.3.1), we rely on the SimFaaS FaaS emulator, which allows us to set the cold start duration and how long the emulated function instances should be kept around (or warm) after the function execution. We set *keep warm* to 20 min, to ensure that function instances are not scaled down during the experiment. Because it is not known of the Chronos workload how many unique functions it includes, we assume—to also emphasize the impact of cold starts— that each task is a unique function requiring its own cold start.

We report two metrics for these experiments:

1. **Workflow runtime overhead** calculated by subtracting the optimal runtime of the workflow invocation from the actual runtime. In turn, the optimal runtime is computed as the sum of the task runtimes; the actual runtime is measured by Prometheus.

2. **Resource cost (rc)** emulated by SimFaaS as a single unit of cost, to resemble the resources used by functions. The resource model is simple: a function instance 'costs' *1 rc* per second that it is deployed (excluding the deployment process).

E. Real-world experiments
Whereas the other experiments focus on evaluating specific components or specific behaviour, in these experiments our goal is to evaluate the performance and cost of a realistic deployment of Fission Workflows. For this, we evaluate and compare in this experiment realistic configurations of Fission Workflows and the realistic alternatives offered by the major cloud providers:

1. **AWS**: We use AWS Step Functions, the WMS offered by the leading cloud provider AWS, to orchestrate the workflows. As the FaaS platform we rely on AWS Lambda, which is the FaaS platform of AWS. Each function in this setup is implemented as a JavaScript function, because it is the most used serverless language [120], with the lowest overhead [86].

2. **Google/Airflow**: We leverage for the workflow management, Google Cloud Composer, a (managed) Apache Airflow deployment within a Kubernetes cluster, and Google Cloud Functions for managing the individual FaaS functions. Although Airflow is not a serverless workflow engine—it is billed by the hour instead of per execution, regardless of whether workflows are executing—it is a very popular workflow engine, with more than 200 organizations relying on it [60]. Each function in this setup is implemented as a JavaScript function, because it is the most used serverless language [120], with the lowest overhead [86].

3. **Azure**: As a third realistic alternative, we choose the Microsoft Azure cloud. It is currently the second biggest cloud provider globally [112]. Within Azure Cloud, we use Azure Logic Apps for the orchestration of the workflows, and Azure Functions for the execution of the functions. Each function in this setup is implemented as a JavaScript function, because it is the most used serverless language [120], with the lowest overhead [86].

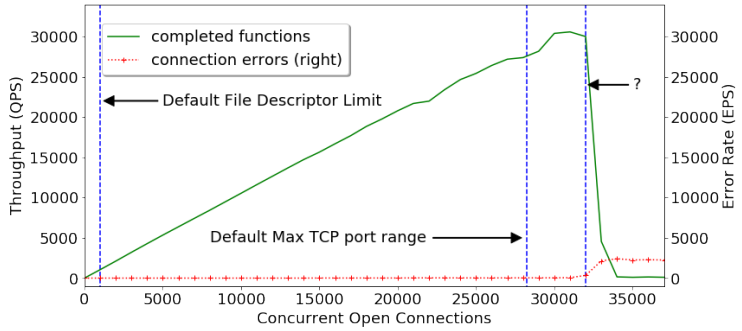
4. **Fission Workflows**: we deploy the Fission Workflows prototype on Kubernetes along with Fission for the FaaS functions. The function for the workflow is implemented as a Python function using the default Python environment. This is the typical setup of a common user of Fission.

We run repeatedly the Chronos workflow trace (see Section 8.1.2.2), as a representative serverless workflow workload, at least 3 times for each of these systems. Before each run, we run the trace once without measurements to warm up the systems. We ingest the workload as close as possible to the cluster, using the default way to ingest workflows (using the CLI for AWS and Fission, and the HTTP trigger for Azure). The only exception is Google Cloud Composer/Airflow, where the CLI crashes for the submission rate of the Chronos trace; for this setup only, we add the workflow invocations directly to the SQL database. This means that the results for the Google setup are underestimating its overhead, and are thus optimistic.

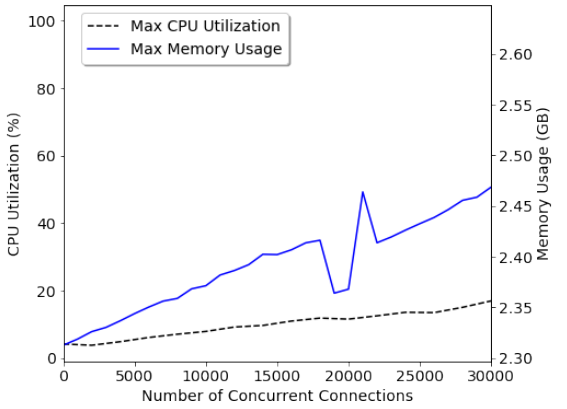
To measure the runtime of the workflows, we rely on the runtimes as reported by the systems themselves. The motivation for this is two-fold: (1) both Airflow and AWS Step Functions do not provide a way to execute workflows synchronously; and (2) the self-reported runtime excludes noisy factors, such as the network latency. We did verify manually that the self-reported runtimes are realistic and include the queuing taking place. This setup choice is consistent to what an expert client would do to monitor the performance of the system.

In contrast to how we measure performance, we calculate the cost of the experiments ourselves for two reasons: (1) the pricing models of these systems vary significantly, with various (incidental) discounts and free tiers which complicate the model and obscure the true cost of the experiment under general conditions; and, (2) by breaking down the costs ourselves, we gain more insight on the various factors affecting costs. Because we execute these experiments entirely within a cloud, we exclude network (e.g., ingress, egress) costs entirely. We also exclude data storage costs, monitoring costs, and other (typically optional) costs.

The full description of the configuration and code used for the realistic experiment can be found in Appendix C.



(a) The effects on throughput of an increasing number of concurrent, simulated 1-second function executions.



(b) Key metrics of the driver; none of them hits a resource limit during the experiment.

Figure 8.4: The impact of concurrent connections on the throughput of SimFaaS and the resource usage of the workload driver.

8.2. Experimental Results

In this section we present the results of the experiments described in the Experimental Setup (Section 8.1). We start by analyzing the performance of the SimFaaS FaaS emulator in Section 8.2.1, to verify that it will not impact the other experiments it is used in. Then, in the remainder of the this section we build up from discussing the internal performance (Section 8.2.3) and fault-tolerance experiments (Section 8.2.2), to initial scheduling experiments (Section 8.2.4), and real-world experiments (Section 8.2.5).

8.2.1. SimFaaS Experiments

Though extensive experimentation with the SimFaaS FaaS-platform emulator is out of scope, in this thesis, the goal of the SimFaaS experiments is to verify that its function execution overhead and resource usage are insignificant compared to the other components of the performance profile of the typical workflow invocation. By analyzing the levels of workload SimFaaS that can easily handle, we can ensure that the FaaS platform will not become the bottleneck in the further experiments. This knowledge allows us to use SimFaaS, instead of relying on complex, full FaaS platforms—which typically have performance quirks, such as the performance of different languages [86]. Towards this goal, we performed experiments in which we study the impact of an increasing number of concurrent function executions. Our key findings are:

1. On the `H2` hardware, SimFaaS can handle the emulation of at least 30,000 concurrent function executions, far exceeding the workloads expected to be handled by single FaaS-platform nodes, and thus in our experiments.
2. There is no resource constrained by SimFaaS during these experiments, though likely the CPU will be the initial bottleneck if the experiments would scale further.
3. The underlying operating system of both the host and driver node needs to be tuned carefully to be able to increase the supported concurrency beyond an initially (low) limit.

We support these findings by analyzing the SimFaaS experiments from three perspectives: (1) the impact on the function runtime overhead; (2) the impact on the resource usage; and (3) the impact on the operating system.

8.2.1.1. Runtime Overhead

The first goal of this experiment was to analyze how much overhead the SimFaaS emulator has on function executions. For the runtime overhead, we find that:

- The throughput (QPS) grows linearly with the number of concurrent connections.

- Aside from a few incidental function executions, the runtime overhead does not unfairly impact parts of the function executions.

Figure 8.4a depicts how the workload of concurrent requests for function executions arriving at SimFaaS impact the throughput. Until 30000 connections, we observe the QPS rising near-linearly with the level of concurrency. The reason for this near-linear relationship is that the function execution each take 1 second. With this setup, in the ideal case, the concurrency of N concurrent, 1-second function executions, should directly translate to N QPS. The results in Figure 8.4a suggest that the SimFaaS does not get congested at the workload ranges used.

In the runtime-part (first row) of the 15-part Figure 8.5, we observe that the function runtime overhead remains stable as the throughput, and pressure on the system, increases. Even at 30000 QPS, the distribution (see Figure 8.5b) of the runtime overhead remains stable. For instance, even the 90th percentile of the distribution barely increases after 10000 QPS. At the extreme percentiles (99th and 100th percentiles in Figure 8.5c), there is an increase in runtime overhead, resulting in near near-second of latency at 30000. Since the TCP connection handling is a major part of the computation effort, the extreme runtime overheads can likely be attributed to delays in the TCP handshake. For instance, at high throughput the random arrival times of many concurrent connection requests might get queued in the TCP backlog. Further analysis of these extremes should be done to investigate the true causes of the runtime overheads in the extreme percentiles—something that is beyond the scope of this work. Overall, both the throughput and the function runtime overhead show that SimFaaS can handle at least 30000 concurrent connections on the $H2$ hardware.

8.2.1.2. Resource Consumption

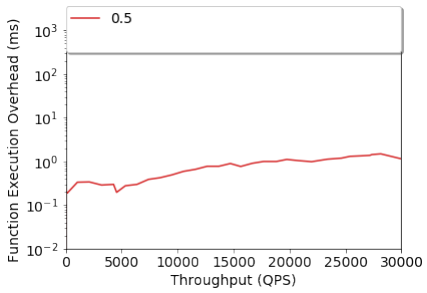
The second goal of the SimFaaS experiments was to see if the SimFaaS system does consume excessive resources, which in turn could impact the performance of other components in other experiments. Key findings:

- CPU usage grows linearly with the throughput, and is likely the constraining resource in deployments of SimFaaS.
- Memory usage grows linearly with the QPS at approximately 20 KB for increment in the QPS.
- Median incoming and outgoing network throughput grow by 0.3 KB, and 0.1 KB respectively for every increase in QPS, but become more jittery at high throughput.
- The workload driver is not constraining the experiment.

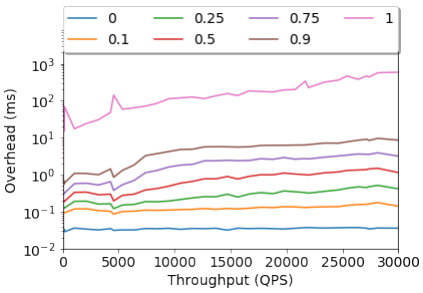
Next to runtime overheads, the 15-part Figure 8.5 depicts key resource usage metrics. The CPU-part (second row) of the Figure 8.5 shows that the CPU increases linearly with the throughput. Although the distribution of the CPU utilization in Figure 8.5e) does not indicate that the CPU is constraining the throughput at 30000 connections yet, extrapolating the trend and observing the extremes already hitting 100% CPU utilization, the CPU is likely to become the constraining resource at even higher throughput. As identified in the analysis of the runtime overhead, a likely major contributor to this CPU utilization increase is the TCP connection handling within the FaaS platform.

Third row in Figure 8.5 shows that, in contrast to CPU, memory usage is unlikely to be a constraining resource. The resource usage follows a predictable linear trend, consuming roughly 20 KB for each QPS. Since the QPS, in this experiment, is equivalent to concurrency, this means that the memory costs of the TCP connection and the associated entities in SimFaaS comprise of roughly 20 KB. The implication of this is that machines used for high-throughput experiments should reserve 1 GB for SimFaaS to fully ensure that SimFaaS does not get memory-constrained at extremely high throughput.

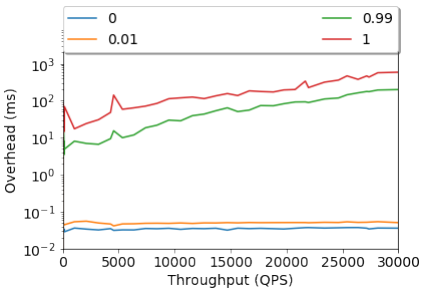
The fourth and fifth rows in Figure 8.5 show the incoming and outgoing network throughput of SimFaaS. The initial observation is that the network throughput is not constraining the throughput, with the median incoming bandwidth growing with 0.3 KB/s per increase in QPS and the median outgoing bandwidth growing with 0.1 KB/s per increase in QPS. The driver and worker machines set up within the same datacenter, which enables these machines communicate with each-other over the datacenter network infrastructure—a link with 1 GB/s bandwidth. Another observation form Figure 8.5 is that the network bandwidth becomes jittery at high throughput. The extremes, and at higher throughput even



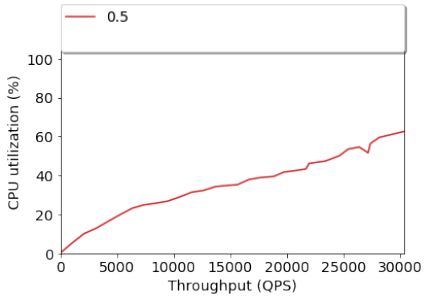
(a) Trend of function runtime overhead.



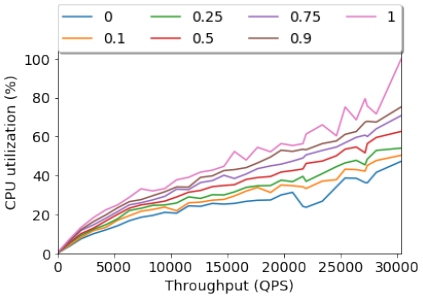
(b) Distribution of function runtime overhead.



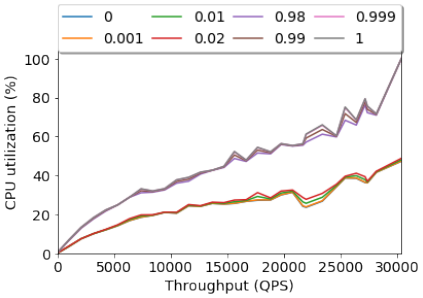
(c) Extremes of function runtime.



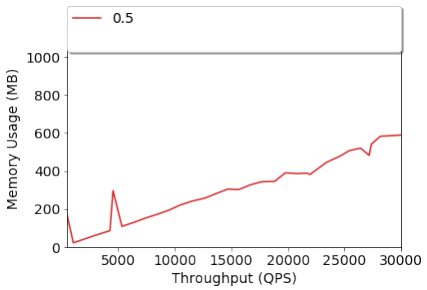
(d) Trend of CPU utilization.



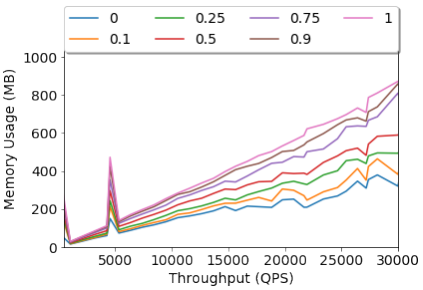
(e) Distribution of CPU utilization.



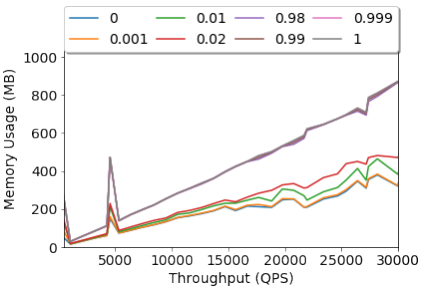
(f) Extremes of CPU utilization.



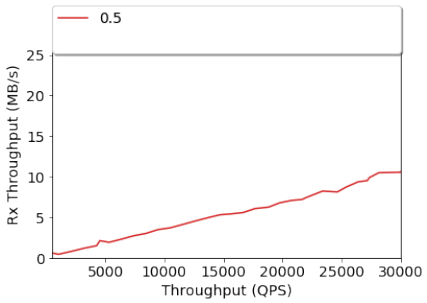
(g) Trend of memory usage.



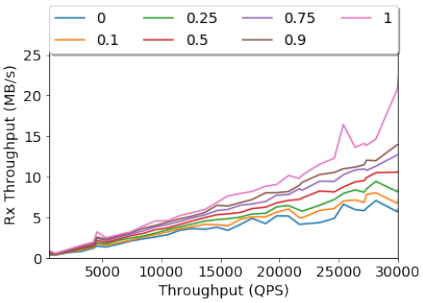
(h) Distribution of memory usage.



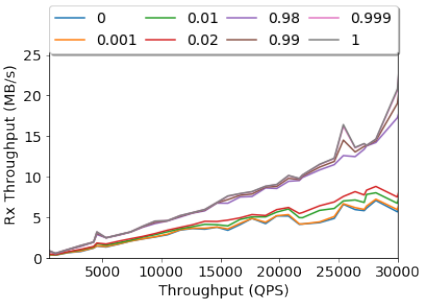
(i) Extremes of memory usage.



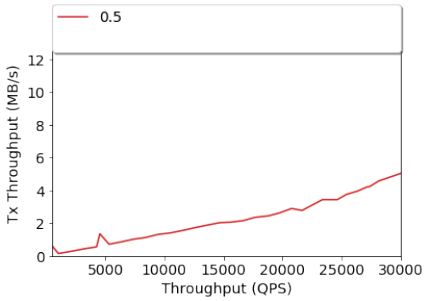
(j) Trend of incoming network throughput.



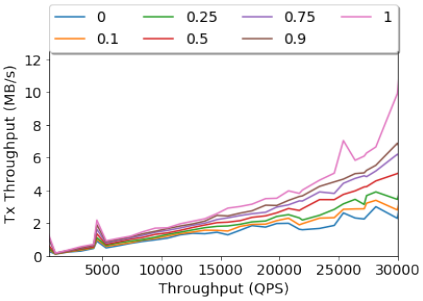
(k) Distribution of incoming network throughput.



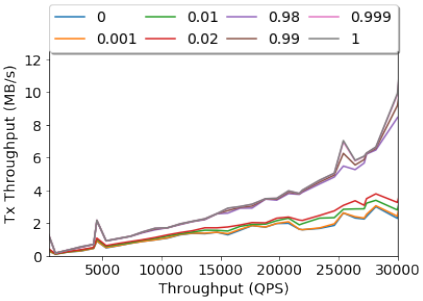
(l) Extremes of incoming network throughput.



(m) Trend of outgoing network throughput.



(n) Distribution of outgoing network throughput.



(o) Extremes of outgoing network throughput.

Figure 8.5: The impact of the throughput on the user and system-level metrics of a single node deployment of the SimFaaS simulator. For the distribution profiles, curves depict the following percentiles (`0`) the zeroth percentile, (`0.01`) the 1st percentile, (`0.1`) the 10th percentile, etc.

the 75th and 25th percentiles, diverge. This indicates that there are certain spikes in network-related activity. For instance, this suggests that batching of TCP packets is taking place.

Finally, Figure 8.4b depicts the key metrics for the driver machine of this experiment. It shows that neither the CPU nor memory is constraining the driver, even at the extremely high-throughput of function executions. This ensures that we can rely on the driver to maintain the concurrent workload to the worker machines.

8.2.1.3. System Tuning

SimFaaS, like any modern application, relies on many existing technologies and systems (such as the operating system, networking stack, and HTTP server implementation). This means that we cannot evaluate SimFaaS without understanding the effects typical tuning of these technologies and systems may have on performance. During early iterations of the SimFaaS experiment, default configurations and limits throughout these underlying technologies limited the maximum concurrency—we were observing limited performance from SimFaaS, although the internal counters did not report anything unusual in our emulator. We report here the results of our deeper investigation into the root causes of performance limitation in the combined system formed by SimFaaS and the underlying OS. Our key findings:

- The default file descriptor limit is rather low, which risks even small experiments to be constrained by it.
- The dynamic TCP port range is the next limit, limiting a machine from making more than 28,231 concurrent connections on a single network interface.
- Further (unknown) operating system limits seem to constrain the experiment in its current form.

