A Language-Based Approach for Web Service Composition

Elyas Ben Hadj Yahia

ACADÉMIE DE BORDEAUX
UNIVERSITÉ DE **B**ORDEAUX
Sciences et **T**echnologies

THÈSE

Présentée au Laboratoire Bordelais de Recherche en Informatique pour obtenir le grade de Docteur de l'Université de Bordeaux

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Mathématiques et Informatique**

A Language-Based Approach for Web Service Composition

par

Elyas BEN HADJ YAHIA

Soutenue le 28 Novembre 2017, devant le jury composé de :

Directeur de thèse
Laurent RÉVEILLÈRE, Professeur ..... Université de Bordeaux, France

Rapporteurs
Luís VEIGA, Professeur ..... Université de Lisbonne, Portugal
Romain ROUVOY, Professeur ..... Université de Lille, France

Examinateurs
Mohamed MOSBAH, Professeur ..... Bordeaux INP, France
Yérom-David BROMBERG, Professeur ..... Université de Rennes, France
Jean-Rémy FALLERI, Maître de conférences ..... Bordeaux INP, France

Membres invités
Alain CADOT, Président-directeur général ..... CIS Valley, France
Raphaël CHEVALIER, Directeur associé ..... CProDirect, France

Abstract

In light of the recent advances in the field of web engineering, along with the decrease of cost of cloud computing, service-oriented architectures rapidly became the leading solution in providing valuable services to clients. Following this trend, the composition of third-party services has become a successful paradigm for the development of robust and rich distributed applications, as well as automating business processes. With the availability of hundreds of thousands of web services and APIs, such integrations become cumbersome and tedious when performed manually. Furthermore, different clients may require different integration requirements and policies, which further complexifies the task. Moreover, providing such a solution that is both robust and scalable is a non-trivial task. Therefore, it becomes crucial to investigate how to efficiently coordinate the interactions between existing web services. As such, this thesis aims at investigating the underlying challenges in web service composition in the context of modern web development practices. We present an architectural framework to support the specification of web service compositions using a language-based approach, and show how we support their execution in a scalable manner using MEDLEY, a lightweight, event-driven platform.

Keywords: *Service composition, Orchestration, Domain-specific languages, Microservices, Distributed systems*

Résumé

Au vu des dernières avancées dans le domaine de l'ingénierie web, ainsi qu'avec la baisse de coût du *cloud computing*, les architectures orientées services sont rapidement devenues la solution prépondérante pour fournir des services à valeur ajoutée aux clients. Suite à cette tendance, la composition de services tiers est devenue un paradigme de référence pour le développement d'applications robustes et riches, ou encore pour l'automatisation de processus métiers. Avec la disponibilité de centaines de milliers de services et APIs web, la réalisation de telles intégrations devient lourde et fastidieuse quand effectuée manuellement. Par ailleurs, chaque client peut exiger des besoins et politiques d'intégration différentes, ce qui complexifie davantage la tâche. De plus, fournir une telle solution qui soit à la fois robuste et *scalable* est une tâche non-triviale. Il est donc primordial d'étudier comment coordiner de manière efficace les intéractions entre les services web existants. Ainsi, cette thèse vise à étudier les problématiques liées à la composition de services web dans le contexte des pratiques de développement web modernes. Nous présentons un cadre architectural permettant la spécification de compositions de services web grâce à une approche orientée langage, et montrons comment supporter leur exécution de manière *scalable* grâce à MEDLEY, une plateforme légère et orientée événements.

Mots clés : *Composition de services, Orchestration, Langages métiers, Microservices, Systèmes distribués*

LaBRI — 351, Cours de la Libération — 33400 Talence — France

Contents

CHAPTER

1 Introduction

This chapter introduces the scope of the work achieved throughout this PhD thesis. Carried out under the CIFRE [1] industrial partnership contract with the French company CPRODIRECT, this thesis aims at studying CPRODIRECT's recent interest in web service orchestration. With the increasing popularity of service-oriented architectures, it becomes crucial to investigate how to efficiently coordinate the interactions between existing web services. As such, we present in this chapter the context of our work, the underlying challenges and our main contributions.

Contents

1. CIFRE: *Convention Industrielle de Formation par la REcherche*, a French funding grant aimed at promoting collaborations between national companies and public research institutions.

1.1 Context: CPRODIRECT

Based in the outskirts of Bordeaux, CPRODIRECT is a web development agency specialized in consulting services, with an emphasis on e-commerce and marketing activities. The center of activity of the company revolves around providing valuable services to their clients, tailored to suit their needs. The company thrives on integrating existing web services to provide an added value to their clients. This enables CPRODIRECT to leverage well-established, high-quality web services to provide relevant solutions to their clients in a timely fashion. However, with the availability of hundreds of thousands of web services and APIs, such integrations become cumbersome and tedious when performed manually. Furthermore, each client may require different integration requirements and policies, which further complexifies the task. Typically, common use cases include automating business processes across multiple applications by reacting to specific external events, propagating and transforming the data along the way according to the client's requirements. For instance, this can consist in monitoring social networks for negative comments about the client's product or brand, then creating an issue in a dedicated CRM (Client Relation Management) tool, and finally notifying a sales representative in order to address the issue as soon as possible. Moreover, providing such a solution that is both robust and scalable is a non-trivial task. The work performed during this thesis lays ground for addressing the underlying issues in web service composition.

1.2 Challenges in web service composition

Over the past decade, distributed applications have been evolving at a frantic pace, critically relying on integrating altogether a plethora of composable services to offer a host of new functionalities with an added value. The abundance of web services available online led researchers and businesses alike to leverage their potential in a number of ways [Alonso et al., 2004]. Historically, several solutions were proposed to address this issue. For instance, BPEL (Business Process Execution Language) used to be the reference solution to orchestrate SOAP services, and has been the subject of a multitude of studies and industrial applications [Andrews et al., 2003]. However, SOAP services are rapidly being deprecated today, in favor of the more flexible REST architectural style [Fielding, 2000]. Moreoever, there are inherent and fundamental differences between legacy web services (SOAP) and the more modern architectural style (REST) for web services in terms of specifications, toolings and best practices. This brings forth its own set of challenges in the context of web service composition.