First, in Figure 8.4a at around a 1000 QPS we hit the default open file descriptor limit of the worker node. As in many Linux distributions, the default in our node was set rather low—to 1024. Because the workload driver does not make use of advanced (HTTP2) functionality, such as TCP connection multiplexing, the concurrency of the function executions corresponds directly to the number of TCP connections made. By setting the open file descriptor limit to unlimited, SimFaaS easily service beyond 1024 concurrent connections.

Second, around 28,000 concurrent function executions, as depicted in Figure 8.4a, the experiment hits the following system-level constraint: the client node exhausts its available TCP ports for connections. By default, of the 16-bit port range (65,535 ports), only the ports between 32,768 and 60,999 are available for (dynamic) TCP connections. This means that once all 28,231 ports are all used for TCP connections, any further incoming connections will eventually fail—after first filling up the TCP backlog. As long as no other applications are occupying ports, we can safely increase the ports available for dynamic TCP connections to 64,511—as generally only the first 1024 ports are restricted. Note that, an alternative approach here is to use a different network interface to increase the number of TCP ports available. However, since our workload drivers are all build assuming a single network interface, we did not pursue this approach.

Then, at roughly a concurrency level of 32000, Figure 8.4a indicates that the experiment hits another operating system limit. The reason that this seems to be an operating system limit, is that there are no clearly constrained resources. Looking at the increasing error rates at concurrency levels larger than 32000 we suspect that we are exceeding another limit or the bump in the dynamic TCP port range described in the previous paragraph causes unexpected side-effects. However, the scope of this experiment is limited—the goal of these experiments were to ensure that SimFaaS does not introduce performance degradation or variability for the other experiments of the workflow engine. At 30000 QPS we have far exceeded the expected workload for the WMS. If we would expend further effort in this space, we expect that we can increasing the limits of this experiment, as we expect that there are further system-level barriers of the underlying systems that prevent us from pushing the concurrency limit even higher.

8.2.2. Fault-Tolerance Experiments

The goal of the fault-tolerance experiments is to asses the resiliency of the workflow engine in the face of failures. Towards this goal we perform three sets of experiments: in the first experiment (Figure 8.8) we

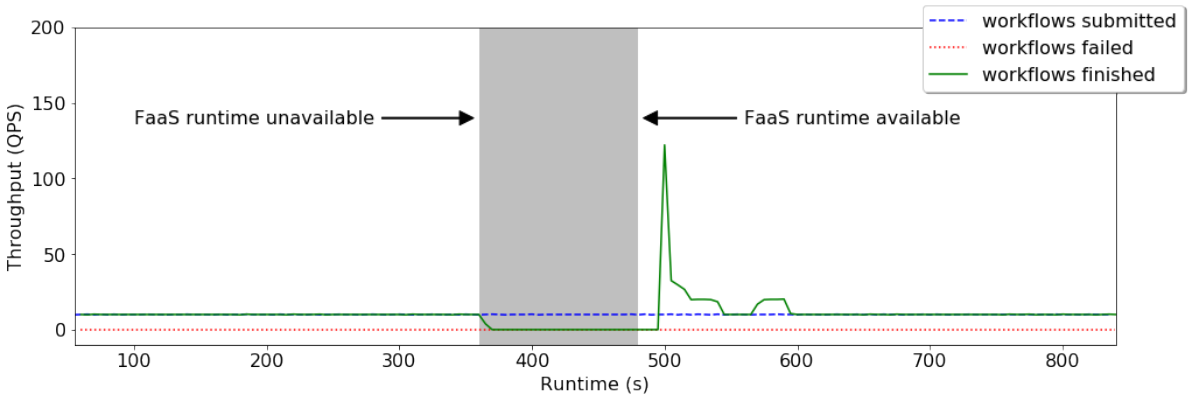


Figure 8.6: The impact of temporary unavailability of the FaaS platform on the workflow engine.

look at the impact of a fail-stop failure on different configurations of the workflow engine; in the second experiment (Figure 8.6) we emulate an intermittent network failure between the workflow engine and the FaaS platform; in the third experiment (Figure 8.7) we analyze the scenario in which there is a period of network unavailability between the workflow engine and the NATS-backed event store. Our key findings are:

1. The prototype can execute workflows in the face of intermittent unavailability of the underlying FaaS platform.
2. The availability of the event store is critical to the overall availability of the workflow engine.
3. The configuration options for the architecture of Fission Workflows provide flexible availability guarantees of the overall WMS.

8.2.2.1. FaaS Platform Unavailability

The goal of this experiment is to evaluate how the WMS copes with temporary unavailability of the FaaS platform. Our key findings:

- The prototype employs periodic retries to overcome temporary unavailability of the FaaS platform.
- Workflows and workflow invocations will not fail due to the FaaS unavailability (unless they are submitted through the FaaS platform).

As observed in Figure 8.6, no workflow executions fail during the period of FaaS unavailability. Instead, the workflow retries submitting function executions for a certain period. The function executions are queued during the unavailability starting at minute 6, and executed once the FaaS platform becomes available after the minute 8. The resolving of the queue of function executions can be observed in Figure 8.6, which happens right after the FaaS runtime becomes available again.

Whether new workflow invocations can be submitted to the workflow engine depends on the deployment configuration. When users are executing the workflows as functions through the FaaS platform—serverless workflows act like any other serverless function after all (see Section 5.1)—the workflows can only be submitted after the connection between the workflow engine and the FaaS platform is restored. However, as shown by the blue submission line in Figure 8.6, in this scenario we are submitting workflows directly to the workflow engine, which are therefore unaffected by the unavailability of the FaaS platform.

8.2.2.2. Event Store Failures

The goal of this experiment is to analyze the effects of event store unavailability on the availability of Fission Workflows. We consider here only the scenario when the event store is external to the WMS—since an internal, in-memory event store cannot become temporary unavailable to the WMS. The key findings for this experiment:

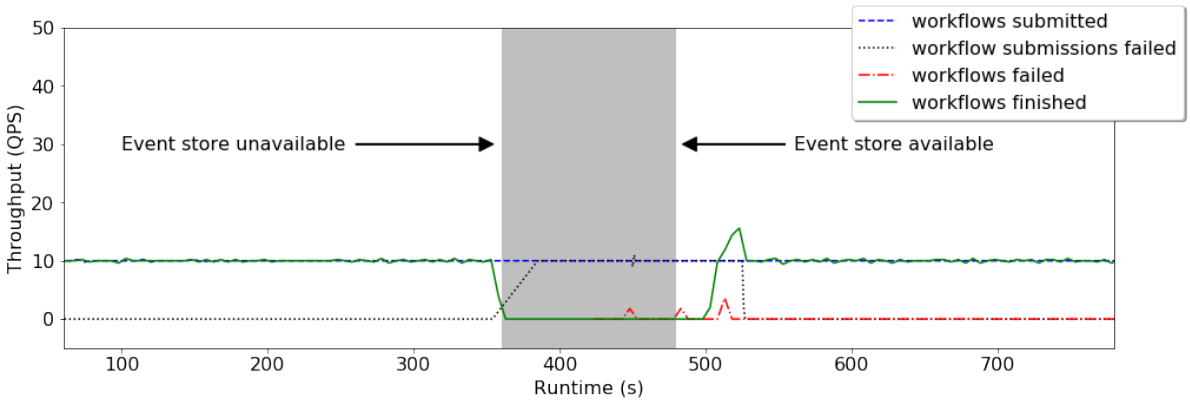


Figure 8.7: The impact of temporary unavailability of the event store on the workflow engine.

- During unavailability of the event store, no new workflows or invocations can be added.
- Workflow invocations already persisted are recovered when the event store is available again.
- Workflow invocations cannot progress when the event store is unavailable.

Figure 8.7 shows that no new workflow invocations can be submitted to the workflow engine. The reason behind this is that to submit a workflow invocation it must be persisted as an event to the store. Although Fission Workflows could be improved by for example caching these workflow invocations while awaiting the event store to restore or even start executing the workflows without having persisted the events, in general the WMS chooses to sacrifice availability to avoid network partitions.

Similarly, no workflow invocations can progress; no tasks can be executed. This is shown in Figure 8.7 by the lack of any workflows being completed during the unavailability of the event store. This halting of progression is by design; as mentioned before, the workflow engine sacrifices availability to avoid inconsistencies and network partitions.

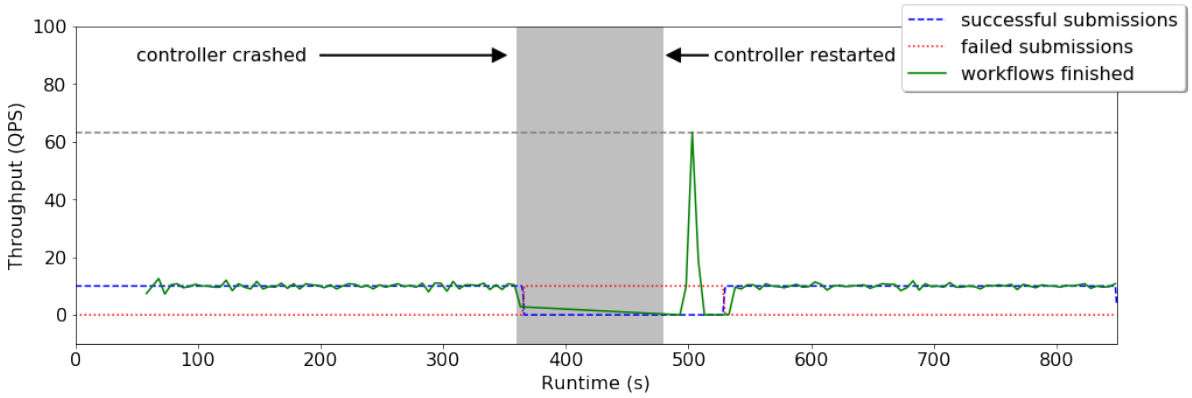
Once the connection is restored to the event store, the controller will synchronize its state with the event store (assuming that the event store did not crash irrecoverably). After this recovery period, the workflow engine will continue where it left of with the active workflow invocations, which is the cause of the spike in completions in Figure 8.7.

8.2.2.3. Deployment Configurations

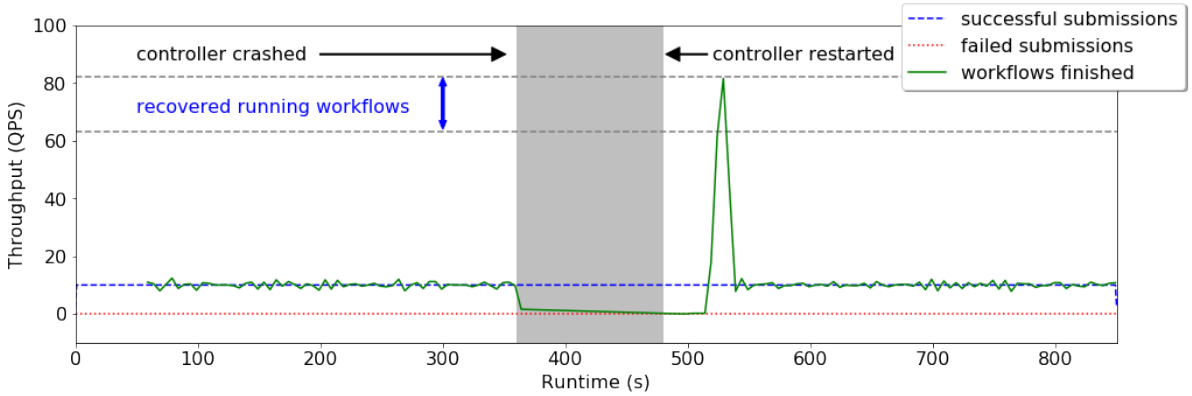
The goal of this experiment is to compare the impact of different deployment configurations on the fault-tolerance of the WMS. The reason for this comparison is that Fission Workflows has a flexible deployment model. It can be configured to run with all components bundled into one process, or in a distributed topology. Moreover, the operator can decide to deploy it with an in-memory event store or an external event store. The choices that the operator makes in this deployment configuration affect the availability that the system can provide. Our key findings:

- The choice of the even store implementation is large factor in the availability guarantees that the WMS can provide, with the in-memory event store providing the least guarantees.
- A distributed deployment of a WMS offers better availability guarantees than a bundled deployment.
- Fission Workflows does not require all components to be available; it can operate without Prometheus or Jaeger being available.

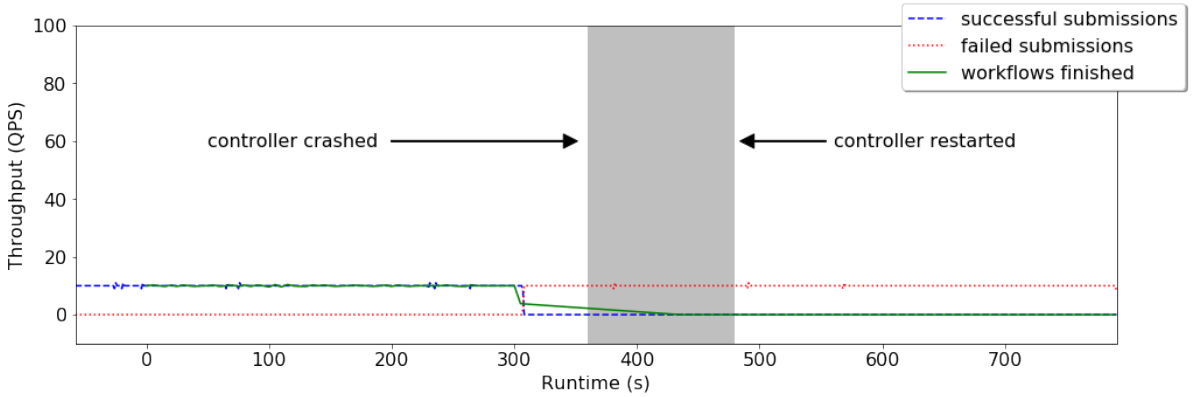
As observed in Figure 8.8, having an external event store—in this case NATS Streaming—allows the workflow engine to recover workflow invocations when it is restarted. In contrast, Figure 8.8c shows that using the in-memory event store, means that in case of a fail-stop the all invocations, completed or running, are lost on a restart. Not only the invocations, but also the workflows are lost, which, as Figure 8.8c depicts after the workflow engine restarts, means that unless the lost workflow is resubmitted, no new invocations can be submitted. Based on this observation, it seems that for the availability



(a) Impact of a fail-stop failure on the bundled workflow engine with a NATS-backed event store.



(b) Impact of a fail-stop failure on distributed workflow engine with a NATS-backed event store.



(c) Impact of a fail-stop failure on a distributed workflow engine with an in-memory event store.

Figure 8.8: Run graphs of a fail-stop failure on three different architectures of the workflow engine.

guarantees the choice of the event store implementation to use is likely one of the most important configurations that the operator of Fission Workflows can make. While an in-memory event store is lightweight and low-latency, the availability guarantees of the external, NATS-based event store are better.

Next to the event store configuration, the WMS operator can influence the availability of the system by using a *bundled* or a *distributed* setup. In the bundled setup all components (including the controller, scheduler, and apiserver) are all deployed in a single process. This simplifies the deployment process—it is typically used in the development environment for developing and testing workflows—but it comes at the cost of availability. In Figure 8.8a, a fail-stop of the bundled workflow engine results not only in halting currently executing workflows, but also prevents any new workflow invocations. In a distributed setting, depicted in Figure 8.8b, we separate the apiserver (the component responsible for handling submissions) from the controller, which allows workflow invocations to be submitted even when the controller is unavailable. The implication of this difference in deployment configuration is that both serve different use cases. The bundled setup is, a single process, is simple to deploy, which makes it more suitable for development and simple use cases. In contrast, given the better availability and recovery mechanism shown in Figure 8.8b, the distributed setup seems to be preferable in business-critical and production use cases.

Finally, the workflow engine has optional components that are not required to be present for any functionality of the workflow engine. In the prototype, these components include Prometheus (for metric collection) and Jaeger (for distributed tracing). In fact, the experiments in this section (Figure 8.8a, and 8.8b) were performed without these components.

8.2.3. Scalability Experiments

The scalability experiments aim to offer a look at how the Fission Workflows prototype scales in terms of performance and resource usage under an increasing load. The measurements are based on experiments focused on evaluating the performance of a specific component or a specific workload characteristic. Based on these experiments, we found that:

1. The performance difference between event store implementations can be more than two orders of magnitude.
2. In the experiments, CPU is generally the performance bottleneck, and the unoptimised prototype is further wasteful in memory consumption.
3. The prototype easily handles large and long-running workflows, but struggles with a high number of parallel tasks, even if each task is small.

8.2.3.1. Performance Evaluation of Event Store Implementations

The goal of this experiment is to compare the performance of different event store implementations. For this we consider the in-memory event store, as well as several variations on the NATS Streaming event store: NATS with SQL database persistence, NATS with filesystem persistence, and NATS without persistence. Our key findings:

- There is an order of magnitude performance difference in performance between in-memory NATS and the in-memory event store.
- The performance varies by an order of magnitude between the best performing NATS event store (in-memory) and the worst performing (SQL).

Figure 8.9 shows the difference between an internal, in-memory event store (*memory*) and an external, in-memory event store (*NATS*). The difference between these modes of implementation is clear: the internal event store propagate most events in under a millisecond, while an external event store processes over half the events in about 1 ms or worse (the median is just below 10ms). The reason for this is that, even though both are in-memory the NATS event store can provide better guarantees on the persistence of the data. Within the NATS cluster the data can be stored on multiple nodes, ensuring high-availability.

Not only is there a difference between an internal event store and an external event store, even for the NATS Streaming event store the configuration can have a magnitude difference in performance. In

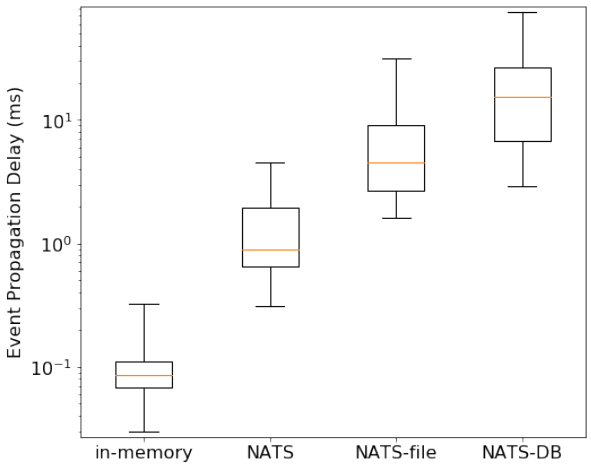


Figure 8.9: A comparison of the impact of event store implementations on the event propagation, based on the Chronos workload.

Figure 8.9, this impact can be seen in the three right-most box plots. The `NATS` option, that is, NATS Streaming without any persistence, can generally process events within a one or couple of milliseconds. However, when we enable the file-based (`NATS-FILE`) or the SQL database (`NATS-DB`) persistence in NATS Streaming, the time to propagate an event increases to over 10 ms on average. The difference can be explained by in part by an observation on the project: the database implementation of NATS is relatively new, and less mature. This could explain the lack of performance optimizations. Additionally, there is a dissonance in the optimization options. Whereas with the file-based persistence there are little optimizations to be done, with any mature SQL database various performance optimizations can be tried. The fact that the NATS-DB in Figure 8.9 is a default PostgreSQL database without any performance optimizations, make it seem likely that it can be tuned to be on par with the `NATS-FILE` alternative.

8.2.3.2. Workflow Ingestion

The goal of this experiment is to analyze the throughput capacity for ingesting (or submitting) workflows into a Fission Workflows deployment. The motivation for this analyzes is to ensure that, like the analysis of SimFaaS, the ingestion of workflows does not become the bottleneck in other (scalability) experiments. Our key findings:

- The WMS can handle the submission of at least 5000 workflow invocations per second, where the 99.9th percentile of the submissions happens under 1 second.
- Both network and memory show a linear relation with the submission rate.

Figure 8.10 shows that the trend in the submission latency[8] remains reasonably stable until roughly 9000 QPS, after the median submission latency shoots up to taking multiple seconds. However, as depicted in the distribution of the submission latency of Figure 8.10, for the 75th percentile the latency spikes already at 5000 QPS, and for the 90th percentile even earlier. This performance degradation can be explained by looking at the CPU graphs in Figure 8.10. CPU contention, at the extremes, starts already at 6000 QPS, which is an indicator that the submission process is CPU-bounded. The reason for this is likely that the invocations undergo a series of checks, and transformations before being persisted as an event.

In contrast the memory and network graphs of Figure 8.10 do not seem to be constraining the submissions. Both metrics follow a linear relation to the submission rate. The memory usage, as we will see in other performance experiments, is a bit inefficient due to the immaturity of the project. However, despite the memory inefficiency it does not impact the overall performance.

[8] the submission latency is the time it takes from the user sending the workflow invocation to the user receiving a confirmation that the invocation has been submitted sucessfully.

8.2.3.3. Workflow Invocation Throughput

The goal of the workflow invocation throughput experiment is to analyze what the limits of the Fission Workflows prototype with regards to an increasing throughput of workflows. Our main findings:

- The WMS currently has a throughput of about 700 QPS.
- The constraint seems to be a scaling issue, rather than a resource constraint.

The makespan-row of the 15-plot Figure 8.10 shows that the WMS can handle throughput well until roughly 700 QPS. The memory grows linearly with the throughput as expected. This holds also true for the network throughput. Even before the 700 QPS mark it is clear that CPU will be the most constraining resource.

After that the 700 QPS makespan spikes, along with the CPU utilization. The cause of this relatively low limit 700 QPS is likely a stability or scalability issue in the source code of the prototype. A hint for this can be seen in the outgoing network throughput in Figure 8.10. At the same moment as the spike in the CPU utilization and the makespan the network throughput drops to near-zero. This indicates that the WMS no longer communicates with the FaaS platform; it does no meaningful work anymore. Further engineering will need to happen to resolve this performance issue.

8.2.3.4. Serial Workflows

The intent of this experiment is to evaluate how the WMS deals with longer-running workflows. Even though these workflows are not resource-intensive, they do occupy resources for a longer time, which could expose memory leaks and other inefficiencies. Based on the experiment, we found the following:

- The prototype has scales well for serial workflows, for the used workload range.
- The inefficient memory usage seems to be the constraining factor at some point for larger serial workflows.

Figure 8.12 shows that for the structure of the workflows being executed we find that duration of the workflow and workflow length does not cause workflow invocation slowdown. Figure 8.12a supports this, showing that the runtime of the workflow grows linearly with the number tasks in the serial workflow—where each task runs for 100ms. We can further observe in Figure 8.12b and Figure 8.12c, there is little slowdown even for large workflows (of over 900 tasks).

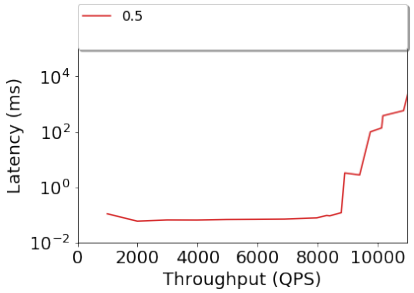
Compared to other experiments, in the serial workflow experiments not the CPU, but the memory usage will likely be the constraining resource were we to extend the range of the experiment. The curve in the memory-row of the Figure 8.12 seems to indicate an exponential growth in the memory usage. There are several explanations for this: (1) the WMS caches aggressively, with much future work left in finding a better balance between aggressive and pessimistic caching; (2) the invocation object always contains both the specification and the status of all tasks. These two causes are both not fundamental to the workflow engine's design and can be remedied in follow-up work.

8.2.3.5. Parallel Workflows

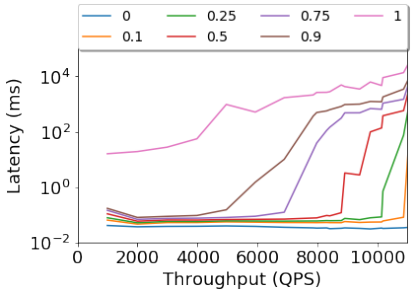
The goal of the parallel workflows experiment is orthogonal to that of the serial workflow experiment: to analyse the behaviour of the prototype with regards to increasingly larger, embarrassingly-parallel workflows. These workflows contain solely tasks that can be executed completely in parallel. Our key findings:

- The makespan of parallel workflows grow exponentially.
- The current implementation of event sourcing adds to the inefficient memory usage and adds to the CPU pressure.

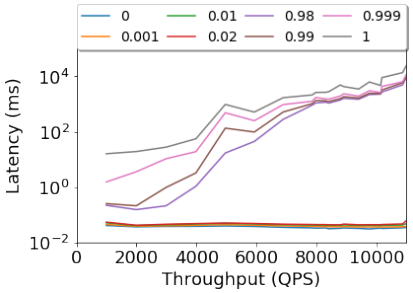
Figure 8.13 shows that the makespan of the parallel workflow invocations grows exponentially. As shown in Figure 8.13a, the workflow invocation slowdown grows steadily at higher levels of parallel tasks—even though, due to the complete parallel nature of the workflows, the theoretical minimal runtime remains at 1 second. The low variance in the execution delay (Figure 8.13b and Figure 8.13c) suggests that this slowdown is inherent the workflow structure, rather than due to queue build ups. Part of this slowdown is unavoidable, because each individual task has execution overhead and produces



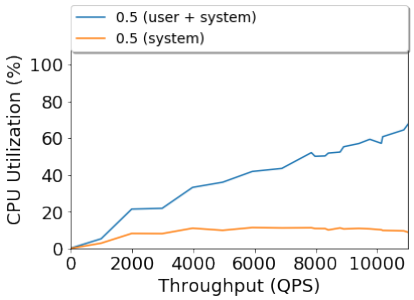
(a) Trend of submission latency.



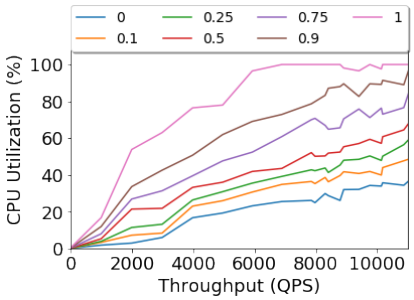
(b) Distribution of submission latency.



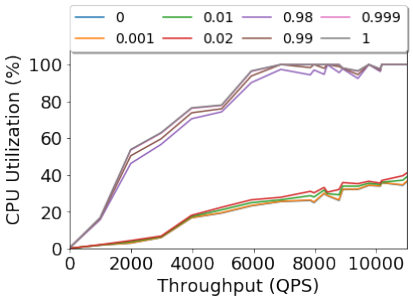
(c) Extremes of submission latency.



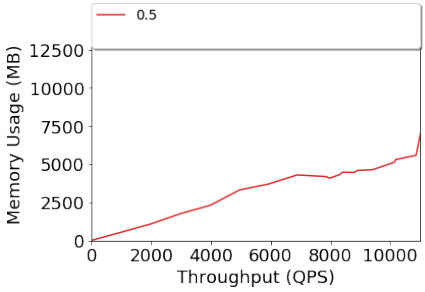
(d) Trend of CPU utilization.



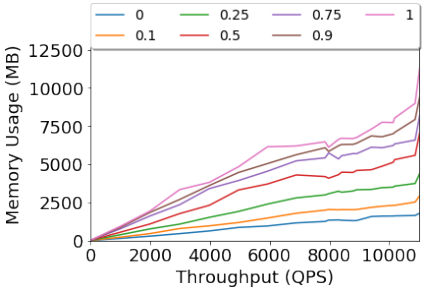
(e) Distribution of CPU utilization.



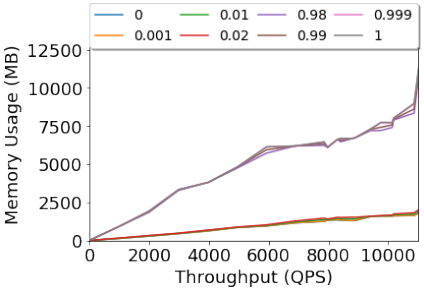
(f) Extremes of CPU utilization.



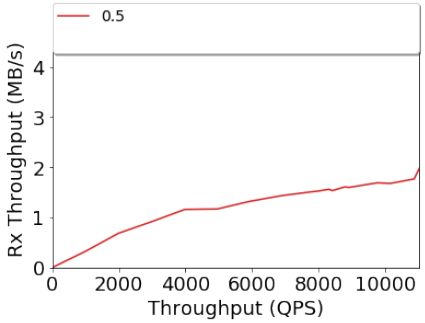
(g) Trend of memory usage.



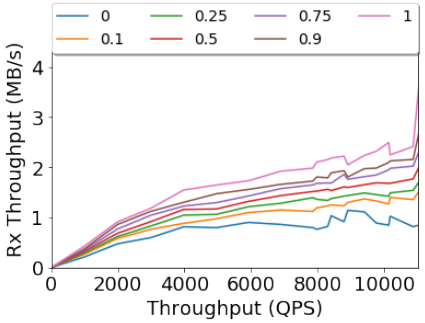
(h) Distribution of memory usage.



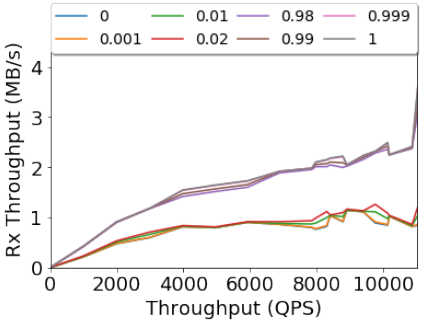
(i) Extremes of memory usage.



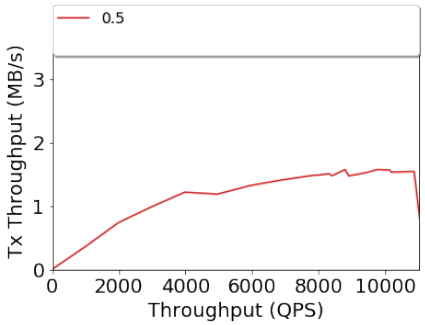
(j) Trend of incoming network throughput.



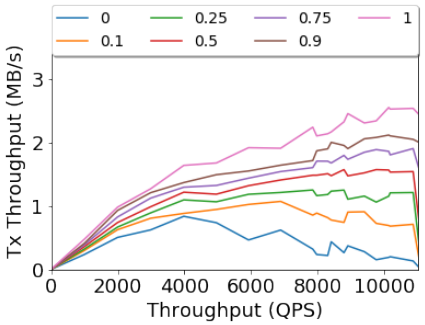
(k) Distribution of incoming network throughput.



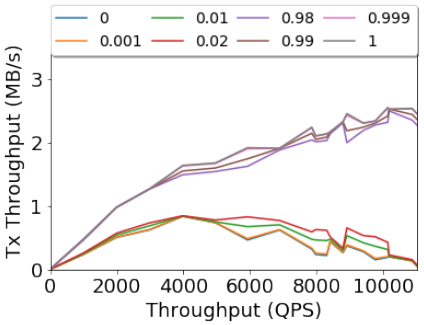
(l) Extremes of incoming network throughput.



(m) Outgoing network throughput trend.

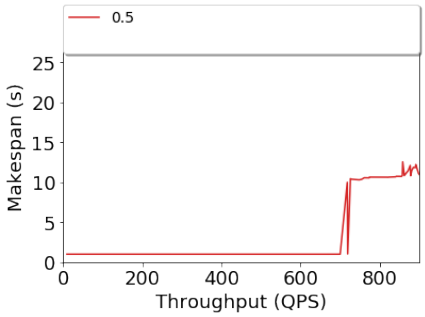


(n) Distribution of outgoing network throughput.

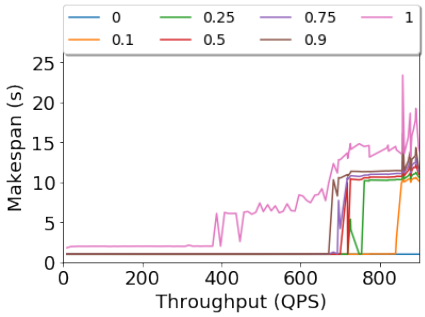


(o) Extremes of outgoing network throughput.

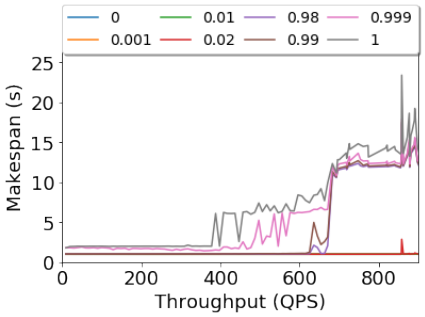
Figure 8.10: The impact of varying levels of load on the workflow invocation submission process. For the distribution profiles, curves depict the following percentiles (`0`) the zeroth percentile, (`0.01`) the 1st percentile, (`0.1`) the 10th percentile, etc.



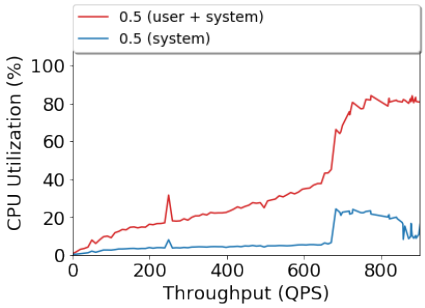
(a) Trend of makespan.



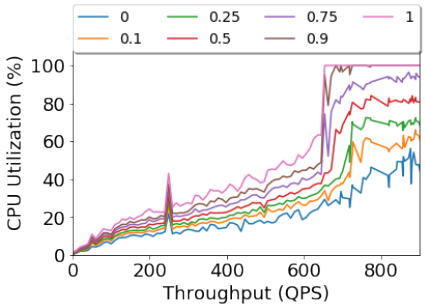
(b) Distribution of makespan.



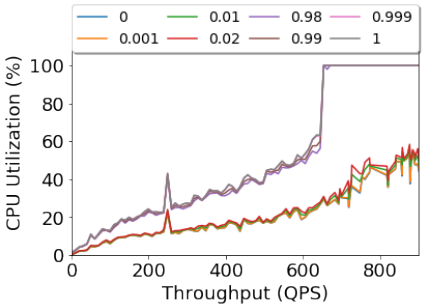
(c) Extremes of makespan.



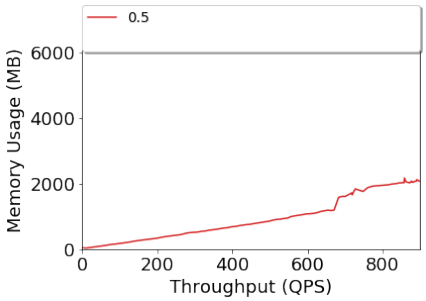
(d) Trend of CPU utilization.



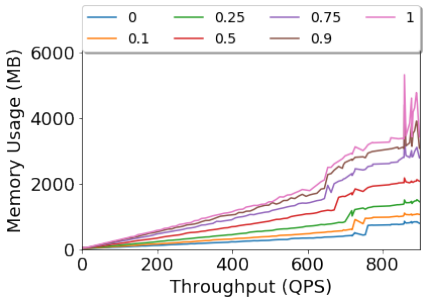
(e) Distribution of CPU utilization.



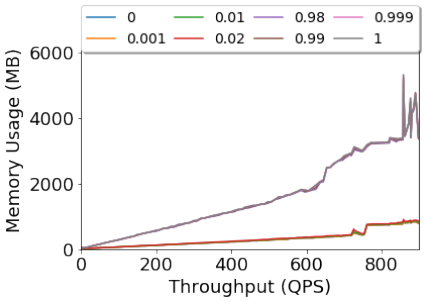
(f) Extremes of CPU utilization.



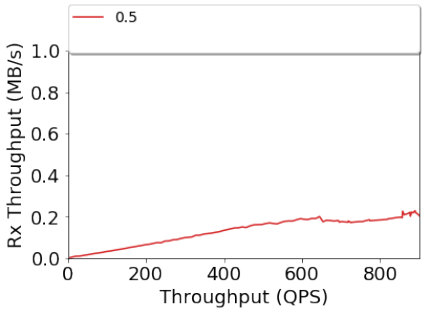
(g) Trend of memory usage.



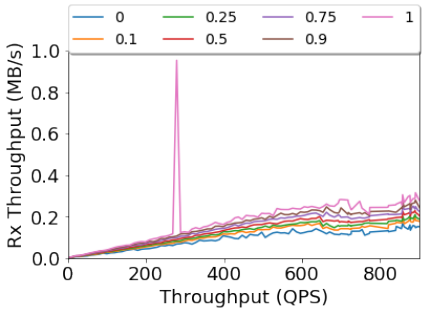
(h) Distribution of memory usage.



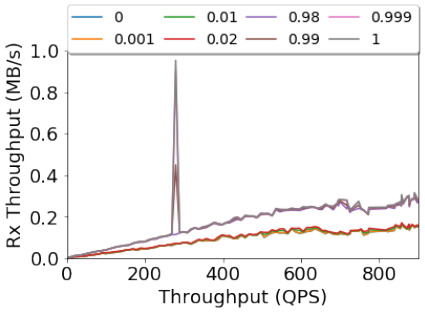
(i) Extremes of memory usage.



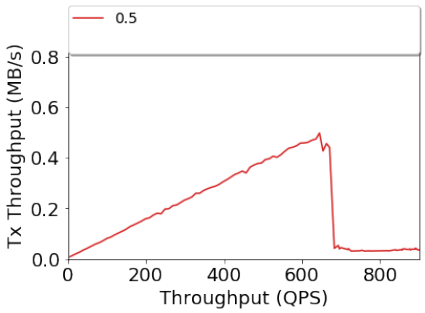
(j) Trend of incoming network throughput.



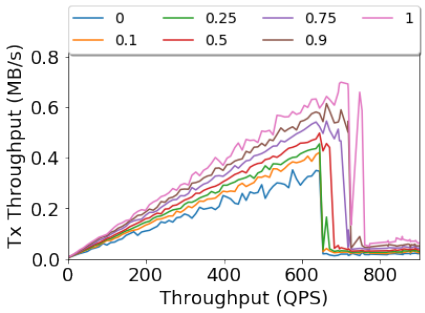
(k) Distribution of incoming network throughput.



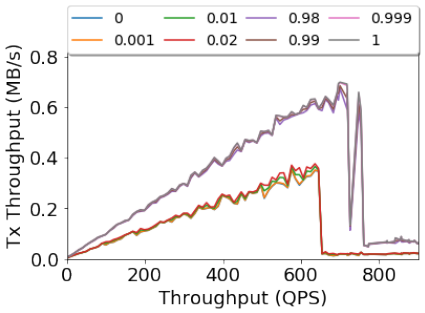
(l) Extremes of incoming network throughput.



(m) Trend of outgoing network throughput.

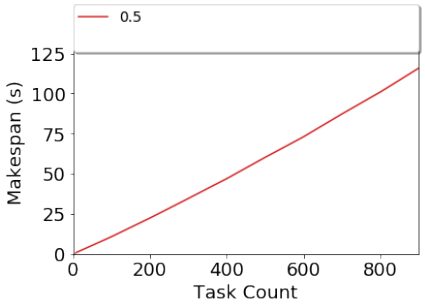


(n) Distribution of outgoing network throughput.

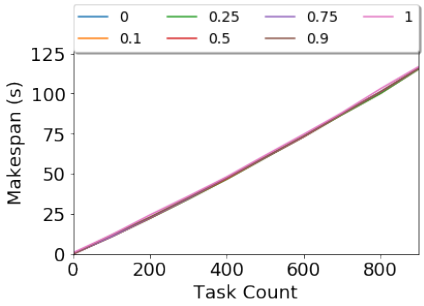


(o) Extremes of outgoing network throughput.