1.2.1 Orchestrating modern web services

Since the early days of distributed computing there was a primitive notion of services that took its origins from RPC mechanisms [Nelson, 1981]. The concept of services was significantly refined across the last decades to have a strong impact on the distributed computing landscape, particularly due to the emergence of the Service-Oriented Architecture (SOA) paradigm. Founded on the Web Service stack (as defined by the WS-* specifications), SOA aims at providing an architectural framework for encapsulating business logic and exposing it through standardized interfaces over the network.

From a higher perspective, SOA has promoted at least two major trends that have a long term impact. First, it has promoted a standardized way to build an application (that can itself be seen as a service) as a set of well specified, independent, self-contained and loosely coupled services that work altogether in concert. Second, it has proven that services act as a valuable paradigm to design complex applications.

As a result, we live in a service-oriented world. Applications ranging from the simplest smartphone application to the web's most complex one strive, in one way or another, to interact with value-added services, potentially made themselves from other services. In other terms, applications are increasingly built using the SOA paradigm and integrate a myriad of composable services. Furthermore, with the wide expansion of cloud providers, IaaS (Infrastructure-as-a-Service) and PaaS (Platform-as-a-Service) offerings have become more accessible and affordable alternatives compared to self-hosted solutions [Zhang et al., 2010]. For instance, cloud providers such as AWS [2], GAE [3], Heroku [4] and DigitalOcean [5] all offer an easy and affordable way for businesses and individuals alike to rapidly deploy, monitor and manage their services on reliable infrastructure.

As services are autonomous and deployed, undeployed and upgraded independently from each other, SOA enables application developers to have a fine-grained control on how to smoothly update their applications and how to make them scalable in a production environment. Hence, nowadays, the development of SOA-based applications goes hand in hand with continuous service development and continuous service integration practices [Fowler and Lewis, 2014]. This new trend coupled with the steady proliferation of services is not without challenges, and potentially obsoletes the traditional vision of SOA [Newman, 2015], along with their classical implementations based on the Web Services (WS-*) specifications. For instance, these long-standing specifications propose standards for defining web services (SOAP: Simple Object Access Protocol), and for defining orchestrations between these services (BPEL: Business Process Execution Language). However, the use of these *de facto* standards as a workflow to compose a plethora of services may be inadequate according to the developers' expectations. In fact, BPEL is a low-level

2. https://aws.amazon.com
3. https://cloud.google.com/appengine
4. https://www.heroku.com
5. https://www.digitalocean.com

and verbose language that describes *how* services need to be composed instead of defining *what* should be realized. Clients need to statically declare in advance the services they depend on to carry out the required orchestration. Then, they have to explicitly specify how to programmatically bind to these services, along with the control flow logic (invoking services, waiting for the responses, error handling, etc.). Furthermore, clients have to account for data flow operations and type incompatibilities between services, which further complexifies the task. Consequently, the quantity of code developers have to write in BPEL grows proportionally to the number of services they want to compose. The high complexity of the written code typically makes the use of BPEL and other conventional techniques not really suitable in practice, and associated visual edition tools unusable. Furthermore, existing workflow languages typically require strongly-typed and well-defined interfaces from composed services. However, defining such interfaces is not the trend anymore due to the fast proliferation of services that most often expose their web APIs without any contracts (such as with REST for instance) [Maximilien et al., 2007]. Thus, there is a need to write some glue code to compose services in an ad hoc and fast manner.

From another perspective, with the emergence of continuous service integration and development (commonly referred to as *DevOps*), workflow languages need to support not only static composition of well-specified services, but also on-the-fly integration of services that have not been previously planned at design time [Pautasso, 2009a; Pautasso and Alonso, 2005]. Doing so enables smoother and faster integrations of new services, while also providing better reliability at the orchestration level, should a required service fail to respond, as an equivalent service could be dynamically selected instead by the runtime system. However, conventional methods fail in this aspect, especially in the context of microservices and REST APIs. Finally, existing workflow languages are typically bundled with an execution engine such as an Enterprise Service Bus (ESB). However, ESBs are well known to be heavyweight execution platforms [Chappell, 2004]. This makes their deployment and administration more costly and time consuming, as they require a lot of (human and computational) resources to operate. As such, they do not meet the trend of lightweight containers and frequent deployments, as popularized by Docker. Docker enables developers to deploy their service compositions wherever they want, such as personal clouds, according to specific privacy requirements [Fuchs and Gürgens, 2013].

Hence, the SOA paradigm has to evolve. Well known service providers such as Netflix, Amazon, Spotify and SoundCloud have already widely adopted a refinement of the SOA paradigm named microservices. Microservices are no more than SOA instances constrained to the basics of HTTP, i.e. with a RESTful style, without the WS-* specifications, and coupled with a variety of tools to promote fast deployment and undeployment of services. However the challenge to compose services stays open to microservices practitioners that are free to use the programming language they want.

1.2.2 Detecting specific changes in web service data

Integration platforms such as IFTTT [6] and Zapier [7] have recently emerged with the aim of orchestrating interactions between a multitude of web services such as Facebook and Twitter [Liu et al., 2000; Pandey et al., 2004]. They enable end users to describe which actions to trigger when a custom event occurs on a web service [Ur et al., 2016]. For instance, one may want to automatically tweet a message when a specific subway line becomes unavailable. However, most of existing web services do not provide a way to specify custom event notifications. To overcome this limitation, platform owners have developed their own notification system by performing a recurrent polling of monitored services. For each service, the current state is periodically fetched and compared against the previous one to identify specific values that vary over time. When a change is detected, the corresponding event is raised. Because specific code needs to be developed for each event of a service, the set of supported services and events is limited and does not necessarily meet user expectations.