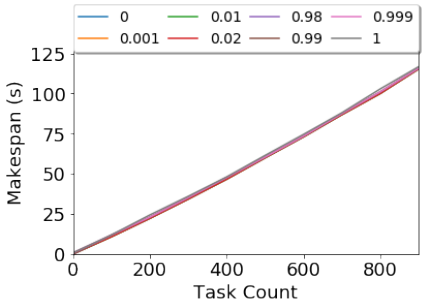
Figure 8.11: The impact of varying levels of load on the execution of a 1-task workflow. For the distribution profiles, curves depict the following percentiles (`0`) the zeroth percentile, (`0.01`) the 1st percentile, (`0.1`) the 10th percentile, etc.



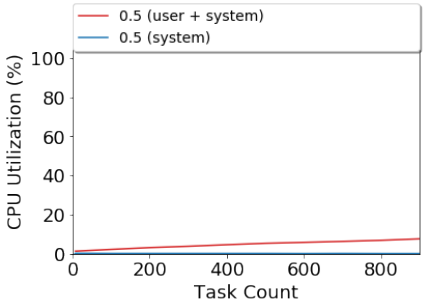
(a) Trend of makespan.



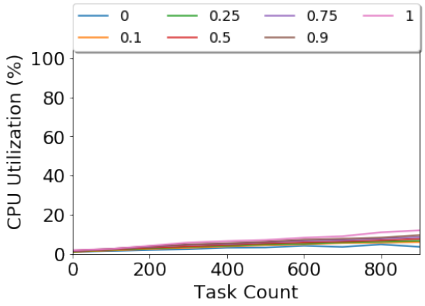
(b) Distribution of makespan.



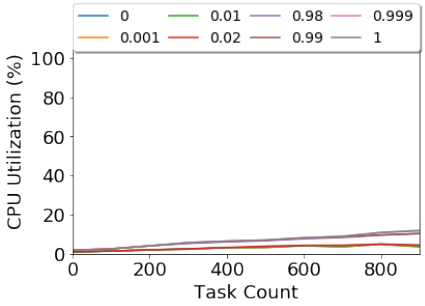
(c) Extremes of makespan.



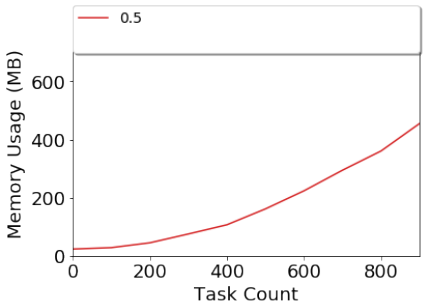
(d) Trend of CPU utilization.



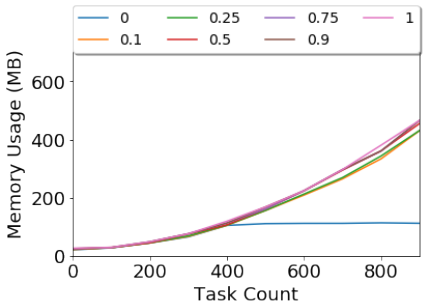
(e) Distribution of CPU utilization.



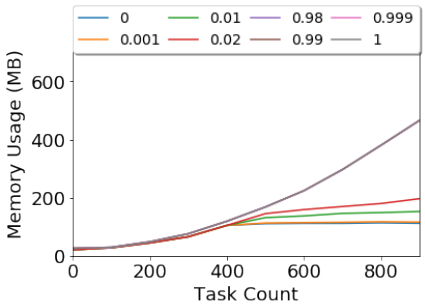
(f) Extremes of CPU utilization.



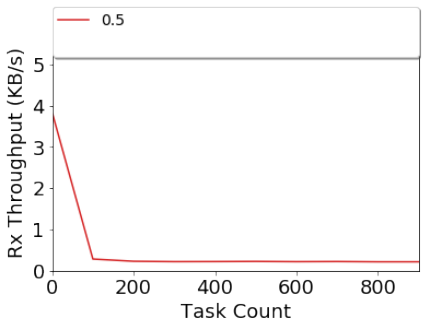
(g) Trend of memory usage.



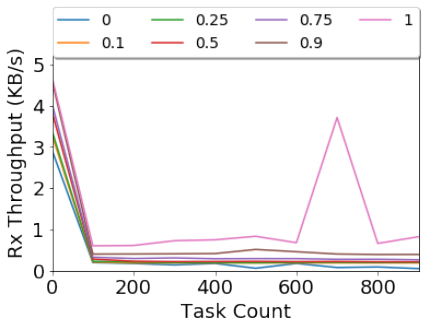
(h) Distribution of memory usage.



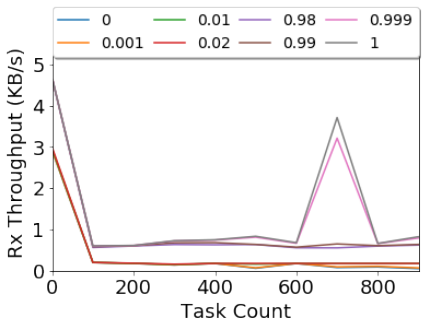
(i) Extremes of memory usage.



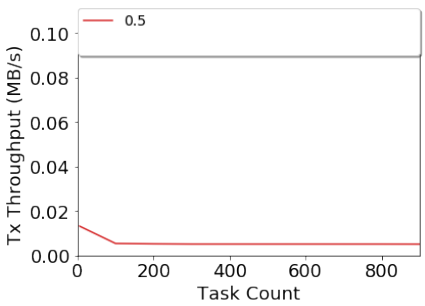
(j) Trend of incoming network throughput.



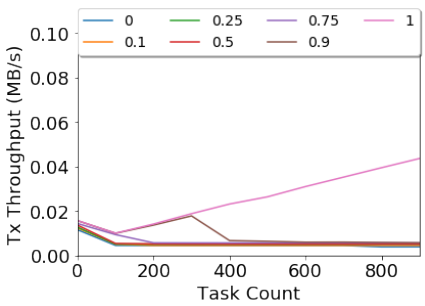
(k) Distribution of incoming network throughput.



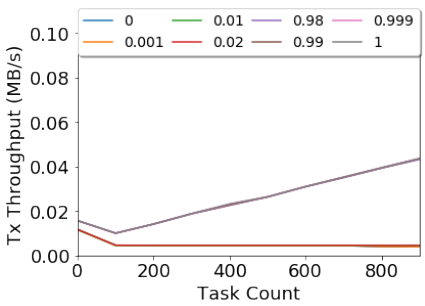
(l) Extremes of incoming network throughput.



(m) Trend of outgoing network throughput.

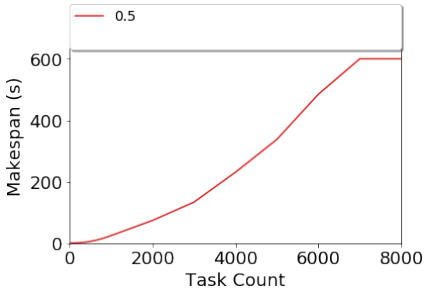


(n) Distribution of outgoing network throughput.

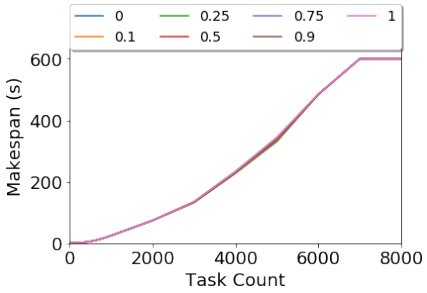


(o) Extremes of outgoing network throughput.

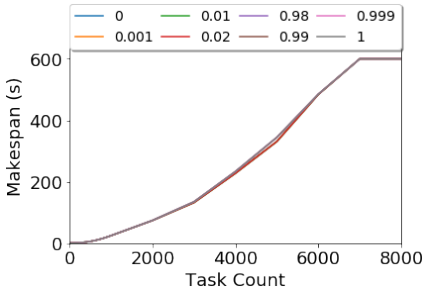
Figure 8.12: The impact of 100 iterations of a varying serial workflow. For the distribution profiles, curves depict the following percentiles (`0`) the zeroth percentile, (`0.01`) the 1st percentile, (`0.1`) the 10th percentile, etc.



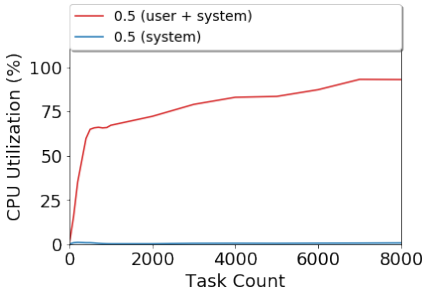
(a) Trend of makespan.



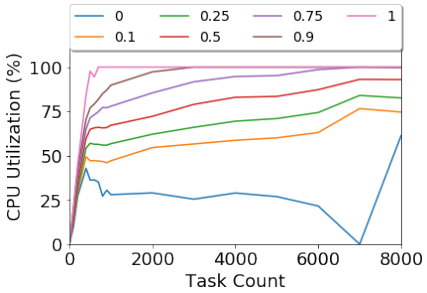
(b) Distribution of makespan.



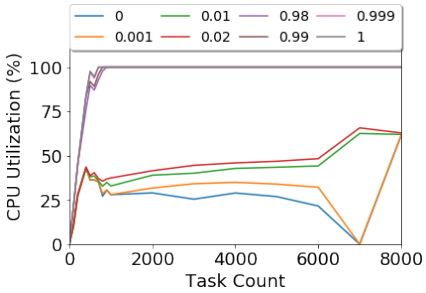
(c) Extremes of makespan.



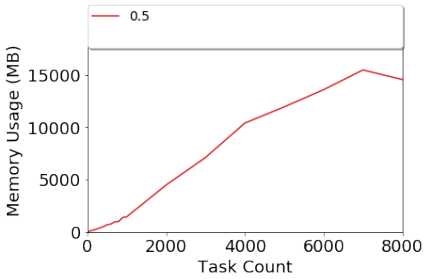
(d) Trend of CPU utilization.



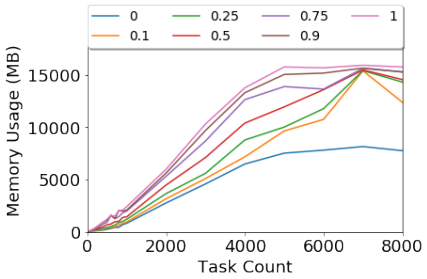
(e) Distribution of CPU utilization.



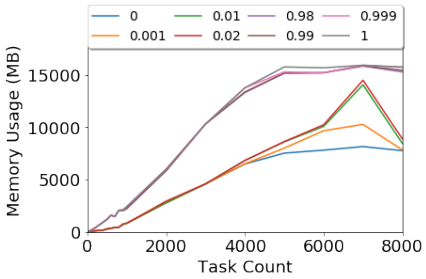
(f) Extremes of CPU utilization.



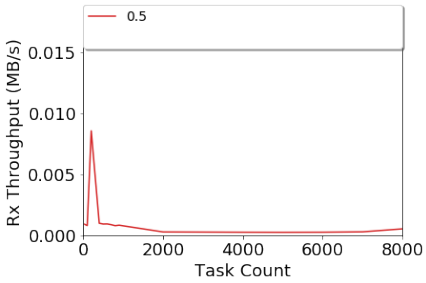
(g) Trend of memory usage.



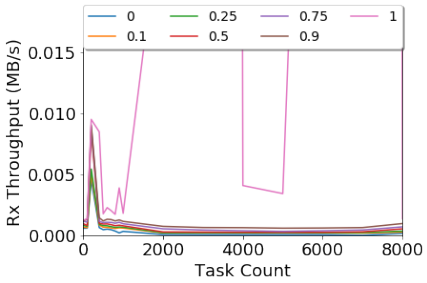
(h) Distribution of memory usage.



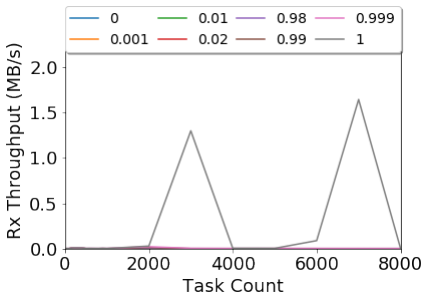
(i) Extremes of memory usage.



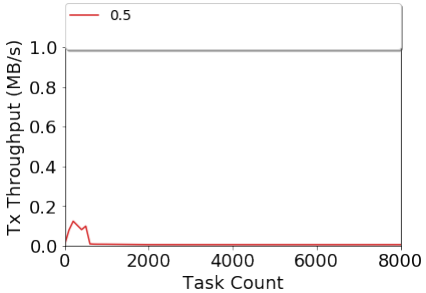
(j) Trend of incoming network throughput.



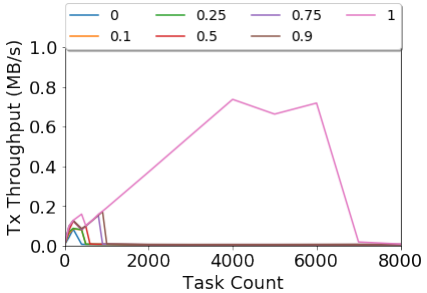
(k) Distribution of incoming network throughput.



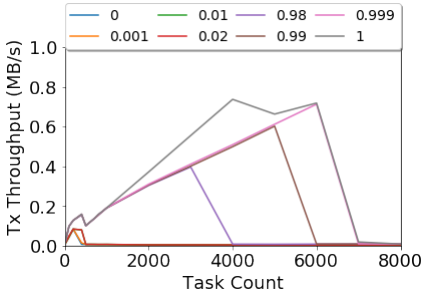
(l) Extremes of incoming network throughput.



(m) Trend of outgoing network throughput.



(n) Distribution of outgoing network throughput.



(o) Extremes of outgoing network throughput.

Figure 8.13: The impact of 100 iterations of a varying parallel workflow. For the distribution profiles, curves depict the following percentiles (`0`) the zeroth percentile, (`0.01`) the 1st percentile, (`0.1`) the 10th percentile, etc.

events that have to be linearized. Furthermore, from Figure 8.13 it is clear that, regardless of the cause, the key culprit of the performance degradation is the CPU utilization, which already spikes at 200 QPS. The reason for this is likely that the controller in Fission Workflows becomes overwhelmed with tasks. Although this is in part something that needs to be addressed in the software, in further experimentation one could try changing some of the configurations related to the controller (see Appendix C).

A final observation from Figure 8.13 is that it uses much memory for the workflows it executes. This also been observed in other experiments, and falls in line with the general observation that the prototype is too memory inefficient. This is in part due to aggressive caching, but also due to the use of event sourcing. Each task generates two events, and each of these events is stored in the system. So at workflows with a large number of tasks, this quickly turns into thousands of tasks that both need to be stored (increasing the memory in Figure 8.13) and processed (increasing the CPU contention).

8.2.4. Scheduling Experiments

Although evaluating schedulers and scheduling policies constitutes a thesis on its own, the goal of these scheduling experiments is to reveal initial insight into the implications of the available scheduling policies. Based on these experiments we arrived at the following findings:

1. As long as cold starts are non-zero, prewarming policies outperform the default policy on workflow invocation runtime, with the `prewarm-all` policy outperforming the other policies by an increasingly wider margin, as the cold start duration increases.
2. Prewarming policies trade-off the improved runtime by expending more resources, until the cold start duration causes (non-prewarming) horizon policy to perform worse simply due to the extended duration.

8.2.4.1. Impact on Workflow Runtime

The intent of this experiment is to evaluate the performance of the scheduling policies. The main motivation behind this is that the scheduling policies focusing on prewarming should outperform non-prewarming policies. Overall, our key findings:

- Both the `prewarm-all` and `prewarm-horizon` policies outperform the non-prewarming `horizon` policy as long as cold starts are non-zero.
- Despite prewarming all tasks, `prewarm-all` scheduled workflows still increase in runtime because the first task of a workflow cannot be prewarmed.

As can be observed in Figure 8.14a, The default `horizon` policy only performs similar the others when cold starts are (near-)zero. Beyond the initial zero cold start, we observe that this policy grows fastest in runtime as cold starts increase. Given the pipeline structure of the workflow, the runtime follows an exponential curve, c^n, where c is the cold start and n the number of tasks in the workflow. Only because of the low task count in the Chronos trace (3 tasks per workflow), this exponential growth is difficult to observe.

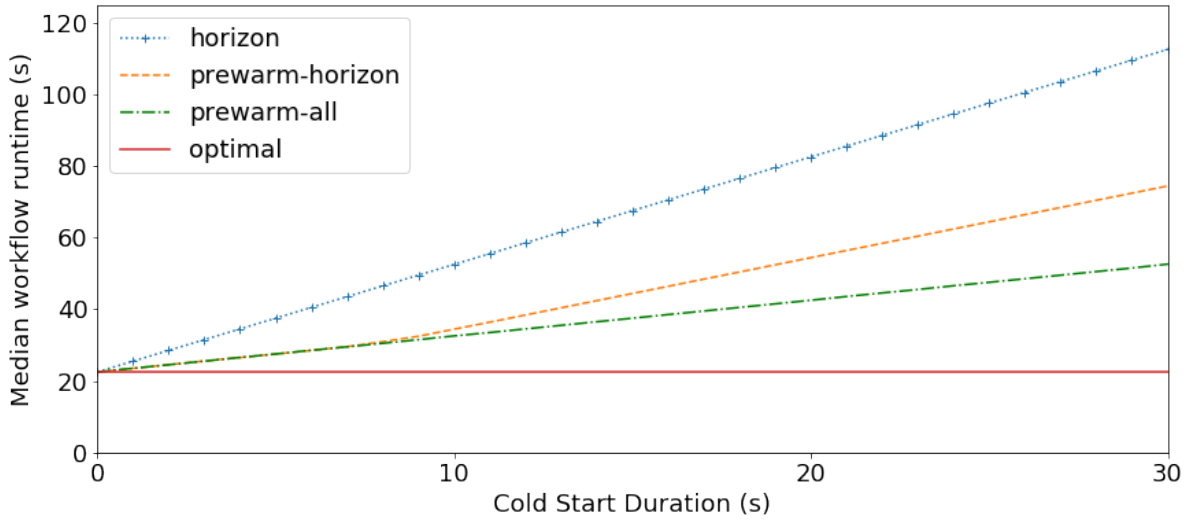
The `prewarm-all` task is continuously closest to the optimal runtime (Figure 8.14a), outperforming both the `horizon` and `prewarm-horizon` policies. In fact, the runtime overhead added to it is solely the cold start of the first task, because the policy does not attempt to predict, let alone prewarm, this first task.

The final policy in Figure 8.14a, `prewarm-horizon`, performs comparable to the `prewarm-all` policy. It starts to perform worse at high cold starts, where the cold start starts to exceed the runtime of tasks—this can be clearly seen at around the 8 seconds of cold start, which is near the median task length as shown in Figure 8.3b. Beyond this point the policy cannot finish prewarming the next task before the current task completes; the cold starts are too long.

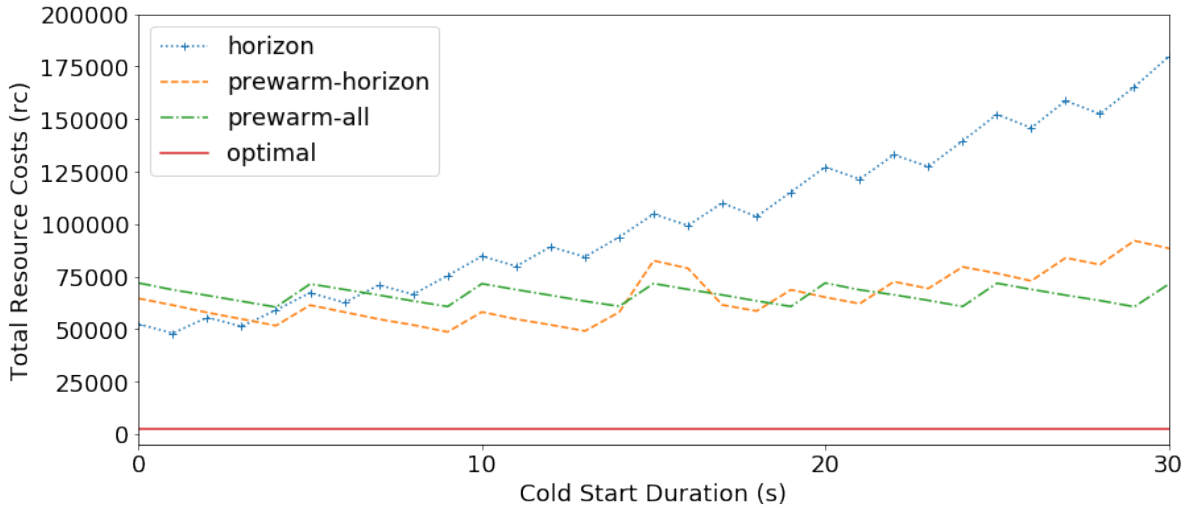
8.2.4.2. Impact on Resource Cost

The goal of this experiment is to analyse the impact of the scheduling policies on the resource consumption. For this we use the resource usage metric exposed by SimFaaS, which is defined as the cost of keeping a function instance alive for 1 second. Based on this evaluation, our main findings are:

- The `horizon` policy is cheaper for small cold starts, but becomes worse as the cold starts increase.



(a)



(b)

Figure 8.14: The impact of varying levels of cold start on the runtime (Figure 8.14a) and the resource consumption (Figure 8.14b) the Chronos workload using the three scheduling policies.

- The `prewarm-all` is wasteful for small cold starts, but starts to be comparatively more efficient with its resources as the cold start duration increases.
- The `prewarm-horizon` is much cheaper in resources than `prewarm-all` while having comparable performance, although this advantages gets lost as cold starts increase.

For this metric, the `horizon` policy performs best initially. However, as shown in Figure 8.14b, at large cold starts (> 5 seconds) the resources consumption of the `horizon` policy start to increase rapidly making it perform worse than the other policies. The reason for this is that we do not downscale the function instances during the experiment. Because the `horizon` policy at extreme cold starts has a long runtime, this also impacts the resources consumed during this (longer) experiment run.

In contrast, the `prewarm-all` policy is wasteful in resource consumption. It prewarms all tasks regardless of how long it will take before they are actually executed. However, as can be observed in Figure 8.14b, these costs stay largely stable, compared to the resource consumption changes in the other policies.

In between these two extremes we find in Figure 8.14b the prewarm-horizon policy. Compared to

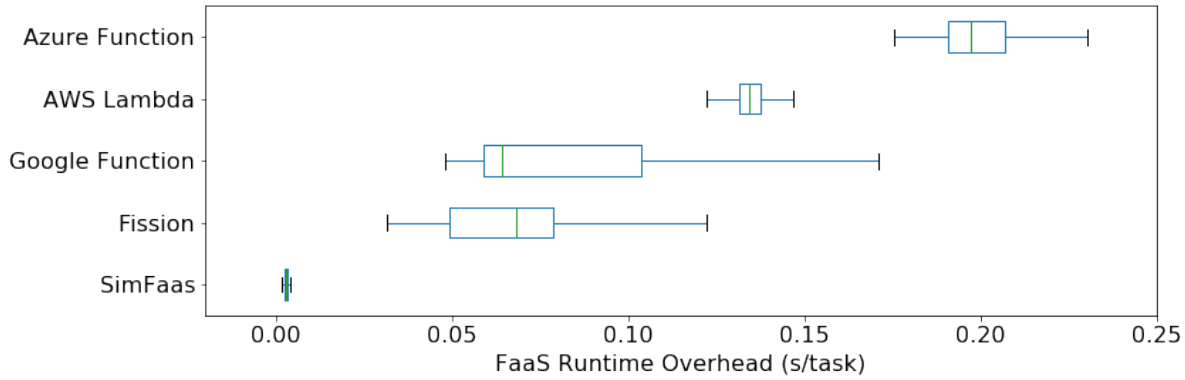


Figure 8.15: The performance overhead of the FaaS platforms with the Chronos workload.

`prewarm-all`, it saves 10% of resources by prewarming only the subsequent functions. Compared to the `horizon` policy, it is still wasteful in resources while attempting to prewarm functions. Similar to the `horizon` policy, due to the cold starts overtaking the task runtimes, the runtime duration increases; leading to increases in resource consumption.

8.2.5. Real-World Experiments

So far we have focused on the impact on performance of specific components, specific features, or specific workload characteristics. The goal of the real-world experiments is to evaluate the entire system as users would use it in production-level environments, and how it compares to the state-of-the-art in the domain of workflow orchestration. These experiments have led us to the following findings:

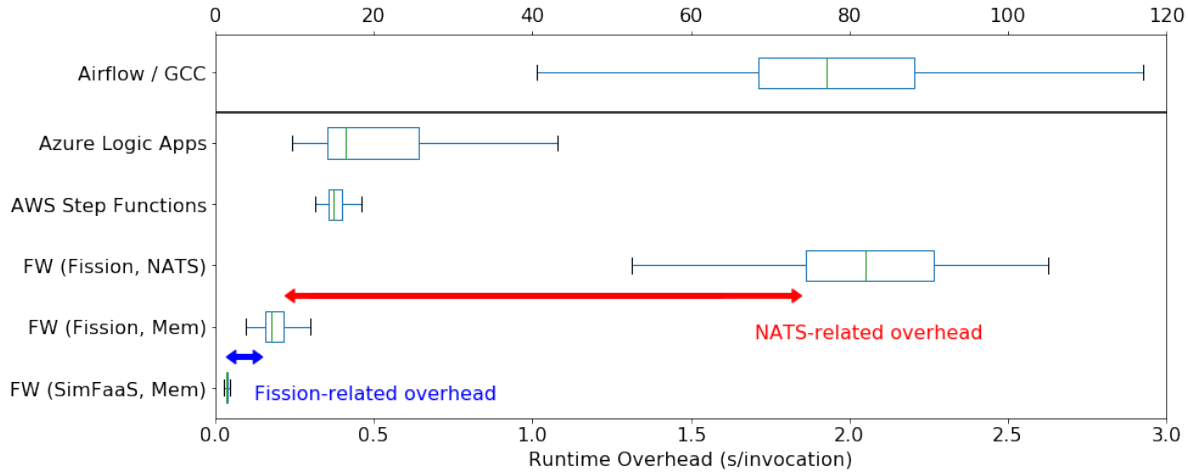
1. The FaaS platforms used in the realistic experiments have performance differences, with Azure Functions being slowest with a median runtime overhead of 180ms, and Google being fastest with a median runtime overhead of about 60ms.
2. Fission Workflows (with Fission and an in-memory event store) is faster than state-of-the-art (serverless) workflow engines when executing the Chronos workload; being about 0.7% faster than Azure Logic Apps, 1.1% faster than AWS Step functions, and 78.2% faster than Apache Airflow.
3. Despite the inefficient implementation of the NATS event store, Fission Workflows with NATS, though slower than Azure Logic Apps and AWS Step Functions, is still 76.7% faster than the open-source alternative, Airflow.
4. Fission Workflows is (amortized) the cheapest workflow engine; being overall about 68.2% cheaper than the next cheapest option Google Cloud Composer, about 89.9% cheaper than AWS, and 97.6% cheaper than Azure Logic Apps.

8.2.5.1. Comparison of FaaS Platforms

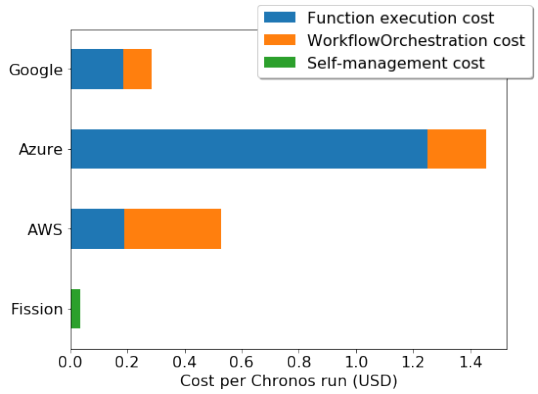
Before we compare the workflow orchestration systems, in this section we first evaluate the performance of the underlying FaaS platforms. Key findings are:

- The FaaS platform have between 50ms and 200ms of runtime overhead per task.
- Fission and Google Functions are the fastest in this evaluation, both with a median of about 50-60ms of overhead.
- Azure Function is comparatively the slowest in this evaluation with a median 180ms of overhead.
- AWS Lambda has an overhead of about 125ms, but has the least variance.

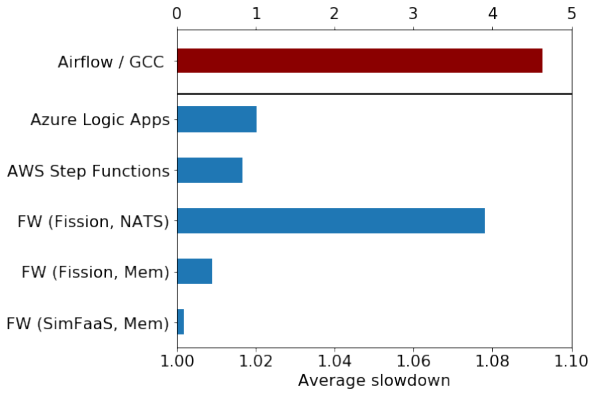
Figure 8.15 shows the performance overhead of the different FaaS platforms used in the realistic experiments. Included as a reference, SimFaaS has the least performance overhead, a median of 4ms. The actual full-sized FaaS platforms instead have an overhead between the 50ms and 200ms.



(a) The performance overhead introduced by the systems per workflow invocation in the Chronos workload.



(b) The cost of running the Chronos workload.



(c) The slowdown of the serverless platforms.

Figure 8.16: Comparison of runtime overhead of WMSs to execute the Chronos trace.

Although these benchmarks have a lot of parameters and factors that can interfere with the evaluation, the numbers we report here are in line with those of other FaaS benchmark papers [86, 137].

Fission and Google Functions have the lowest runtime overhead of about 50-60 ms, as observed from Figure 8.15. However, both have a wide variance, which could indicate that these platforms are trading in performance variability guarantees for better performance. In contrast, even though AWS Lambda performs a bit worse than Google Functions, it has a far smaller variability. Azure Functions performs comparatively worst with a runtime overhead of roughly 180ms. Surprisingly, this is after the additional work we have had to put into Azure to improve the performance (see Appendix C).

8.2.5.2. Comparison of Performance Overhead

The first goal of the real-world experiments is to compare the performance overhead of a realistic deployment of Fission Workflows to that of the state of the art. Our key findings:

- Fission Workflows with NATS has a large performance overhead due to workarounds in the NATS integration.
- Apache Airflow / Google Cloud Composer (GCC) cannot run the Chronos workload without simplifying it.
- Azure Logic Apps and AWS Step Functions have similar performance, but with AWS having a much smaller performance variability.

Cost item	Fission Workflows	AWS	Azure	Google Cloud
Compute nodes	4	-	-	4
Node cost per hour	0.0475	-	-	0.0475
Other hourly costs	-	-	-	0.4092
Function cost per second	-	0.00000208	0.000014	0.00000203
Function cost per execution	-	0.0000002	0.000000169	0.0000004
Orchestration cost per workflow	-	0.0001	0.00006	-
Experiment duration (seconds)	651.0	653.2	650.5	650
Total Costs	0.0343	0.5278	1.455	0.293

Table 8.5: The cost calculations for each of the evaluated platforms. All prices are quoted in euros and were referenced in February 2019. For Google Cloud, we assume the experiment duration to be 650 seconds, since the Chronos trace could not be run.

Figure 8.16a shows the runtime of three variations of the Fission Workflows deployment. First, Fission Workflows using SimFaaS, which serves as a reference point for all the compared systems. Second, Fission Workflows with Fission and the in-memory event store. The difference between the Fission Workflows with SimFaaS and Fission Workflows with Fission is in line with the overhead we found for Fission in Figure 8.15: with three function executions (one for each task), the about 200-250ms of runtime overhead should be added to the workflow. Fission Workflows with NATS is far slower than the other two deployments. The reason for this is the way that we implemented the integration with NATS. As we have addressed in the implementation (see Chapter 7), there were several features missing in NATS. Although we worked around these missing features, these workarounds have had to sacrifice performance. This is likely biggest contributor of the added overhead.

Google Cloud Composer (GCC), or Apache Airflow, is significantly slower than the other options. These results (as observed in Figure 8.16a and Figure 8.16c) indicate that Airflow (despite being the only system that required considerable re-configuring to handle even the modest Chronos workload) is not built for serverless workflows nor built for high throughput. Not only is it slower, as described in Appendix C, after several attempts, we were not able to run the Chronos workload. So the results of Airflow in Figure 8.16a and Figure 8.16c are run on a simplified Chronos workload, where the max concurrency of the workflow invocations is 1.

8.2.5.3. Comparison of Execution Costs

Increasing performance of a system is easy when you do not consider cost; simply add more resources. Despite the huge difference in cost (models), it is therefore key in this experiment to also consider the costs—to evaluate if a system is not simply more performant because it employs more (expensive) resources. Our key findings:

- Fission Workflows is (amortized) the cheapest workflow engine; being at least 68.2% cheaper than the next cheapest option Google Cloud Composer.
- The cost models of the managed WMS systems is opaque.

Figure 8.16b highlights the difference in cost: when executing the Chronos workload Fission Workflows is 89.9% cheaper compared to AWS Step Functions, 97.6% cheaper compared to Azure Logic Apps, and 68.2% cheaper compared to Google Cloud Composer. The main reason for this likely that Fission Workflows is the only self-managed WMS deployment. All other systems either are fully-managed, or, in the case of Google Cloud Composer, in partly-managed by the cloud provider, which adds to the overall cost of the platform.

Although we initially expected at least the cloud provider options to be reasonably comparable in pricing, even between those the costs of executing the Chronos workload vary: from 0.29 EUR (Google), to 0.53 EUR (AWS), to 1.45 EUR (Azure). The cost breakdown in Table 8.5 highlights a current issue in serverless computing in general: pricing is complex and opaque. There seems to be a clear pricing model for FaaS executions, where you pay for starting the execution and for the subsequent CPU time and memory used per 100 ms. For orchestration the pricing model is different

for the compared systems. Where AWS charges for *transitions* (edges in the workflow), Azure charges for *actions* (tasks in the workflow). Another extreme is Google Cloud Composer in which you get a (managed) instance of Airflow in a Kubernetes cluster, for which you pay hourly the costs of both the Kubernetes nodes and the other components of Airflow (e.g., database, web server).

8.3. Discussion

The prototype outperforms state-of-the-art (serverless) workflow management systems in both performance and cost. We found that for a real-world workload Fission Workflows has 76% less overhead, while costing 68% less, compared to the open-source industry alternative, Airflow.

These results can largely be attributed to the system being purpose-built to serve serverless workloads. Compared to other managed workflow engines (e.g., AWS Step Functions), the prototype offers scheduling policies to boost performance by specifically optimizing the scheduling of FaaS functions, and it allows users to increase performance by sacrificing degrees of persistence. Compared to traditional workflow engines (e.g., Apache Airflow), the prototype not only performs better, but also does allow for typical serverless use cases, such as synchronous workflow invocations, mapping workflow invocations from/to web requests/responses.

There are strengths to these experiments: (1) the experiments are diverse in nature to verify that the prototype is not improving only one metric at severe expense of another (which is one of the reasons why we included experiments on fault-tolerance and measured the cost in others); (2) careful effort was spent to ensure that the tools used in the experimentation are not impacting the experiments themselves (such as the experiments on the SimFaaS tool); and (3) the experiments themselves have been performed with high granularity (e.g., the small intervals used in the scalability experiments), automated, and repeated several times.

However, before and during experimentation we also identified several threats to validity:

1. **Lack of diverse infrastructure**: Due to the limited time that could be spent on the evaluation, we limited the experimentation to two infrastructure setups to deploy the systems on: (1) two commodity nodes for the synthetic experiments, and (2) a 5-node Kubernetes cluster hosted on Google Cloud. To mitigate this we did not rely on specific features or characteristics of the infrastructure used. Furthermore, we believe that our experiments already reveal, through the analysis, the general behavior of and relations between components.

2. **Limited control over and insight into comparison systems**: As already noted in our surveys (Chapter 4), the hosted options used in our experiments are closed-source, proprietary platforms. This limits our, and the community's, ability to understand the choices made within these systems. Furthermore, it impedes our ability to create a controlled experiment environment. We tried to mitigate this by not only focusing on performance, but also on the cost of executing the workloads. Moreover, we also were conservative in how we compared Fission Workflows to the other systems: we relied on commodity (cloud-based) hardware; and we held the compared systems to weaker standards than our prototype.

3. **Limited workload variety**: At the moment of writing there are no publicly available workload traces for either FaaS or serverless workflows. Nor are there studies available describing the (performance) characteristics of these workloads. This lack of data makes it challenging to evaluate the workflow engine realistically. For this reason we reverted to using synthetic workloads in most cases, and the *Chronos* workload trace [92] as a proxy for a realistic serverless workflow workload trace.

4. **Uncovered functionality and configurations**: With the experiments in this thesis we aimed to cover a broad range of aspects, rather than going into depth for any given component or functionality. Yet, the system itself is vast in terms of functionality and the configurations of this functionality. Experimentation of advanced functionality was deferred to future work, which includes data transfer and management, expressions, dynamic workflows, internal functions. Furthermore, of the topics addressed in the experimentation typically only part of the configuration space was covered. For example, we did not evaluate the impact different event store implementations in the scalability and scheduling experiments. Instead, for the sake of keeping this thesis within reason-

able bounds—time and page-wise—we focused the experimentation on the configurations that we believed were most relevant and realistic.

5. **Performance issues and bugs**: A workflow management system is a complex piece of software. Despite the (more than usual) effort put into Fission workflows, when pushed to its limits in the experiments, we uncovered and resolved multiple performance issues and bugs. Other issues were identified, classified as non-critical and were deferred to be resolved in the future (outside of the time dedicated to this thesis). For example, the system was at the time of the experimentation quite wasteful in its memory consumption when running at large scale. Though this can be observed in the experimental results, as long as enough memory is allocated it was not critical to resolve for the experiments.

As mentioned throughout this thesis, there is much experimentation left to be done. Part of our future work is to go into far more detail with regards to performance evaluation of specific functionality. This is especially true for the scheduler and scheduling policies, which provide ample opportunities to further investigate.

The results of the evaluation of our prototype support the initial motivation for this research: the state-of-the-art in function composition and workflow orchestration do not yet fit the emerging serverless paradigm. The prototype shows that a basic workflow engine focusing specifically on serverless workflows improves the state-of-the-art, with further opportunity left to improve it further. By leveraging both the capabilities and constraints of the serverless and FaaS model, serverless workflow orchestration has the opportunity to become part of a long-awaited notion: a universal cloud programming and execution model.

9

Conclusion and Future Work

Serverless computing and FaaS have seen, and continue to see, rapidly increasing adoption within the field of cloud computing. The prospect for users of pay-per-use, fully-managed functions has caused the cloud community to propel the serverless paradigm into a wide variety of use cases. As users move more and more complex workloads to serverless models, there is a need for better tooling in the serverless ecosystems. Serverless function composition—composing existing FaaS functions into more complex FaaS functions—is an effective way to manage the complexity of these workloads. To provide users with an intuitive, low-overhead, fully-managed approach to function composition, in this thesis we propose a serverless workflow management system, which provides the user with a minimal set of primitives to compose FaaS functions together into ever-more complex functions, and execute these fast, reliable, and following the characteristics of FaaS. To validate the approach we designed, implemented, and evaluated Fission Workflows.

In the remainder of this chapter, in Section 9.1 we summarize how the approach in this work has answered the research question posed in the introduction (Chapter 1). This is followed by an overview of the future work: from the conceptual notion of function compositions, to the Fission Workflows system, and to serverless workflow schedulers.