Each step of the monitoring process can be relatively complex. As an example, consider the use of the Facebook service to detect new photos with a given tagged user in a given album. To implement this scenario, one needs first to periodically poll several Facebook API endpoints (the one for the photos and the one for the tags) and navigate through the paginated responses. The resulting aggregated state is then compared against the previous one. However, this comparison requires focusing only on new photos (identified by their unique IDs) while ignoring other irrelevant changes such as the last update time. Even such a simple use case underlines the complexities of this process, which are declined in two different challenges: state computation and change detection.

Although the computation of a state sometimes requires fetching a unique resource from a single API endpoint, it is often necessary to implement more complicated policies. For instance, the construction of a state may require navigating through a set of API endpoints, where several requests must be chained in a particular order to correctly fetch the relevant data. In addition, responses returned by a service can be paginated and thus necessitate several subsequent requests to accumulate all the data. Thus, constructing a state can quickly become laborious.

Once a state has been computed, it is necessary to detect changes with the previous one. However, off-the-shelf techniques can produce unexpected or irrelevant results as in the previous Facebook example in which photos with only a modified last update time should not be reported as different. Developing a generic differencing tool is a well-known complex problem, and can be NP-hard depending both on the change operations that are considered, and on the guarantees about the output size [Buttler, 2004].

6. https://ifttt.com
7. https://zapier.com

1.2.3 Scaling a service composition platform

Designing a scalable service composition platform as envisioned by the CPRODIRECT company that would be capable of efficiently supporting hundreds of thousands of users is a non-trivial task, and highlights two key challenges.

First, the easiest way to make such a platform scale (i.e. without altering its existing software architecture) is to perform *vertical scaling*. However, scaling vertically requires increasing the capacity of the existing server, for instance by investing in more raw processing capacity, and/or more memory. Obviously such hardware updates are well known to be expensive and limited [Cáceres et al., 2010]. Furthermore, it does not provide any auto-provisioning nor auto-scaling features, which are required to scale smoothly according to the number of users and simultaneous executions of compositions [Vaquero et al., 2011]. Hence, the deployment of such a platform is not cloud-friendly: the billing is independent of the resources consumed, which has a direct consequence on operational costs.

Second, the proposed platform architecture needs to directly take into account API rate limit rules and quota policies of third-party services that are composed. Such rules or quotas are often applied to avoid inappropriate use of services, by limiting the number of requests a client can perform in a given timeframe. API rate limits also allow service providers to achieve better performance (especially during traffic peaks), better security (reduces impact of Denial-of-Service attacks), and enables them to provide higher rate limits as premium offerings. Without these restrictions, a set of clients issuing requests to the same service at the same time can severely degrade the experience for all the other clients. As a consequence, the composition platform can potentially be blocked or black-listed if the rate limits are exceeded. In addition, excessive invocation of services within a composition increases average execution time and thus resource usage. State of the art techniques are not straightforward to apply in the context of service composition to address both of these key issues.

1.3 MEDLEY: an event-driven lightweight platform for service composition

The contribution of this thesis aims at investigating the underlying challenges in web service composition in the context of modern web development practices. The ultimate goal of this thesis is to provide an architectural framework to support the specification and execution of web service compositions in a scalable manner. To this extent, we propose four complementary contributions.

First, we introduce ARIA, a domain-specific language for describing service compositions using high-level constructs and domain-specific semantics. ARIA is specifically designed to tackle the aforementioned problematic issues encountered when orchestrating the composition of various heterogeneous web services. By providing an abstraction layer

between the low-level implementation and the high-level business logic, the language allows users to express compositions with fine-grain tuning of both control flow and data flow. Additionally, ARIA meets the current trends in terms of continuous service integration and development to promote a continuously evolving service-oriented architecture.

Second, we introduce a declarative language-based approach, POLLY, to simplify change detector construction. POLLY enables describing change detection strategies in JSON data fetched from RESTful APIs. The domain-specific language provides declarative, simple yet highly-expressive constructs for describing how to construct a state from one or multiple API endpoints, how to identify changes in states, and how to produce a custom output. The POLLY compiler automatically produces an efficient JavaScript implementation which runs on top of a runtime system and hides low-level requirements such as HTTP authentication and pagination. In our context, POLLY change detectors enable generating custom events to automatically trigger the execution of ARIA compositions when a change occurs in targeted web service data.

Third, we present the architecture of MEDLEY, an event-driven lightweight platform for service composition. The MEDLEY platform comprises a runtime system to support the execution of service compositions specified using the ARIA language, and enables the fast integration of third-party service providers. Once defined, ARIA specifications are compiled into low-level code which runs on top of MEDLEY. The runtime system relies on an event-driven, process-based communication paradigm for a lightweight and highly perfomant execution model. MEDLEY also supports the integration of service provider components specified using POLLY, thus enabling triggers for the execution of ARIA compositions based on change events detected by POLLY change detectors.

Furthermore, to ensure the scalability of the MEDLEY platform in production environment, we focus on a novel approach for efficient scheduling in service orchestration engines. The main challenge is to support an increasing number of users while taking into account the API rate limits of third-party services used by the service compositions. To the best of our knowledge, this issue has not been addressed yet in the current state of the art. In particular, we design MEDLEY to support *horizontal scaling*. Scaling horizontally enables creating applications that scale across nodes. To this end, in a way similar to Docker Swarm [Merkel, 2014], we introduce a custom scheduler to the MEDLEY platform to be able to create a MEDLEY cluster capable of dynamically increasing or decreasing the number of MEDLEY nodes to distribute the incoming workload. However, in contrast to Docker Swarm which is agnostic to the containerized application, our own scheduler is able to dispatch composite services according to both their dependencies, and the resources that the composed services consume. As a consequence, the MEDLEY platform can be easily deployed on public cloud infrastructures, thus enabling the billing of only the resources that are effectively consumed. Furthermore, to overcome API rate limit rules of third-party services, the MEDLEY platform is enhanced with caching capabilities on each node of the cluster. The MEDLEY scheduler relies on a heuristic-based approach to optimize cache

affinity, thus reducing the total number of requests to third-party services, and improving the scalability of the platform.