9.1. Conclusion

In this work we relied on a broad approach, consisting of conceptual, software, and experimental contributions to find an answer to the question: *How to support serverless computing based on functional composition through a distributed systems approach?* We found that a workflow-based approach is likely an appropriate candidate for serverless function composition. A conclusion which we strengthened by designing and evaluating a workflow-based function composition prototype (Fission Workflows). Specifically, using this approach, we can answer the four research questions posed in the introduction (Chapter 1):

RQ1: What are the requirements to cloud function compositions? Given the rapidly and uncontrolled evolving of the field of serverless computing, there is no universally agreed upon set of use cases, workloads, characteristics for FaaS functions—let alone for function compositions. Even the definitions *serverless computing* and *FaaS* vary from publication to publication, and from organization to organization.

To find out what the requirements, use cases, workloads and characteristics of cloud function compositions are, we approached this using a top-down approach. Partly in our work with the SPEC RG CLOUD, we surveyed the field on a high level, proposing definitions and characteristics for the key notions of *serverless*, *FaaS*, and *cloud functions* [132, 134].

In this thesis, we moved further into detail regarding the requirements and characteristics specific to *function compositions*. By surveying and analyzing industry and research sources, we found that function compositions have unique requirements, workload characteristics, and constraints, and enable serverless computing for additional use cases. This resulted in both a list of user-level requirements and system-level constraints relevant to FaaS function compositions.

RQ2: How to compare approaches to the composition of cloud functions? Using the requirements analysis (Chapter 3), we have a clear picture of what characteristics the ideal function composition approach should have. However, as with most problems in computer science, there are many possible solutions, each with their own strengths, weaknesses, and trade-offs.

To determine which approach to function composition satisfies the requirements and constraints, we qualitatively compared and analyzed the following aspects:

1. **FaaS platforms**: we found that the *Fission FaaS platform* satisfies to the satisfies the serverless model, and has the architecture that satisfies the requirements to serve as the FaaS platform underlying the function composition prototype.
2. **Function composition approaches**: we found that a *workflow-based approach*, with its decoupled, centralized architecture, and extensive body of previous work is a good candidate for function composition.
3. **Workflow management systems**: we found that no existing open-source WMS fulfills the user-level requirements and system-level constraints of serverless workflows. However, for the various components of the serverless WMS we can take inspiration from or entirely re-purpose specific components from existing workflow engines.
4. **Workflow languages**: similar to the WMS, we found that no workflow language accurately matches the data and execution model of serverless workflows. To prevent becoming locked into an ill-fitted workflow language, we decided to design a new serverless workflow language alongside the prototype.

Based on the surveys and the requirements analysis, we posed the serverless function composition workloads as a novel type of workflow: *serverless workflows*. This type of workflow contains unique dimensions and characteristics that make it a distinct from other, existing workflow domains, such as business processes, scientific workflows, or business-critical workflows.

RQ3: How to design a system for composing cloud functions? Based on the requirements analysis and surveys, we proposed Fission Workflows: a workflow management system that specifically targets serverless workflows. It focuses on reliable, low-overhead workflow executions. Moreover, it has features specific for the serverless model, such as the design principle that workflows are exposed as FaaS functions. It further contains an extensible scheduling model to allow users and researchers to evaluate custom scheduling policies. We created three scheduling policies: the *horizon* policy, *prewarm-horizon*, and a *prewarm-all* policies which aim to deploy function instances ahead of time.

Alongside this workflow system, we proposed a new abstract workflow language: *Serverless Workflow Language (SWL)*. SWL satisfies the requirements of serverless workflows, using a minimal, DAG-based language.

RQ4: How to evaluate cloud function composition systems experimentally? To experimentally evaluate the workflow system prototype, following the system-level constraints of FaaS (Chapter 3), we proposed a serverless-oriented testing framework. As a part of this testing framework we created SimFaaS, a FaaS emulator, to more reliably isolate the experiments that require a FaaS platform.

Using this framework we evaluated whether the workflow engine satisfies the user-level requirements in terms of scalability, overhead, and resiliency. Within these experiments we also evaluated the impact of the scheduling policies on the relevant metrics. Finally, we compared the workflow engine to state-of-the-art serverless workflow engines using realistic scenarios.

9.2. Future Work

In our prior work [132–134], we identified the current perspectives and challenges in the emerging field of serverless computing, many of which also apply to this thesis. For this reason, we focus solely on the future work directly related to the contributions of this work. We divide these into three categories: future work related to concept of serverless function composition; possible extensions and improvements that can be made to Fission Workflows; and, future work related to scheduling of serverless workflows.

9.2.1. Serverless Function Composition

The field of serverless workflows, which we introduced in this work, contains the following promising research direction:

1. **Evaluate other approaches to function composition quantitatively.**

   In this time- and resource-constrained study we focused solely on workflows as the function composition approach, after an initial qualitative comparison (Section 4.2). Yet, this by no means should imply that the field of function composition has been explored exhaustively. Especially, hybrid solutions seem interesting directions to investigate further. For instance, one could use the expressive coordinator approach, but add a pause and resume functionality—which can be fast enough for web applications use cases as Oh et al. [102] demonstrated—using the event-sourced workflow approach in the execution model.

2. **Research higher-level approaches to express function composition.**

   Although we proposed an architecture in which the workflow language takes the role of an Intermediate Representation (IR) for serverless function compositions, ways for high-level expression this direction is left mostly unexplored in this work. The emergence of serverless computing as a conceptual model and technology provide a well-defined basis for further research into new ways of expressing distributed applications.

3. **Find the optimal granularity of serverless function composition.**

   A trade-off that is left unexplored in this thesis is regarding the optimal granularity of the serverless function compositions. On one extreme, we can go as granular as possible: define a small minimal set of functions needed to express computation, such as the set of μ-recursive functions [95], and use multiple layers of function compositions to create higher-level functions. On the other extreme we can choose to only create workflows for a few high-level functions. Workflows with many granular functions increase the options schedulers have to optimize the execution, while small workflows limit the potential overhead introduced by the function composition. Finding the optimal balance between these two extremes would help establish which granularity serverless computing should aim for.

4. **Determine when to use which approach to function composition.**

   From our experiences in research and in industry, our intuition is that there does not seem to be a 'one-size-fits-all' approach to function composition. Especially, the difference between the centralized nature of workflows and the decoupled nature of event-based composition seems to indicate that they both have distinct use cases in which one or the other is better to use. A thorough survey and mapping of approaches to use cases is needed to improve community understanding of the advantages and disadvantages of the approaches.

9.2.2. Fission Workflows

Fission Workflows still has many improvements and extensions left as future work. These include:

1. **Improve data transfer between tasks.**

   Similar to most state-of-the-art workflow management systems, Fission Workflows has so-far left data transfer as an exercise to the user. It naively passes input from the engine to the function, and output back to engine—persisting the data. Although this is useful for debugging and visualizing the workflow execution, this slows down functions attempting to pass large amounts of data to each other. Various improvements are possible here to solve this issue, such as our proposal using *distributed promises* [131] or a solution based on a feature analogous to Direct Memory Access (DMA) in operating systems [126].

2. **Provide support for multi-tenancy.**

   In the current state, Fission Workflows assumes a single stakeholder to be using it; it lacks accountability and fairness in its scheduling of workflows. Extending the serverless workflow engine with multi-tenancy support should help us understand the impact of multi-tenancy in serverless workflows.

3. **Explore further versioning support and strategies.**

   Fission Workflows contains basic versioning support; users are able to pin functions to specific versions or FaaS providers. However, future work could expand further on this notion. How can we express ranges of versions? What kind of version upgrading strategies should be supported? Given the granularity of functions, at a large scale extensive support for these concepts is needed.

4. **Improve the *serverless developer* experience.**

   As one the first prototypes in this area it was challenging to predict how users will use it, expect from it, and for what purpose do they use it. Like all other projects in this space, we have yet to determine the optimal workflow of a developer of serverless applications. What development tools are needed? What kind of debugging and testing environments are needed? What metrics to monitor?

5. **Investigate the options for legacy migration.**

   A potential use case of serverless has been touted to be the ease of migrating legacy codebases to a FaaS platform (e.g., serverless COBOL[1]). Although this so-far has yet to take off, workflows add options for further migration of legacy software. To what extend can we migrate legacy software automatically to serverless workflows?

6. **Research security and privacy implications.**

   The orchestration of business processes and web applications will frequently involve confidential data. Currently, the workflow engine logs and stores all inputs and outputs, which might cause it to unintentionally expose sensitive information in logs and data storage. With few to none workflow engines addressing issues like this, Fission Workflows could form the basis of security and privacy research for both workflow and serverless ecosystems.

7. **Explore the workload trace extrapolation trade-off.**

   In the evaluation of the prototype we only used a single realistic workload, Chronos (see Section 8.1.2.2). Moreover, this workload was based on a short—10 minutes—time-frame. We extrapolated this workload to enable longer running experiments. This presents a trade-off in our experiments: the extrapolation limits the realism of the experiments, but it does enable us to conduct longer-running measurements that limit the effects of spurious system instability. Further investigation of this trade-off could lead to new methodological guidelines for analyzing the performance of serverless platforms.

9.2.3. Serverless Workflow Scheduling

The upside of starting a thesis in a relatively unexplored field of research is that you can collaborate on the fundamentals, such as the case with the workflow engine. The downside is that there are too many fundamental issues to resolve before you can start working on the most interesting parts—as is the case with serverless workflow scheduling. Although in this work we explored this field in Chapter 6, there is much more to explore. This section contains the future work in serverless workflow scheduling:

1. **Explore options to schedule for data and software locality.**

   Similar to the future work related to data transfer, research is needed for scheduling policies that take data locality into account. Although there is much existing research on data-locality-aware workflow scheduling, the dynamic nature of FaaS makes it challenging for schedulers to generate comprehensive executions plans that optimize for data locality, such as those described by Szabo et al. [125] In prior work [20] we made initial steps towards software locality, which could be improved further with the help of the knowledge of the workflow scheduler.

2. **Generate models of functions based on historical data.**

   The advantage of the high-frequency executions of serverless workflows is that we can collect more data points to model the performance characteristics of the functions and workflows. This knowledge allows for predicting more accurately when certain functions need to be deployed.

[1] `https://github.com/morecobol/cobol.run`

This can reduce the idle time of functions being deployed too soon, or the slowdown related to functions being deployed to late.

3. **Explore predictive scheduling possibilities to speed up dynamic workflows.**

   Building upon the generations of models, we can use these models to explore policies to improve the predictive scheduling of dynamic workflows. Currently, with dynamic workflows (see Section 5.2.3) we loose the benefits of statistical analysis of the scheduler. Following lessons learned in programming languages and CPU research, future work can overcome this loss of performance by emphasizing runtime analysis of dynamic workflows.

4. **Investigate scheduling policies in a cluster with heterogeneous resources.**

   Fission Workflows, and the Fission FaaS underlying it, currently largely consider resources to be homogeneous. However, with multi-cloud situations this assumption could no longer be valid. Future research could focus on exploiting the heterogeneous resources to optimize the execution of serverless workflows.

5. **Consider non-functional requirements (NFRs) during scheduling.**

   With the serverless computing field being yet another step up in the abstraction layer, it becomes increasingly important to enable users to set more human-level, non-functional, requirements for their serverless applications. In future work, the support of NFRs, such as priority, security, and cost, could be explored further.

6. **Explore Scheduling policies in dynamic pricing and cloud contexts.**

   A specific NFR that should be considered future work is that of cost. In the increasingly dynamic cloud computing market, with multiple cloud vendors and resource marketplaces (such as AWS Spot Instances[2]), a sophisticated workflow scheduler could optimize the execution of workflows across different cloud vendors. The cheap-to-deploy, and with a largely consistent cross-cloud API, FaaS model makes this domain attractive to investigate.

9.3. Disclaimer

During the design, implementation and evaluation stages of this thesis we collaborated intensively with Platform9 Systems in Sunnyvale, CA: the company leading the development of the Fission FaaS platform.

[2] `https://aws.amazon.com/ec2/spot/`

Bibliography

[1] Aws serverless application repository. https://aws.amazon.com/serverless/serverlessrepo/, . Accessed 2019-03-04.

[2] Amazon states language. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html, . Accessed 2019-02-15.

[3] Schema reference for workflow definition language in azure logic apps. https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-workflow-definition-language. Accessed 2019-02-15.

[4] Tools for monitoring resources - kubernetes. https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/. Accessed 2019-05-31.

[5] Openstack mistral: Welcome to the mistral documentation! https://docs.openstack.org/mistral/latest/. Accessed 2019-02-15.

[6] Fission - serverless functions and workflows with kubernetes and nats. https://nats.io/blog/serverless-functions-and-workflows-with-kubernetes-and-nats/. Accessed 2019-03-04.

[7] Aws re:invent 2014 | announcing aws lambda. https://www.youtube.com/watch?v=9eHoyUVo-yg, 2014. Accessed: 2018-7-28.

[8] Autodesk goes serverless in the aws cloud, reduces account-creation time by 99 https://aws.amazon.com/solutions/case-studies/autodesk-serverless/, 2018. Accessed 2018-07-29.

[9] Aws lambda: Customer case studies. https://aws.amazon.com/lambda/resources/#Customer_Case_Studies, 2018. Accessed 2018-07-29.

[10] Azkaban workflow manager. https://github.com/azkaban/azkaban, 2018.

[11] Azure: Serverless computing. https://azure.microsoft.com/en-gb/overview/serverless-computing/, 2018. Accessed 2018-07-29.

[12] Case study: Financial engines cuts costs 90computing. https://aws.amazon.com/solutions/case-studies/financial-engines/, 2018. Accessed 2018-07-29.

[13] Case study: Finra adopts aws to perform 500 billion validation checks daily. https://aws.amazon.com/solutions/case-studies/finra-data-validation/, 2018. Accessed 2018-07-29.

[14] Google cloud functions. https://cloud.google.com/functions/, 2018. Accessed 2018-07-29.

[15] Cloud composer is now in beta: build and run practical workflows with minimal effort. https://cloud.google.com/blog/products/gcp/cloud-composer-is-now-in-beta-build-and-run-practical-workflows-with-minimal-effort, 2018. Accessed: 2018-01-30.

[16] Guardian news and media automates subscription fulfilllment using aws step functions. https://aws.amazon.com/solutions/case-studies/the-guardian/, 2018. Accessed 2018-07-29.

[17] Luigi: Documentation. https://luigi.readthedocs.io/en/stable/, 2018. Accessed: 2018-01-30.

[18] Netflix and aws lambda case study. https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/, 2018. Accessed 2018-07-29.

[19] Serverless use cases. https://serverless.com/learn/use-cases/, 2018. Accessed 2018-07-29.

[20] Cristina L. Abad, Edwin F. Boza, and Erwin van Eyk. Package-aware scheduling of faas functions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*, pages 101–106, 2018.

[21] Martin L Abbott and Michael T Fisher. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009.

[22] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick HJ Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Generation Computer Systems*, 29(1):158–169, 2013.

[23] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pages 923–935, 2018.

[24] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[25] Waleed Ali, Siti Mariyam Shamsuddin, Abdul Samad Ismail, et al. A survey of web caching and prefetching. *Int. J. Advance. Soft Comput. Appl*, 3(1):18–44, 2011.

[26] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.

[27] Peter Amstutz, Michael R Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, et al. Common workflow language, v1. 0. Technical report, CWL Initiative, 2016.

[28] Martin Arlitt and Tai Jin. *Workload characterization of the 1998 world cup web site*. Hewlett Packard Laboratories, 1999.

[29] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[30] Karl Johan Aström and Richard M Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.

[31] AWS. Case study: irobot ready to unlock the next generation of smart homes using the aws cloud. https://aws.amazon.com/solutions/case-studies/irobot/, 2017. Accessed 2018-07-29.

[32] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[33] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 89–103, 2017.

[34] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[35] G Bruce Berriman, Ewa Deelman, John C Good, Joseph C Jacob, Daniel S Katz, Carl Kesselman, Anastasia C Laity, Thomas A Prince, Gurmeet Singh, and Mei-Hu Su. Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand. In *SPIE Astronomical Telescopes+ Instrumentation*, pages 221–232. International Society for Optics and Photonics, 2004.

[36] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10. IEEE, 2008.

[37] Gerrit A Blaauw and Frederick P Brooks Jr. *Computer architecture: Concepts and evolution*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[38] Robert B. Bohn, John Messina, Fang Liu, Jin Tong, and Jian Mao. NIST cloud computing reference architecture. In *World Congress on Services, SERVICES 2011, Washington, DC, USA, July 4-9, 2011*, pages 594–596, 2011.

[39] Egon Börger. Approaches to modeling business processes: a critical analysis of bpmn, workflow patterns and yawl. *Software & Systems Modeling*, 11(3):305–318, 2012.

[40] Frederick P Brooks Jr. *The design of design: Essays from a computer scientist*. Pearson Education, 2010.

[41] Robert Grover Brown, Patrick YC Hwang, et al. *Introduction to random signals and applied Kalman filtering*, volume 3. Wiley New York, 1992.

[42] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[43] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on software engineering*, 14(2):141–154, 1988.

[44] Tao Chen, Rami Bahsoon, and Xin Yao. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Computing Surveys (CSUR)*, 51(3):61, 2018.

[45] CNCF. Kubernetes components. `https://kubernetes.io/docs/concepts/overview/components/`, 2018. Accessed 2018-02-05.

[46] CNCF. Replicationcontroller. `https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/`, 2018. Accessed 2018-02-05.

[47] Sarah Cohen-Boulakia, Jiuqiang Chen, Paolo Missier, Carole Goble, Alan R Williams, and Christine Froidevaux. Distilling structure in taverna scientific workflows: a refactoring approach. *BMC bioinformatics*, 15(1):S12, 2014.

[48] Yan Cui. I'm afraid you're thinking about aws lambda cold starts all wrong. `https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f`, 2016.

[49] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future generation computer systems*, 25 (5):528–540, 2009.

[50] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[51] John Demian. Serverless case study - coca-cola. `https://dashbird.io/blog/serverless-case-study-coca-cola/`, 2018. Accessed 2018-07-29.

[52] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[53] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[54] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Making state explicit for imperative big data processing. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 49–60, 2014.

[55] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, et al. Software engineering meets control theory. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 71–82. IEEE Press, 2015.

[56] Sadjad Fouladi, Dan Iter, Shuvo Chatterjee, Christos Kozyrakis Matei Zaharia, and Keith Winstein. A thunk to remember: make-j1000 (and other jobs) on functions-as-a-service infrastructure. Technical report, 2017.

[57] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, pages 363–376, 2017.

[58] Martin Fowler. Event sourcing. `http://martinfowler.com/eaaDev/EventSourcing.html`, 2017. Accessed 2018-04-13.

[59] Bogdan Ghit, Mihai Capota, Tim Hegeman, Jan Hidders, Dick Epema, and Alexandru Iosup. V for vicissitude: The challenge of scaling complex big data workflows. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 927–932. IEEE, 2014.

[60] Susan Hall. Airflow, a workflow orchestrator for big data. `https://thenewstack.io/airflow-a-workflow-orchestrator-for-big-data/`, 2019. Accessed 2018-07-29.

[61] Gregory M Harry, LIGO Scientific Collaboration, et al. Advanced ligo: the next generation of gravitational wave detectors. *Classical and Quantum Gravity*, 27(8):084006, 2010.

[62] Mihály Héder. From nasa to eu: The evolution of the trl scale in public sector innovation. *The Innovation Journal*, 22(2):1–23, 2017.

[63] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016.*, 2016.

[64] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. *Elastic*, 60:80, 2016.

[65] Nikolas Herbst, Rouven Krebs, Giorgos Oikonomou, George Kousiouris, Athanasia Evangelinou, Alexandru Iosup, and Samuel Kounev. Ready for rain? a view from spec research on the future of cloud metrics. *arXiv preprint arXiv:1604.03470*, 2016.

[66] Nikolas Herbst, André Bauer, Samuel Kounev, Giorgos Oikonomou, Erwin Van Eyk, George Kousiouris, Athanasia Evangelinou, Rouven Krebs, Tim Brecht, Cristina L Abad, et al. Quantifying cloud performance and dependability: Taxonomy, metric design, and emerging challenges. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(4):19, 2018.