1.4 Thesis outline

The remainder of this document is organized as follows. Chapter 2 presents an overview of the state of the art in the field of web services and service orchestration. We describe their fundamental concepts while highlighting their shortcomings in our context. In Chapter 3, we propose two domain-specific languages (DSLs). First, we present POLLY, a DSL for detecting custom changes in web service data. We explain the DSLs semantics, operators and grammars while illustrating their usefulness through relevant scenarios (Section 3.2). Second, we present ARIA, a DSL for specifying service compositions using high-level constructs and domain-specific semantics (Section 3.3). Then, we introduce MEDLEY in Chapter 4. MEDLEY is an event-driven lightweight platform for service composition. We show how MEDLEY supports the execution of ARIA compositions, which can be triggered by change events detected by POLLY change detectors. In Chapter 5, we present a thorough evaluation of our contributions. We evaluate the expressivity and features of the proposed DSLs, and undertake a performance evaluation of the MEDLEY platform, then discuss the results. Finally, Chapter 6 concludes this document by summarizing our contributions and exposing several perspectives of this work.

CHAPTER

2 Background

In this chapter, we present fundamental concepts related to web service composition. We give a brief overview of service-oriented architectures covered in this thesis, describing the transition from legacy architectural styles, to the more modern microservices architectures. We then present several existing languages, models and platforms used for service composition. Finally, we introduce some notions in change detection in the context of web services data.

Contents

2.1 Genesis of service-oriented architectures

In a broad sense, the concept of architecture is what allows systems to evolve and provide a certain level of service throughout their lifecycle. In the context of software engineering, this translates into high-level concerns for bridging the gap between system functionality and target requirements that the system has to meet. Over the past several decades, software architectures have been thoroughly studied, constantly evolving and adapting according to the latest technological advances and trends.

Ever since the 1970s, developers experienced problems associated with large-scale software development [Brooks, 1975]. As such, the following decades witnessed a huge rise of interest from the research community for software design and its implications on the development process. References to the concept of software architecture also started to appear around the 1980s [Bergland, 1981]. However, a solid foundation on the topic was only established in 1992 in a publication authored by Perry and Wolf [Perry and Wolf, 1992]. They define software architecture distinctly from software design. Evern since, their work has generated an evolving community of researchers that actively studied the notion and the practical applications of software architecture. In the years to follow, software architecture concepts were broadly adopted by both industry and academia. Bosch's work [Bosch, 2004] provides a good overview of the current research state in software engineering and architecture, highlighting the challenges to investigate. Since its appearance, software architecture has developed into a mature discipline making use of notations, tools, and several techniques.

As a result, software engineers have come up with different ways to design, implement and compose systems that provide broad functionality and satisfy a wide range of requirements. In the remainder of this section, we provide a brief overview on the evolution of software architectures.

2.1.1 Monolithic applications

In essence, IT businesses face the challenges of minimizing costs while also meeting the growing need for evolution. Driven by ever-changing user requirements and competitive offerings, they have to deliver better services and user experiences in a shorter amount of time. However, legacy applications typically tend to be built as monolithic applications, leading to higher costs of maintenance and evolution.

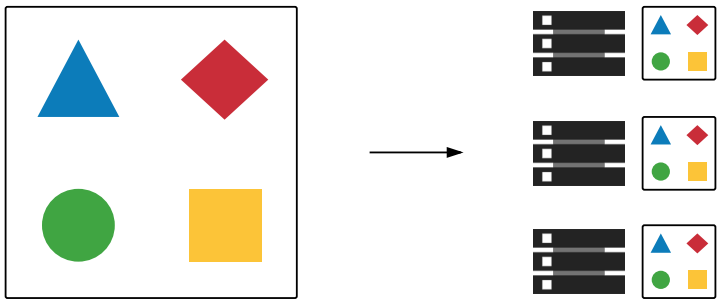


Figure 2.1 – *"A monolithic application puts all its functionality into a single process, and scales by replicating the monolith on multiple servers"* [Fowler and Lewis, 2014].

> **Monolithic applications.** A monolithic application is built as a single-tiered unit. It is typically characterized by a set of distinguishable functionalities (data processing, persistence, error handling, user interface, etc.) which are all interwoven, forming a single logical executable. Any change to the system involves rebuilding and deploying a new version of the application. Likewise, scaling requires scaling the entire application rather than parts of it that require greater resources (see Fig. 2.1).

As applications grow, large monoliths tend to become difficult to maintain and evolve, due to their increased complexity. Developers must coordinate their development and deployment efforts due to the lack of clear boundaries between the constituents of the application. Common tasks like contributing code and tracking down bugs require long perusals throughout the code base, leading to decreased developer productivity. As a monolithic application is deployed as a single executable artefact, any change in a component of the monolith requires rebooting the whole application. For large-sized projects, restarting usually entails considerable downtimes, hindering development, testing, and the maintenance of the project. Thus, it becomes difficult to apply continuous development practices which typically promote frequent updates. Furthermore, as monoliths are composed of several different components, each component may have different resource requirements at runtime [Balalaie et al., 2015]. As such, deploying a monolithic application is usually sub-optimal with regards to the required resources: some components can be memory-intensive, others computational-intensive, etc. When choosing a deployment environment, developers must compromise with a one-size-fits-all configuration, which is either expensive or sub-optimal with respect to the individual components. To scale a monolithic application, developers can either deploy the monolith on a more powerful host (*vertical scaling*), or replicate the entire monolith on several machines and distribute the load among them (*horizontal scaling*). Either way, only a subset of the components is stressed, leading to inefficient and costly scaling, as each component cannot be scaled independently in monolithic applications.