[67] Troy Hunt. Seamless a/b testing, deployment slots and dns rollover with azure functions and cloudflare workers. https://www.troyhunt.com/seamless-a-b-testing-deployment-slots-and-dns-rollover-with-azure-functions-and-cloudflare-workers/, 2018. Accessed 2018-07-29.

[68] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro Vittorio Papadopoulos, Bogdan Ghit, Dick H. J. Epema, and Alexandru Iosup. An experimental performance evaluation of autoscaling policies for complex workflows. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 75–86, 2017.

[69] FaaSlang Initiative. Faaslang: Function as a service language. https://github.com/faaslang/faaslang, feb 2018. Accessed 2018-07-29.

[70] Broad Institute. Workflow description language - user guide. https://software.broadinstitute.org/wdl/documentation/, 2018.

[71] Alex Iosup, Xiaoyun Zhu, Arif Merchant, Eva Kalyvianaki, Martina Maggio, Simon Spinner, Tarek Abdelzaher, Ole Mengshoel, and Sara Bouchenak. *Self-awareness of Cloud Applications*, pages 575–610. Springer International Publishing, Cham, 2017. ISBN 978-3-319-47474-8.

[72] Alexandru Iosup and Dick Epema. Grenchmark: A framework for analyzing, testing, and comparing grids. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, volume 1, pages 313–320. IEEE, 2006.

[73] Alexandru Iosup, Georgios Andreadis, Vincent Van Beek, Matthijs Bijman, Erwin Van Eyk, Mihai Neacsu, Leon Overweel, Sacheendra Talluri, Laurens Versluis, and Maaike Visser. The opendc vision: Towards collaborative datacenter simulation and exploration for everybody. In *Parallel and Distributed Computing (ISPDC), 2017 16th International Symposium on*, pages 85–94. IEEE, 2017.

[74] Alexandru Iosup, Alexandru Uta, Laurens Versluis, Georgios Andreadis, Erwin van Eyk, Tim Hegeman, Sacheendra Talluri, Vincent van Beek, and Lucian Toader. Massivizing computer systems: a vision to understand, design, and engineer computer ecosystems through and beyond modern distributed systems. *CoRR*, abs/1802.05465, 2018.

[75] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. The atlarge vision on the design of distributed systems and ecosystems. *CoRR*, abs/1902.05416, 2019.

[76] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, et al. Fireworks: A dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015.

[77] Raj Jain. The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation and modeling (book review). *SIGMETRICS Performance Evaluation Review*, 19(2):5–11, 1991.

[78] Qingye Jiang, Young Choon Lee, and Albert Y Zomaya. Serverless execution of scientific workflows. In *International Conference on Service-Oriented Computing*, pages 706–721. Springer, 2017.

[79] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[80] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th international conference on Autonomic computing*, pages 117–126. ACM, 2009.

[81] Evangelia Kalyvianaki, Wolfram Wiesemann, Quang Hieu Vu, Daniel Kuhn, and Peter Pietzuch. Sqpr: Stream query planning with reuse. In *2011 IEEE 27th International Conference on Data Engineering*, pages 840–851. IEEE, 2011.

[82] Evangelia Kalyvianaki, Themistoklis Charalambous, Marco Fiscato, and Peter Pietzuch. Overload management in data stream processing systems with latency guarantees. In *7th IEEE International Workshop on Feedback Computing (Feedback Computing'12)*, 2012.

[83] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 427–444, 2018.

[84] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 169–173. ACM, 2017.

[85] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. *Self-Aware Computing Systems*. Springer Publishing Company, Incorporated, 1st edition, 2017. ISBN 3319474723, 9783319474724.

[86] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450. IEEE, 2018.

[87] Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen. Case study: Building a serverless messenger chatbot. In *International Conference on Web Engineering*, pages 75–86. Springer, 2017.

[88] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. Web service composition: a survey of techniques and tools. *ACM Computing Surveys (CSUR)*, 48(3):33, 2016.

[89] William S Levine. *The control handbook: Control system fundamentals*. CRC press, 2010.

[90] Bryan Liston. Resize images on the fly with amazon s3, aws lambda, and amazon api gateway. `https://aws.amazon.com/blogs/compute/resize-images-on-the-fly-with-amazon-s3-aws-lambda-and-amazon-api-gateway/`, 2017. Accessed 2019-03-04.

[91] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. A survey of data-intensive scientific workflow management. *J. Grid Comput.*, 13(4):457–493, 2015.

[92] Shenjun Ma, Alexey Ilyushkin, Alexander Stegehuis, and Alexandru Iosup. Ananke: A q-learning-based portfolio scheduler for complex industrial workflows. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 227–232. IEEE, 2017.

[93] Matt Makai. How to send sms text messages with aws lambda and python 3.6. `https://www.twilio.com/blog/2017/05/send-sms-text-messages-aws-lambda-python-3-6.html`, 2017. Accessed 2019-03-04.

[94] Greg Michaelson. *An introduction to functional programming through lambda calculus*. Courier Corporation, 2011.

[95] Marvin L Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.

[96] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77 (4):541–580, 1989.

[97] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[98] Nima Jafari Navimipour, Amir Masoud Rahmani, Ahmad Habibizad Navin, and Mehdi Hosseinzadeh. Resource discovery mechanisms in grid systems: A survey. *J. Network and Computer Applications*, 41:389–410, 2014.

[99] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[100] Nordstrom. Hello, retail! `https://github.com/Nordstrom/hello-retail`, 2017. Accessed 2018-07-29.

[101] Rory V O'Connor, Peter Elger, and Paul M Clarke. Continuous software engineering—a microservices architecture perspective. *Journal of Software: Evolution and Process*, 29(11):e1866, 2017.

[102] JinSeok Oh and Soo-Mook Moon. Snapshot-based loading-time acceleration for web applications. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 179–189. IEEE Computer Society, 2015.

[103] Object Management Group (OMG). Business process model and notation specification version 2.0. Technical report, 2011.

[104] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[105] Knative Organization. Documentation for users of knative components. `https://github.com/knative/docs/`, 2018.

[106] Marcel Panse. Hello, retail! `https://blog.unless.com/a-case-study-of-teletext-io-the-serverless-startup-c1e08022bdc2`, 2016. Accessed 2018-07-29.

[107] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[108] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4):73, 2018.

[109] Eric S Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional, 2003.

[110] Jan Recker, Marta Indulska, Michael Rosemann, and Peter Green. How good is bpmn really? insights from theory and practice. *Proceedings of the 14th European Conference on Information Systems, ECIS 2006*, 01 2006.

[111] Peter Resnick. Internet message format. Technical report, The Internet Society, 2008.

[112] Cloud Computing RightScale. State of the cloud survey 2018. `https://www.rightscale.com/lp/2018-state-of-the-cloud-report`, 2017. Accessed: 2018-01-30.

[113] Mike Roberts. Serverless architectures. `https://martinfowler.com/articles/serverless.html`, 2018. Accessed 2018-07-29.

[114] Nick Russell, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. *Workflow Patterns: The Definitive Guide*. MIT Press, 2016. ISBN 9780262029827.

[115] Rizos Sakellariou, Henan Zhao, Eleni Tsiakkouri, and Marios D Dikaiakos. Scheduling workflows with budget constraints. In *Integrated research in GRID computing*, pages 189–202. Springer, 2007.

[116] Josep Sampé, Marc Sánchez Artigas, Pedro García López, and Gerard París. Data-driven serverless functions for object storage. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, pages 121–133, 2017.

[117] Amazon Web Services. Aws serverless application model (sam). `https://github.com/awslabs/serverless-application-model/blob/master/versions/2016-10-31.md`, jul 2018. Accessed 2018-07-29.

[118] Siqi Shen, Vincent van Beek, and Alexandru Iosup. Statistical characterization of business-critical workloads hosted in cloud datacenters. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*, pages 465–474, 2015.

[119] Marigianna Skouradaki, Dieter H Roller, Frank Leymann, Vincenzo Ferme, and Cesare Pautasso. On the road to benchmarking bpmn 2.0 workflow engines. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 301–304. ACM, 2015.

[120] Clay Smith. Aws lambda in production: State of serverless report 2017. `https://blog.newrelic.com/product-news/aws-lambda-state-of-serverless/`, 2017. Accessed: 2018-05-05.

[121] James E Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[122] Josef Spillner. Snafu: Function-as-a-service (faas) runtime design and implementation. *CoRR*, abs/1703.07562, 2017. URL `http://arxiv.org/abs/1703.07562`.

[123] Josef Spillner, Cristian Mateos, and David A Monge. Faaster, better, cheaper: The prospect of serverless scientific computing and hpc. In *Latin American High Performance Computing Conference*, pages 154–168. Springer, 2017.

[124] OASIS Standard. Web services business process execution language version 2.0. Technical report, 2007.

[125] Claudia Szabo, Quan Z Sheng, Trent Kroeger, Yihong Zhang, and Jian Yu. Science in the cloud: Allocation and execution of data-intensive scientific workflows. *Journal of Grid Computing*, 12 (2):245–264, 2014.

[126] Andrew S Tanenbaum. *Modern operating system*. Pearson Education, Inc, 2009.

[127] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[128] Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, Michael Adams, and Nick Russell, editors. *Modern Business Process Automation - YAWL and its Support Environment*. Springer, 2010. ISBN 978-3-642-03120-5. URL `http://www.yawlbook.com/home/`.

[129] Berners-Lee Tim. Uniform resource identifier (uri): Generic syntax. Technical report, 2005.

[130] Erwin van Eyk. Four techniques serverless platforms use to balance performance and cost. `https://www.infoq.com/articles/serverless-performance-cost`, 2019. Accessed 2019-03-04.

[131] Erwin van Eyk and Soam Vasani. Optimizing latency in function-as-a-service with distributed promises. `http://erwinvaneyk.nl/presentation-hotcloudpef2018-distributed-promises/`, 2018. Accessed: 2018-01-30.

[132] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. The spec cloud group's research vision on faas and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 1–4. ACM, 2017.

[133] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. A spec rg cloud group's vision on the performance challenges of faas cloud architectures. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 9 - 13, 2018*, 2018.

[134] Erwin van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uta, and Alexandru Iosup. Serverless is more: From paas to present cloud computing. *IEEE Internet Computing*, 22(5), 2018. Sep/Oct issue.

[135] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. The spec-rg reference architecture for faas: From microservices and containers to serverless platforms. 2019.

[136] Laurens Versluis, Erwin van Eyk, and Alexandru Iosup. An analysis of workflow formalisms for workflows with complex non-functional requirements. In *Proceedings of the First International Workshop on Hot Topics in Cloud Computing Performance, in conjuction with ICPE, Berlin, Germany, April 9, 2018*, 2018.

[137] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146. USENIX, 2018.

[138] Mathias Weske. Business process management architectures. In *Business Process Management*, pages 333–371. Springer, 2012.

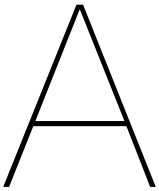
[139] CNCF Serverless WG. Cncf wg-serverless whitepaper v1.0. `https://github.com/cncf/wg-serverless/blob/master/whitepaper/cncf_serverless_whitepaper_v1.0.pdf`, 2018. Accessed 2018-07-29.

[140] Zhiqiang Yan, Hajo A Reijers, and Remco M Dijkman. An evaluation of bpmn modeling tools. In *International Workshop on Business Process Modeling Notation*, pages 121–128. Springer, 2010.

[141] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.

[142] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, pages 249–265, 2014.

[143] Michael zur Muehlen and Jan Recker. How much bpmn do you need. `http://www.bpm-research.com/2008/03/03/how-much-bpmn-do-you-need`, 2008. Accessed 2018-07-29.



Serverless Workflow Language

The Serverless Workflow Language (SWL) is the workflow language that we designed specifically to express serverless workflows. In this work we introduced the conceptual design of SWL in Section 5.2. In Chapter 7, we provided an overview of main the reference implementation of SWL with SWL-YAML.

We provide a further, more in-depth specification of SWL-YAML in documentation of the Fission Workflows project at `https://github.com/fission/fission-workflows/tree/master/Docs`. It includes an overview of all the pre-provided tasks (such as, loops, conditional branches, etc.), and a guide to the workflow expressions.

B

Fission Workflows Publicity and Resources

This section highlights some of the publications and events in the industry discussing or mentioning Fission Workflows (as of May 2019).

B.1. Blog Posts and Articles

1. *Platform9's Fission Workflows makes it easier to write complex serverless applications* - Frederic Lardinois, 2017. https://techcrunch.com/2017/10/03/platform9s-fission-workflows-makes-it-easier-to-write-complex-serverless-applications
2. *Four Techniques Serverless Platforms Use to Balance Performance and Cost* - Erwin van Eyk, 2019. https://www.infoq.com/articles/serverless-performance-cost/
3. *Send in the Robots: Platform9 Automates Serverless Workflow* - Dan Meyer, 2017. https://www.sdxcentral.com/articles/news/send-in-the-robots-platform9-automates-serverless-workflow/2017/10/
4. *Platform9 Brings Workflows to Serverless Computing* - Mike Vizard, 2017. https://devops.com/platform9-brings-workflows-serverless-computing/
5. *Serverless technology obfuscates workflows, performance data* - Mike Matchett, 2018. https://searchitoperations.techtarget.com/opinion/Serverless-technology-obfuscates-workflows-performance-data
6. *Fission - Serverless Functions and Workflows with Kubernetes and NATS* - Soam Vasani, 2017. https://nats.io/blog/serverless-functions-and-workflows-with-kubernetes-and-nats/
7. *SPLab Colloquium on Serverless Scientific Computing* - Josef Spillner, 2018. https://blog.zhaw.ch/icclab/splab-colloquium-on-serverless-scientific-computing/
8. *What is Serverless and what it means for you – Part 3: Kubernetes and Serverless* - Vamsi Chemitiganti, 2018. https://www.itproportal.com/features/what-is-serverless-and-what-it-means-for-you-part-3-kubernetes-and-serverless/
9. *A Comparison of Serverless Frameworks for Kubernetes: OpenFaas, OpenWhisk, Fission, Kubeless and more* - Phil Winder, 2018. https://winderresearch.com/a-comparison-of-serverless-frameworks-for-kubernetes-openfaas-openwhisk-fission-kubeless-and-more/

10. *Creating Visualizations from Kubernetes metrics with Fission.io and some notes on Fission Workflows* - Joseph D. Marhee, 2017. https://medium.com/@jmarhee/creating-visualizations-from-kubernetes-metrics-with-fission-io-and-some-notes-on-fission-workflows-62fcc4f2502f
11. *FaaS Function Composition with Fission Workflows and NATS* - Soam Vasani & Timirah James, 2018. https://blog.fission.io/posts/fission-workflows-with-nats/
12. *Function Composition: What It Means, and Why You Should Care* - Timirah James, 2018. https://blog.fission.io/posts/function-composition/
13. *Getting Started: Composing Serverless Functions with Fission Workflows (Part 1)* - Timirah James, 2018. https://blog.fission.io/posts/fission-workflows-pt-1/
14. *Getting Started: Composing Serverless Functions with Fission Workflows (Part 2)* - Timirah James, 2018. https://blog.fission.io/posts/fission-workflows-pt-2/
15. *Follow That Transaction! APM Across Clouds* - Mike Matchett, 2017. http://smallworldbigdata.com/2017/11/follow-that-transaction-apm-across-clouds/
16. *Fission Workflows: Using Serverless For Processing Kubernetes Metrics* - Chris Wright, 2017. https://platform9.com/blog/fission-workflows-using-serverless-for-processing-kuberrnetes/
17. *Everything you need to know about serverless: What does the future hold?* - Erwin van Eyk (interview), 2019. https://jaxenter.com/servereless-interview-van-eyk-158600.html
18. *Fission Workflows brings serverless technology to app development* - Erwin van Eyk, 2017. https://sdtimes.com/fission-workflows/fission-workflows-brings-serverless-technology-app-development/

B.2. Conferences and Meet-ups

1. **KubeCon 2019 Barcelona**: *Serverless Operations: From Dev to Production* - Erwin van Eyk. https://www.youtube.com/watch?v=5ftE6LBdkGY
2. **KubeCon 2018 Copenhagen**: *Function composition in a serverless world* - Erwin van Eyk & Timirah James. http://erwinvaneyk.nl/kubecon18eu-function-compoosition-in-a-serverless-world/
3. **KubeCon 2018 Shanghai**: *Serverless performance on a budget* - Erwin van Eyk. http://erwinvaneyk.nl/kubecon-china-2018-serverless-performance/
4. **Serverless Architecture Conference 2019**: *Serverless Performance on a budget* - Erwin van Eyk. http://erwinvaneyk.nl/eccsa-2018-serverless-performance/
5. **JAX DevOps 2019**: *Going FaaSter: Cost-Performance Optimizations of serverless on Kubernetes* - Erwin van Eyk. https://devops.jaxlondon.com/cloud-platforms/going-faaster-cost-performance-optimizations-of-serverless-on-kubernetes/
6. **European Symposium on Serverless Computing and Applications (ESCCA) 2018**: *Performance on a budget* - Erwin van Eyk. http://erwinvaneyk.nl/eccsa-2018-serverless-performance/
7. **VelocityCon 2018**: *Function composition in a serverless world* - Soam Vasani, Timirah James. https://conferences.oreilly.com/velocity/vl-ca-2018/public/schedule/detail/66827
8. **OSCon 2018**: *Approaches to composing FaaS functions together* - Soam Vasani. https://conferences.oreilly.com/oscon/oscon-or-2018/public/schedule/detail/67760

9. **Serverless Computing London 2018**: *Function Composition in a Serverless World* - Timirah James. http://2018.serverlesscomputing.london/sessions/function-composition-serverless-world/
10. **SATURN 2019**: *Function Composition in a Serverless World* - Josh Hurt. https://sched.co/LY4o

B.3. Podcasts and Screencasts

1. *TGI Kubernetes 013: Serverless with Fission* - Joe Beda, 2017. https://www.youtube.com/watch?v=3XgBB4mATNM
2. *TGI Kubernetes 020: Argo workflow system* - Joe Beda, 2018. https://www.youtube.com/watch?v=M_rxPPLG8pU
3. *Fission Workflows: Composing Complex Serverless Functions* - Amrish Kapoor, 2018. https://www.youtube.com/watch?v=90xpDv3zf6Y
4. *Fission Workflows with Slack Demo* - Erwin van Eyk, 2018. https://www.youtube.com/watch?v=v-m6E9kvGTQ
5. *Fission Workflows Overview* - Erwin van Eyk, 2018. https://www.youtube.com/watch?v=dqIzKnLg95Y

C

Experiment Configurations

This appendix contains a comprehensive overview of the configurations used in the experimental evaluation (Chapter 8). Section C.1 contains the default configuration for the experiments. Each experiment uses this default configuration, and (optionally) has a subsequent section describing the changes to the configuration that we made.

C.1. Fission Workflows Experiments

This section contains the configurations used for all experiments of the Fission Workflows prototype. Unless specified otherwise, all these experiments use these configurations.

C.1.1. NATS Streaming

Table C.1 describes the configuration used for the NATS Streaming-based event store. For the FILE and SQL-backed NATS Streaming clusters used in the Fault Tolerance experiments, the default settings were used.

C.1.2. SimFaaS

The default configuration used for SimFaaS in the experiments is depicted in Table C.2. For the experiments, we used the `sleep` function, which is built into SimFaaS.

C.1.3. Fission

Table C.3 contains the configuration used for the Fission (and the underlying Kubernetes cluster) in the experiments. We deployed Fission on a regular Kubernetes cluster, provided by Google Kubernetes Engine. To emulat function executions, we defined a Python function (`sleep`), which is listed in Listing 2 with the deployment configuration listed in Table C.4.

C.1.4. Fission Workflows

The default configuration is used for Fission Workflows is described in Table C.5. This configuration is used unless otherwise specified. For the experiments that rely on the Chronos workload, the Fission Workflow's workflow to implement the Chronos workflow is listed in Listing 3.