2.1.2 Service-oriented architectures

Ever since, a particular focus has been given to the fundamental principle of separation of concerns (SoC) [Hürsch and Lopes, 1995]. SoC is a design principle that dictates the separation of a program into distinct sections, such that each section addresses a separate, well-defined concern. This allows better control over design, implementation and evolution of software systems. In this sense, software architectures gradually evolved from monolithic applications, to a more loosely coupled set of web services. Web services are the building blocks of service-oriented architectures (SOA).

> **Web services.** According to the W3C Web Services Glossary, a web service is a "software system designed to support interoperable machine-to-machine interaction over a network". In other words, a web service is a self-contained application that can be invoked over the network to perform a given operation, and relies on open standards for communication and messaging.

With the establishment of web services, developers could harness the complexity of distributed systems and to integrate different software applications [MacKenzie et al., 2006]. Service-oriented architectures (SOA) rely on a set of guidelines and protocols for defining web services. These include encapsulation, interchangeability, abstraction and business cohesion [Wang and Fung, 2004]. Typically, a web service exposes its functionalities to other components via a well-defined interface. It relies on standard protocols for message passing. As such, this enables modularity and reuse of services across different systems, as well as implementation independence. Furthermore, as SOA relies on the principle of separation of concerns, it enables the implementation of an application as a set of distinct services, developed by dedicated teams.

2.1.3 Microservices

As the sheer scale of applications increases (in terms of data consumption, processing and output), it becomes increasingly important to find fault tolerant, scalable ways to manage both systems and the data they manage. Further refining the SOA paradigm is the microservice architectural style. This paradigm is a more modern interpretation of service-oriented architectures used to build distributed software systems [Namiot and Sneps-Sneppe, 2014].

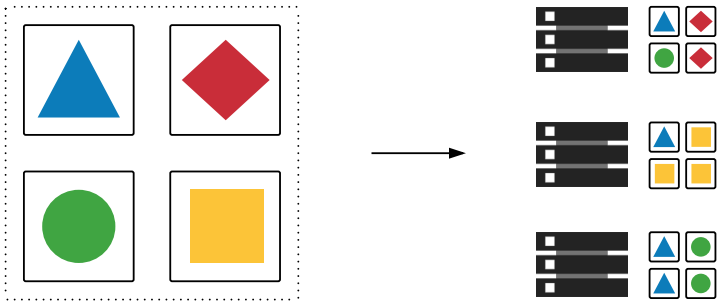


Figure 2.2 – *"A microservices architecture puts each element of functionality into a separate service, and scales by distributing these services across servers, replicating as needed"* [Fowler and Lewis, 2014].

> **Microservices.** The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often using an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery (see Fig. 2.2).

First introduced in 2011, the term *microservices* was introduced as a way to describe a new paradigm for programming applications using a modern, flexible architectural style to meet the demands of the fast-paced web. This new trend in software architecture emphasizes the design and development of highly maintainable and scalable software [Dragoni et al., 2016]. Although the microservices architecture gained popularity relatively recently, it has already been the subject of numerous studies to discuss patterns and applications of microservices [Krause, 2014].

This architectural style allows managing growing complexity by functionally decomposing large systems into a set of independent services. From a technical point of view, microservices are self-contained components that are independently developed and tested, and conceptually deployed in isolation and equipped with their own data persistence solutions. The distinguishing behaviour of a microservice architecture derives from the composition and coordination of microservices, each running its own processes and communicating via lightweight mechanisms. This approach delivers all sorts of benefits in terms of maintainability and scalability. As microservices are implemented independently from each other, their code base tends to be inherently smaller. As such, developers can more easily develop, test and investigate the behaviour of a functionality independently from the rest of the system. Furthermore, it becomes possible to seamlessly update an application by deploying new versions of a microservice and gradually transitioning the incoming traffic from the old version to the new version of the microservice. Instead of rebooting the whole system, individual microservices can be updated at different rates, as required. Unlike monolithic applications, a microservices architecture enables the system to conve-

Figure 2.3 – Overview of a continuous deployment pipeline, defining all stages from development to release. Tests range from unit tests, to integration and acceptance tests. Existing tools for continuous integration, development and delivery include Jenkins [1], Travis CI [2], GitLab CI [3], Circle CI [4], and Codeship [5].

niently scale up or down each individual microservice independently from the other services that constitute the application, according to its load [Gabbrielli et al., 2016].

These characteristics foster continuous integration [Fowler and Foemmel, 2006] and greatly ease the maintenance of the application, while also promoting faster and more frequent update cycles. To facilitate working with such distributed systems, several automation tools and techniques emerged, with the aim of accelerating the development, deployment and maintenance of microservices. This trend, commonly known as *DevOps* [Balalaie et al., 2016], relies on the use of container-based solutions (such as Docker [Merkel, 2014]) to promote faster test and build cycles, while also streamlining automated deployments through continuous integration practices [Smeds et al., 2015]. Such containerisation enables developers to enjoy a high degree of freedom in the configuration of the deployment environment that best suits their needs (in terms of costs and quality of service). Figure 2.3 illustrates an example of a *DevOps* pipeline for continuous deployment.

2.2 Overview of the Web Service stack

Founded in 1994 by Tim Berners-Lee, the World Wide Web Consortium (W3C) is responsible for developing and maintaining protocols and standards to ensure long-term growth for the Web. With over thousands of drafts and specifications, W3C is the leading reference in the web community. Among the proposed standards, the Web Services (WS-*) specification suite [Weerawarana et al., 2005] proposes a technological stack aimed at standardizing how web services are defined, described, published, located and invoked. Figure 2.4 presents a quick overview of some of the WS standards that are relevant to our work.

5. https://jenkins.io
5. https://travis-ci.org
5. https://docs.gitlab.com/ce/ci
5. https://circleci.com
5. https://codeship.com

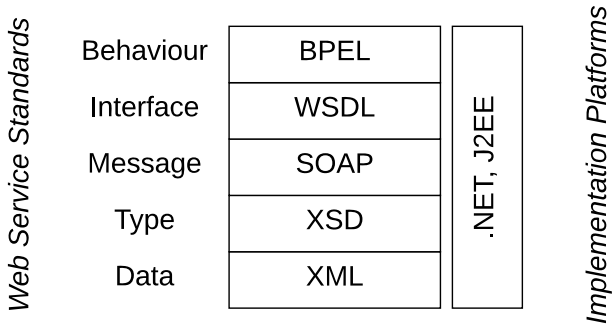


Figure 2.4 – Web Services technological stack.

2.2.1 Service invocation with SOAP

Designed in 1998 as part of the WS-* stack, SOAP (Simple Object Access Protocol) is a protocol specification for implementing web services [Box et al., 2000]. It relies on the transmission of SOAP messages for messaging and remote procedure calls (RPC), and leverages existing transport protocols such as HTTP and SMTP for communication instead of defining its own protocol. Exchanged SOAP messages are wrapped in SOAP envelopes, structured as XML documents, as shown in Fig. 2.5. The SOAP envelope identifies the XML document as a SOAP message, and contains a header and a body. The header contains optional metadata information (e.g. authentication, routing details, delivery settings), while the body contains the payload of the message destined to be processed by the receiver.

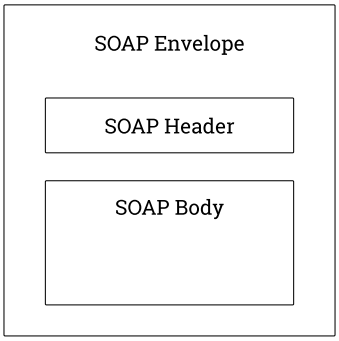


Figure 2.5 – Structure of a SOAP message.

2.2.2 Service description with WSDL

SOAP services are described using the WSDL (Web Service Description Language) standard [Christensen et al., 2001]. WSDL defines the service interface description in a standard, implementation-independent way. It provides details about how to locate the ser-

vice, and describes the set of supported operations. Operation signatures are described using the XSD (XML Schema Definition) standard [Gao et al., 2009]. XSD is an XML-based meta-language for formalizing the structure of XML documents and specifying data structures exchanged between services in the context of SOA.

2.2.3 Service discovery with UDDI

To support the discovery of existing web services, UDDI (Universal Description Discovery and Integration) registries enable publishing web services interfaces, making them available for external clients [OASIS, 2004]. By offering users a unified and systematic way to find service providers through a centralized registry of services, UDDI registries can be queried to locate web services based on their characteristics. Similar to a phone directory, UDDI registries encode infomation about web services under three categories: (i) *white pages* include name and contact details, (ii) *yellow pages* include a categorization based on business and service types, and (iii) *green pages* include technical details about the service.

2.2.4 The Web Service model: putting things together

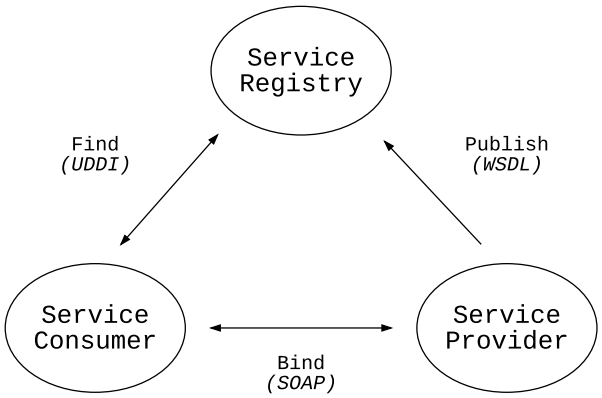


Figure 2.6 – Architecture of the WS model.

Figure 2.6 gives an overview of the elements introduced previously, and shows how they come to play together. The service provider implements the web service and describes its interface using WSDL. To make the service discoverable, it is published in a central service registry using UDDI. The service registry indexes published services, enabling clients to easily locate the services. Finally, the service consumer queries the registry to lookup an existing service, and uses the WSDL service description obtained to bind to and invoke the web service.

2.3 The REST architectural style

Initially presented in the doctoral dissertation of Roy T. Fielding [Fielding, 2000], the REST (Representational State Transfer) architectural style is by far the most widely adopted way of exposing services over the web today [Danielsen and Jeffrey, 2013]. It revolves around the central notion of resources, which are abstract entities identified by URIs, and manipulated through a uniform interface. REST relies on HTTP as the underlying transport protocol, enabling clients to manipulate resources using the standard HTTP verbs. For instance, the `GET`, `POST`, `PUT` and `DELETE` verbs are typically used to read, create, update and delete resources, respectively.

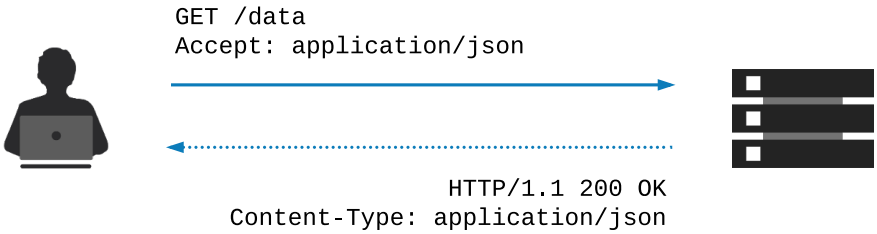


Figure 2.7 – A request/response roundtrip between a client and a REST web API. The `Accept` header allows specifying the resource representation that the client wishes to receive, while the `Content-Type` header specifies which representation is returned in the response.

HTTP status codes. REST also relies on the standard HTTP response status codes to provide a uniform interface for specifying the semantics of the response, allowing clients to react accordingly [Fielding et al., 1999]. For instance, status codes in the 2xx range convey information about successful operations, 3xx about redirections, 4xx about client errors, and 5xx about server errors (see Fig. 2.7).