C.2. Azure Experiments

This section describes our experimentation process to evaluate Azure in the realistic experiment (see Section 8.2.5). First, in Section C.2.1, we provide a description of the experimental setup used, and the configuration used. Then, we include the general steps that we have taken to be able to run the Chronos workload on the system, obstacles that we ran into, and the workarounds that we used to move beyond the obstacles (in Section C.2.2).

Configuration key	Configuration value
version	0.11.2
store	MEMORY
max_channels	0
max_subs	0
max_msgs	0
max_bytes	0
max_age	0s
max_inactivity	0
cluster size	1

Table C.1: The configuration used for the NATS Streaming cluster used in the experiments.

Configuration key	Configuration value
version	1.0.0
cold-start	0
keep-warm	0

Table C.2: The configuration used for the SimFaaS deployment used in the experiments.

```
import time

from flask import request


def main():
    time.sleep(float(request.args.get("runtime")))
    return {
        "statusCode": 200,
    }
```

Listing 2: The Python function, named `sleep`, used to emulate the tasks of the Chronos trace in Fission.

C.2.1. Experiment Setup

In order to run the Chronos workload in Azure we use Azure Functions as the FaaS platform, and Azure Logic Apps as the WMS. In this section we describe the configuration used.

Azure Functions We implemented the emulator function as a JavaScript function in Azure Functions Functions. The source code of this function can be found in Listing 4. The function was deployed using the configuration listed in Table C.6.

Azure Logic Apps Similarly to Azure Functions, Azure Logic Apps does not require upfront operational setup. Furthermore, there is little configuration exposed to the user. The configuration used is listed Table C.7. In this case, the `high-throughput` increases the maximum concurrency. The implementation of the Chronos workflow in Azure Logic Apps is listed in Listing 6.

C.2.2. Experiment Process

Usage of Azure Functions and Azure Logic Apps had a couple of pitfalls that we encountered during the experimentation: First, to keep the languages similar on all FaaS platforms, we initially choose to deploy each function as a Python function. However, whereas the performance of languages on AWS is comparable, on Azure the large differences in the performance of the different languages, an issue that has been investigated in detail by Lee et al. [86]

Second, a similar situation holds true for the operating system underlying the function. Initially, we

Configuration group	Configuration key	Configuration value
Fission	version	1.3.0
	deploymentVariant	fission-all
Kubernetes cluster	Master version	1.12.7-gke.10
	Vertical Pod Autoscaling	Disabled
	Master zone	europe-west2-a
	Node zones	europe-west2-a
	Cloud TPU	Disabled
	Network	default
	Subnet	default
	VPC-native	Disabled
	Workload Identity	Disabled
	Network policy	Disabled
Kubernetes node pool	Size	4
	Node version	1.10.11-gke.1
	Node image	Container-Optimized OS (cos)
	Machine type	n1-standard-1 (1 vCPU, 3.75 GB RAM)
	Automatic node upgrades	Disabled
	Automatic node repair	Enabled
	autoscaling	Off
	Preemptible nodes	Disabled
	Boot disk type	Standard persistent disk
	Boot disk size in GB (per node)	100
	Local SSD disks (per node)	0
	Sandbox with gVisor	Disabled

Table C.3: The configuration used for the Fission deployment used (and the underlying Kubernetes cluster) in the experiments.

Configuration key	Configuration value
Environment	fission/python-env:1.3.0
ExecutorType	Pooled

Table C.4: The configuration used to deploy the Fission function described in Listing 2.

choose to deploy the function on the Linux OS, to keep things similar to the other FaaS platforms. However, after our experiment returned diseastruous results, the cause turned out to be the choice of the OS. At the time of the experiment, the Linux OS was still in a preview mode, which included a (hidden) maximum limit of two function instances.

C.3. AWS Experiments

This section describes our experimentation process to evaluate AWS in the realistic experiment (see Section 8.2.5). First, in Section C.3.1, we provide a description of the experimental setup used, and the configuration used. Then, we include the general steps that we have taken to be able to run the Chronos workload on the system, obstacles that we ran into, and the workarounds that we used to move beyond the obstacles (in Section C.3.2).

C.3.1. Experiment Setup

In order to run the Chronos workload in AWS we use AWS Lambda as the FaaS platform, and AWS Step Functions as the WMS. In this section we describe the configuration used.

AWS Lambda We implemented the emulator function as a Python function in AWS Lambda. The source code of this function can be found in Listing 5. The function was deployed using the configuration listed in Table C.8.

```
apiVersion: 1
output: t3
tasks:
  t1:
    run: "fission://sleep"
    inputs:
      query:
        runtime: "{ param('query').t1 }"
  t2:
    run: "fission://sleep"
    inputs:
      query:
        runtime: "{ param('query').t2 }"
    requires:
      - t1
  t3:
    run: "fission://sleep"
    inputs:
      query:
        runtime: "{ param('query').t3 }"
    requires:
      - t2
```

Listing 3: The workflow that implements the Chronos workflow in Fission Workflows. The Fission function `sleep`, which emulates the Chronos tasks, is defined in Figure 7.

```
module.exports = async function (context, req) {
    runtime = parseFloat(req.query.runtime) * 1000;
    await sleep(runtime);
    context.res = {
        status: 200,
        body: runtime
    };
};

function sleep(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}
```

Listing 4: The JavaScript function, named `sleep`, used to emulate the tasks of the Chronos trace in Azure Functions.

Configuration key	Configuration value
version	0.6.0
WorkflowCacheSize	10000
InvocationsCacheSize	100000
executorMaxParallelism	1000
executorMaxTaskQueueSize	100000
invocationJobQueueMaxSize	10000
workflowStorePollInterval	1 minute
invocationStorePollInterval	1 second
workflowSubscriptionBuffer	50
invocationSubscriptionBuffer	1000
schedulerPolicy	horizon

Table C.5: The default configuration used for Fission Workflows in the experiments.

Configuration key	Configuration value
Zone	Europe West
Operating System	Windows
Runtime	JavaScript
Memory	128 MB

Table C.6: The configuration of the Azure Function listed in Figure 4.

AWS Step Functions Similarly to AWS Lambda, AWS Step Functions does not require any extensive setup to use. Furthermore, there is little configuration exposed to the user. The only meaningful configuration, the deployment zone, is set to ”us-east-1”. The reason for the deployment in the US, rather than in a EU region, is that during the experimentation AWS Step Functions was not supported in EU regions. The implementation of the Chronos workflow in AWS Step Functions is depicted in Listing 6.

C.3.2. Experiment Process

In contrast to the other two cloud providers, the Chronos experiment ran without any problems on the AWS stack. The only thing that had to be adapted in the function is the parsing of the input, which supports getting the runtime from either the body or the query. This is because AWS Step Functions' canonical approach to passing data to functions is not based on HTTP. Although it is possible to construct the event in the step function in such a way that it resembles a HTTP request, we only learned about this after the experimentation had completed.

C.4. Google Cloud Experiments

This section describes our experimentation process to evaluate Google Cloud in the realistic experiment (see Section 8.2.5). In this section we include the general steps that we have taken to be able to run the Chronos workload on the system, obstacles that we ran into, and the workarounds that we used to move beyond the obstacles.

As described in the experimental setup (see Section 8.1), we choose for Google Cloud Composer [15] for the Workflow Management System, and Google Cloud Functions for the FaaS platform. The reasons for this choice were two-fold: (1) Google Cloud is one of the key cloud providers in the domain [112]; and, (2) Google Cloud Composer is a managed version of Apache Airflow [60], with no changes made to the core. Since Airflow is representative for the state-of-the-art in open-source WMSs (see Section 4.3), this choice enables us to compare the prototype with the open-source state-of-the-art.

In the remainder of this section we describe the process of this experiment in three sections. In Section C.4.1, we cover the process of setting up the experiment. Then, in Section C.4.2, we describe the obstacles and events that occured while running iterations of the experiment.

```python
import json, time

def lambda_handler(event, context):
    try:
        runtime = float(event["queryStringParameters"]["runtime"])
    except KeyError:
        try:
            runtime = float(event["runtime"])
        except KeyError:
            runtime = 0.0

    time.sleep(runtime)
    return {
        "statusCode": 200,
        "headers": { 'Content-Type': 'application/json' },
        "body": json.dumps(runtime),
    }
```

Listing 5: The Python function, named `SimTask`, used to emulate the tasks of the Chronos trace in Fission.

```
{
  "StartAt": "1",
  "States": {
    "1": {
      "Type": "Task",
      "Resource": "<FUNCTION_ID>:function:SimTask",
      "InputPath": "\$.t1",
      "ResultPath": "\$.output.task1",
      "Next": "2"
    },
    "2": {
      "Type": "Task",
      "Resource": "<FUNCTION_ID>:function:SimTask",
      "InputPath": "\$.t2",
      "ResultPath": "\$.output.task2",
      "Next": "3"
    },
    "3": {
      "Type": "Task",
      "Resource": "<FUNCTION_ID>:function:SimTask",
      "InputPath": "\$.t3",
      "ResultPath": "\$.output.task3",
      "End": true
    }
  }
}
```

Listing 6: The AWS Step Function definition created to implement the Chronos workflow. The `<FUNCTION_ID>` contains the actual AWS Resource Name (ARN) to the function, and has been partly redacted in this example.

Configuration key	Configuration value
Zone	Europe West
High-throughput	Enabled

Table C.7: The configuration of the Azure Logic App workflow listed in Figure 9.

Configuration key	Configuration value
Zone	us-east-1
Runtime	Python 3.6
Memory	128 MB
Timeout	5 min
Max concurrency	1000

Table C.8: The configuration used to deploy the AWS Lambda function described in Listing 5.

C.4.1. Experiment Setup

This section describes the setup of the experiment infrastructure. It further contains the configuration and code used to evaluate Google Cloud Composer and Google Cloud Functions.

Google Cloud Functions Following the serverless principles [132], the FaaS platform (Google Cloud Functions) does not require any enabling or setting up of the platform. We implemented the emulator function as a JavaScript function in Google Cloud Functions. The source code of this function can be found in Listing 7. The function was deployed using the configuration listed in Table C.10.

Google Cloud Composer The WMS, Google Cloud Composer or Apache Airflow, does require an extensive configuration to setup, which is documented in Table C.9. Rather than being fully-managed by the cloud provider, Google Cloud deploys Airflow onto a new Kubernetes cluster. It exposes this Kubernetes cluster to the user, to allow the user to augment the *managed* airflow deployment using common Kubernetes tools and services.

To implement the Chronos workflow, we implemented a workflow in the Python-based Airflow format, which is listed in Listing 8. There are several things to highlight about this workflow. First, we implemented the tasks as `BashOperator`. In hindsight, it might have been better to use the `HTTPOperator`. However, both operators are simple implementations, which makes us believe that the difference should not impact the results significantly.

Second, for this Airflow workflow, we needed to workaround the workflow definition to enable it to be used for serverless workflows. On line 9, we were required to input a `start_date` for the workflow. This start date needs to be set in the past, otherwise Airflow will not execute the workflow. Similarly, on Line 15-17 we had to set the `schedule_interval` to zero, because, otherwise, Airflow would backfill the workflow from the `start_date` to now.

```
1 exports.sleeper = async (req, res) => {
2   let runtime = req.query.runtime;
3   await sleep(runtime * 1000);
4   res.status(200).send(runtime);
5 };
6
7 function sleep(ms) {
8   return new Promise(resolve => setTimeout(resolve, ms));
9 }
```

Listing 7: The Python function, named `sleep`, used to emulate the tasks of the Chronos trace in Google Cloud Functions.

```
from datetime import datetime, timedelta

from airflow import DAG
from airflow.operators.bash_operator import BashOperator

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2015, 1, 1),
    'retries': 0,
}

function_url = "https://<REDACTED>.cloudfunctions.net/sleep"

dag = DAG('chronos',
          default_args=default_args,
          schedule_interval=timedelta(days=0))

t1 = BashOperator(
    task_id="t1",
    bash_command='curl -Ls https://' + function_url
                 + '?runtime={{ dag_run.conf["t1"]["runtime"] }}',
    dag=dag)

t2 = BashOperator(
    task_id="t2",
    bash_command='curl -Ls https://' + function_url
                 + '?runtime={{ dag_run.conf["t2"]["runtime"] }}',
    dag=dag)

t3 = BashOperator(
    task_id="t3",
    bash_command='curl -Ls https://' + function_url
^^I^^I             + '?runtime={{ dag_run.conf["t3"]["runtime"] }}',
    dag=dag)

t1 >> t2 >> t3
```

Listing 8: The Airflow workflow created to represent the Chronos workflow. The `function_url` contains the actual url to the function, and has been partly redacted in this example.

Configuration group	Configuration key	Configuration value
Google Cloud Composer	Zone	europe-west1-b
	Worker Node count	5
	Image version	composer-1.7.1-airflow-1.10.2
	Python version	3
Kubernetes cluster	Master version	1.12.8-gke.6
	Vertical Pod Autoscaling	Disabled
	Master zone	europe-west1-b
	Node zones	europe-west1-b
	Cloud TPU	Disabled
	Network	default
	Subnet	default
	VPC-native	Disabled
	Workload Identity	Disabled
	Network policy	Disabled
Kubernetes node pool	Size	5
	Node version	1.12.8-gke.6
	Node image	Container-Optimized OS (cos)
	Machine type	n1-standard-1 (1 vCPU, 3.75 GB RAM)
	Automatic node upgrades	Disabled
	Automatic node repair	Enabled
	autoscaling	Off
	Preemptible nodes	Disabled
	Boot disk type	Standard persistent disk
	Boot disk size in GB (per node)	20
	Local SSD disks (per node)	0
	Sandbox with gVisor	Disabled

Table C.9: The relevant configuration of Google Cloud Composer and the underlying Google Kubernetes Engine cluster. For details on the individual settings, see the Google Cloud documentation.

C.4.2. Experiment Process

Executing the Chronos workload on Google Cloud Composer/Airflow is cumbersome and problematic. Next having to work around the execution model of Airflow (see Section C.4.1), we have spend considerable time to try to tweak Google Cloud Composer/Airflow to be able to run the Chronos workload trace.

Attempt 1: Using the default configuration. Given the limited time available for the evaluation, one of the principles with all evaluated systems is that we use the default configuration, where possible. So, our initial attempt was to try to run the Chronos workload using a default setup: a default Google Cloud Composer deployment on 5 nodes.

We found out that the functionality for invoking the workflow manually already does not scale to the Chronos workload—in which we invoke the workflow on average 6 times per second. Airflow does not expose a public API for manually invoking workflows. Moreover, the (experimental) Airflow CLI does not support parallel submissions, and is too slow to invoke the workflow more than once every couple of seconds.

Attempt 2: Inserting invocations directly into the Airflow database. To overcome the initial obstacle of being able to invoke the Chronos workflow at the required QPS, we decided to bypass Airflow, and add the workflow invocation directly to the SQL Database that backs the Airflow deployment. By inserting new rows to the `dag_run` table with the column `scheduled_at` matching the start date according to the Chronos workload trace.

Although this works for a self-deployed version of Airflow, Google Cloud Composer provides the next obstacle. Instead of exposing the Airflow database as a regular Google SQL Database, it hides—by

Configuration key	Configuration value
Zone	europe-west1-b
Memory	128 MB
Timeout	60 seconds
Maximum function instances	unlimited
Runtime	Node.js 8

Table C.10: The configuration of the Google Cloud Function listed in Listing 7.

design—the database completely from the user. This prevents us from inserting the Chronos workload trace directly to the database.

Attempt 3: Hijacking the `airflow-sqlproxy` to access the Google Cloud Composer database. Even though Google Cloud Composer hides the SQL databse from the user, it does not fully prevent you from accessing it. This is because the Airflow instance is deployed in a Kubernetes cluster completely accessible by the user. To be able to interact with the database, we connect to the Kubernetes cluster that manages Airflow. In this cluster, Google Cloud Composer has deployed, alongside it, a service called `airflow-sqlproxy`. This service is responsible for exposing the remote Google SQL Database to the Airflow deployment. We use `kubectl exec` to spawn and connect to a shell within the `airflow-sqlproxy` service. Once inside this service, we can launch a `mysql` session, and interact with the Google Cloud Composer database.

Now, even though we can insert the full workload trace into the database, it becomes clear that Airflow is not built for these frequent workflow executions. Only a few of the thousands of workflow executions of Chronos are actually executed by Airflow.

Attempt 4: Tweaking performance settings. Although we initially planned not to make any changes to the default settings for the experiment, we decided to try to tweak the settings of Airflow a bit to at least get it to run the Chronos workload at all. First, we bumped up the artificial limits on the number of workflow invocations and tasks that can be executed in parallel (changing `core.max_active_runs_per_dag`, `core.dag_concurrency`, and `core.parallelism`). Then, we inferred from the logs, that the scheduler evaluates the executions too inefficiently. To improve the scheduler performance, we increased `scheduler.max_threads` to the number of cores of the machine. Then, to improve the worker performance we tried adding additional worker nodes, and increasing settings of the job queue (Celery), by increasing `celery.worker_concurrency`. However, despite these attempts, we could not get Airflow to run the Chronos trace.

Attempt 5: Simplifying the workload. At this point, we have by already exceeded by far the time assigned to this experiment. Although, with more time and expert knowledge, we might be able to tweak Airflow to such an extend that it can handle the Chronos workload, at this point we halted any further optimization attempts. Especially since for the other evaluated platforms, no attempts were made to tweak the performance at all.

So, to at least get an indication of how Airflow performs, we simplified the Chronos workload. We changed the execution timestamps to ensure that only one workflow is executed at a time. As presented in the results, Airflow is able to run this extremely simplified version of Chronos, running the workload at 0.016 QPS rather than the average 6 QPS.

```
{
    "definition": {
    "$schema": "https://schema.management.azure.com/providers/
                Microsoft.Logic/schemas/2016-06-01/workflowdefinition.json#",
        "actions": {
        "Response": {
            "inputs": {
                "body": "@workflow()['run']['name']",
                    "statusCode": 200
            },
            "kind": "Http",
                "runAfter": {
                "t3": [
                    "Succeeded"
                ]
            },
            "type": "Response"
        },
        "t1": {
            "inputs": {
                "function": { "id": "<FUNCTION_ID>" },
                "method": "POST",
                    "queries": {
                    "runtime": "@triggerBody()?['t1']?['runtime']"
                }
            },
            "runAfter": {},
            "type": "Function"
        },
        "t2": {
            "inputs": {
                "function": { "id": "<FUNCTION_ID>" },
                "method": "POST",
                    "queries": {
                    "runtime": "@triggerBody()?['t2']?['runtime']"
                }
            },
            "runAfter": {
                "t1": [
                    "Succeeded"
                ]
            },
            "type": "Function"
        },
        "t3": {
            "inputs": {
                "function": { "id": "<FUNCTION_ID>" },
                "queries": {
                    "runtime": "@triggerBody()?['t3']?['runtime']"
                }
            },
            "runAfter": {
                "t2": [
                    "Succeeded"
                ]
            },
```

```
                "type": "Function"
            }
        },
        "contentVersion": "1.0.0.0",
            "outputs": {},
        "parameters": {},
        "triggers": {
            "manual": {
                "inputs": {
                    "schema": {
                        "properties": {
                            "t1": {
                                "properties": {
                                    "runtime": {
                                        "type": "integer"
                                    }
                                },
                                "type": "object"
                            },
                            "t2": {
                                "properties": {
                                    "runtime": {
                                        "type": "integer"
                                    }
                                },
                                "type": "object"
                            },
                            "t3": {
                                "properties": {
                                    "runtime": {
                                        "type": "integer"
                                    }
                                },
                                "type": "object"
                            }
                        },
                        "type": "object"
                    }
                },
                "kind": "Http",
                    "type": "Request"
            }
        }
}
```

Listing 9: The Azure Logic Apps definition created to implement the Chronos workflow. The <FUNCTION_ID> contains the Azure resource name of the function, and has been partly redacted in this example.