Resource representation. Resource states are commonly represented using the JSON (JavaScript Object Notation) format, although any other standard or arbitrary media types can be used (such as XML). The client specifies during content negotiation which content representations it can process, and the server either supplies one of the requested representation if possible, or an `406 Not Acceptable` error if it cannot automatically make a selection (see Fig. 2.7). This fosters reusability, interoperability and loose-coupling.

JavaScript Object Notation. Although REST does not enforce any particular resource representation format, the most commonly used format is JSON [Rodríguez et al., 2016], due to its simplicity, ease of use and smaller footprint compared to other alternatives. A

JSON document is a textual serialization of structured data, derived from the object literals of JavaScript. It consists of a tree composed of three kinds of nodes: literals, arrays and objects. A literal node can be one of the following primitive types: a number, a string, a boolean or the `null` literal. An object node is an unordered collection of zero or more key/value pairs, where each key is a string that is unique within the object, and a value that is a node. An array node is an ordered sequence of zero or more nodes. Figure 2.8 shows an example of a JSON document.

```
{
  "id": 123,
  "details": {
    "title": "Summer 2017",
    "subtitle": null,
    "tags": ["vacation", "beach", "sun"]
  },
  "private": false
}
```

Figure 2.8 – Example of a JSON document describing a photo album.

2.3.1 Hypermedia-driven discovery with HATEOAS

The REST architectural style supports the dynamic discovery of an application's capabilities entirely through hypermedia. HATEOAS (Hypermedia As The Engine Of Application State) is a REST constraint that enables the client to navigate the REST API interface dynamically by including hypermedia links with the server responses. The media types used for the resource representations and the link relations they may contain are standardized. The client navigates through application states by following the links included within a representation or by manipulating the representation in other ways afforded by its media type. This capability differs from that of SOA-based systems and WSDL-driven interfaces, where endpoints are statically fixed [Alarcon et al., 2010]. As an illustration, Fig. 2.9 shows an example of a HATEOAS-based API response that provides the user's name, while including a self-linking URL where that user is located, and how to locate that user's albums. The `rel` attribute defines the relationship of the link with regards to the resource itself.

2.3.2 REST API description methods

To enable the description of REST services, the XML-based language WADL (Web Application Description Language) standard was proposed as part of the W3C as a simpler alternative to WSDL (which was initially designed for SOAP services) [Hadley, 2006]. A WADL document describes the set of resources that can be manipulated using the REST service, giving details about the access method (HTTP verbs) and XSD descriptions of the resource

```
{
  "name": "Alice",
  "links": [
    {
      "rel": "self",
      "href": "https://api.example.com/users/1"
    },
    {
      "rel": "albums",
      "href": "https://api.example.com/users/1/albums"
    }
  ]
}
```

Figure 2.9 – Example of a HATEOAS-based response containing hypermedia links.

types. Alternatively, other JSON-based API specification languages have been developed by the industry, such as RAML [6], Blueprint [7] and the proprietary Google API Discovery Service [8]. More recently, a consortium of several major API vendors came together to found the OpenAPI Initiative [9] (founded in November 2015), in an effort to standardize how REST web APIs are described. OpenAPI relies on the JSON Schema standard [Galiegue et al., 2013; Pezoa et al., 2016] to provide a machine-readable API definition, making possible use-cases such as interactive documentation, client-side and server-side code generation, and automation of test cases. Although OpenAPI is gaining more and more traction (with over 350,000 downloads per month), it is still far from being widely adopted by the majority of the web API community [Fokaefs et al., 2015; Lucky et al., 2016]. Instead, serivce providers tend to simply provide plain human-readable HTML descriptions of the documentation. However, web APIs clients have no control over the API and the service behind the API, as a provider may change either or both, potentially causing breaking changes.

2.4 Service composition overview

Due to the considerable cost decrease in cloud computing, the past decade witnessed the emergence of a fairly large number of web services. Inherently, this enables clients to rely on a set of existing services in order to develop new ones. However, this comes with its own set of challenges. A number of languages have been proposed to define how services can be composed into business processes [Sheng et al., 2014].

In the service-oriented paradigm, the essential idea lies not only in the reusability of coarse-grained *business* functionalities exposed as services, but more importantly in

6. https://raml.org
7. https://apiblueprint.org
8. https://developers.google.com/discovery
9. https://www.openapis.org

the definition of loosely-coupled entities, specified in business terms instead of technical ones [Papazoglou, 2003]. At the service level, a composition refers to a business behavior, which assembles (beyond a single program and language) invocations of several services to perform a given task.

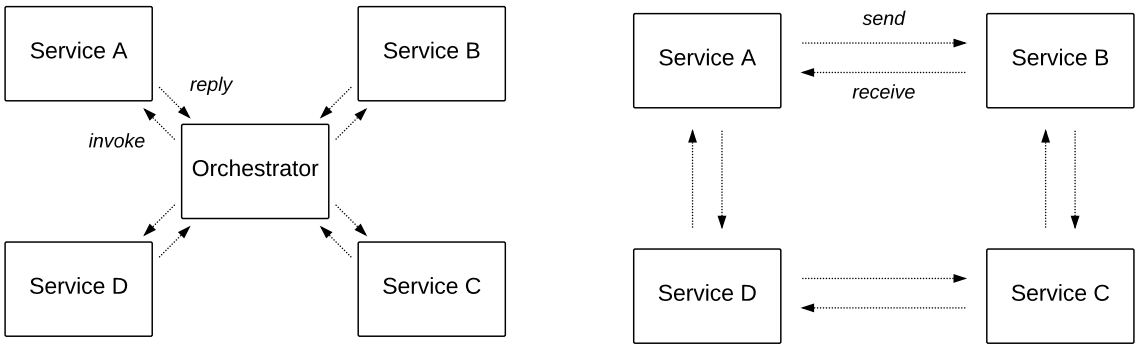
> *"Services reflect a "service-oriented" approach to programming, based on the idea of describing available computational resources, e.g., application programs and information system components, as services that can be delivered through a standard and well-defined interface. [...] Service-based applications can be developed by discovering, invoking, and composing network-available services rather than building new applications."* [Papazoglou, 2008].

Although existing technologies (such as the WS-* stack) provide means to describe, locate, and invoke services over the network, they fail in giving a rich behavioral description about the role of the service in a broader, more complex collaboration. Such a collaboration consists in a sequence of activities and relationships between activities, which constitutes the logic of a business process. In this sense, service composition consists in creating higher level, cross-organizational business processes from a set of existing web services, focusing on the composition business logic rather than the technical details. Service composition can be achieved using one of two paradigms: service orchestration and service choreography [Peltz, 2003].

> **Orchestration.** Service orchestration represents a single centralized executable business process (the orchestrator) which coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services. The relationship between all the participating services are described by a single endpoint (i.e. the composite service). The orchestration includes the management of transactions between individual services. Orchestration employs a centralized approach for service composition.

> **Choreography.** Service choreography is a global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints. It allows each involved party to describe its part in the interaction. Choreography employs a decentralized, collaborative approach for service composition.

In other words, a choreography describes the collaborative interactions between multiple services, whereas orchestration represents a centralized control from one party's perspective. This means that a choreography differs from an orchestration with respect to



(a) Service orchestration.

(b) Service choreography.

Figure 2.10 – Service orchestration *vs.* service choreography.

where the logic that controls the interactions between the services involved should reside. Figure 2.10 illustrates these approaches at a higher level.

In the remainder of this dissertation, we employ the term *"composition"* to denote the composition of service invocations to perform a business-oriented task.

Composition models

Composition models define abstractions and languages to specify the order in which and the conditions under which web services are invoked [Dustdar and Schreiner, 2005]. They rely on process-modeling languages, such as UML activity diagrams, Petri-nets, state-charts, rule-based orchestrations, activity hierarchies, and π-calculus. Data access models define how data is specified and exchanged between parties. The service selection model deals with static and dynamic binding to specify how a web service is selected as a component (either statically at design-time, or dynamically during runtime). Transactions define which transactional semantics can be associated to the composition and how this is done. Finally, a model for exception handling is required to handle exceptional states during the execution of the composite service without aborting the composition. Other composition approaches introduce the notion of automated [Narayanan and McIlraith, 2002], ontology-based [Agarwal et al., 2003] and semantic web services composition [Rao and Su, 2003], as alternatives to manual composition techniques. The semantic web community provides interesting approaches to support the adaptation of business processes based on semantic descriptions [Küster and König-Ries, 2006].

2.4.1 Process algebras & concurrency models

Various formalisms were proposed in the first half of the 20th century to formalize the concept of the behaviour of a system, leading to the foundation of process algebras [Morimoto, 2008; Aceto and Gordon, 2008]. Process algebras are a diverse family of abstract

languages used to formally specify the execution model of concurrent systems. Such languages provide the necessary semantics to express interactions, communications and synchronizations between several independent processes [Magee et al., 1999; Ferrara, 2004; Foster et al., 2005]. These formalisms are founded on algebraic laws and support the automatic verification of properties of systems behavior. They enable one to reason formally on systems and apply various model-checking techniques to verify properties, variants and invariants of concurrent systems [Baeten, 2005]. Throughout its execution lifecycle, a system may interact with one or several other systems. To describe parallel or distributed systems, a process algebra relies on a set of structural laws, i.e. a given set of atomic actions, and basic operators to compose these into more complicated processes. Typically, basic operators include parallel composition, alternative composition, and sequential composition.

There is a considerable amount of work and applications realized in a number of process algebras. Among the multitude of proposed algebras, CCS (Calculus of Communicating Systems) [Milner, 1989] was historically the first with a complete theory, introducing the semantics of algebraic operators. On the other hand, CSP (Calculus of Sequential Processes) [Hoare, 1978] adopts the message passing paradigm of communication, using synchronous communication and is a guarded command language. Later on, it was found that this model was lacking, for instance because deadlock behaviour is not preserved. Further contributions lead to the specification of ACP (Algebra of Communicating Processes) [Bergstra and Klop, 1985], which emphasizes the algebraic aspect, using an equational theory with a range of semantic models and a more general communication scheme. Finally, LOTOS (Language of Temporal Ordered Systems) [Bolognesi and Brinksma, 1987] is a formal specification language based on temporal ordering of events, used for protocol specification. It provides means to describe data and operations based on abstract data types, while also enabling the description of concurrent processes based on process calculus. Other formal languages such as Petri nets can be used for model-checking of existing orchestrations [Murata, 1989], while the more expressive π-calculus [Milner, 1999] offers constructs to compose business processes in terms of sequential, parallel and conditional executions, leading to compositions of arbitrary complexity.

2.4.2 BPEL: Business Process Execution Language

With the rapid expansion of service-oriented architectures, the need for a workflow modeling framework became clearer, leading to the development of BPEL (Business Process Execution Language) [Andrews et al., 2003]. Standardized by the OASIS organization in 2004, BPEL is made part of the standard Web Service stack (under the name WS-BPEL). It consists in an XML-based language defining several constructs to describe business processes across a set of web services. It defines a set of basic control structures such as conditions, loops, and elements to invoke web services and receive messages from them. The language also provides a model for describing the behavior of a composition based on its

interactions with the composed services. Message structures can be manipulated, assigning parts or the whole of them to variables that can in turn be used to send other messages. BPEL relies heavily on WSDL interfaces [Christensen et al., 2001] to define links with partner services and uses an XML-based data model.

BPEL supports two different types of business processes: executable and abstract business processes. On one hand, executable processes model the actual behavior of a participant in a business interaction. They follow the orchestration paradigm and can be executed by an orchestration engine. On the other hand, abstract processes are partially specified processes. They hide some of the internal behaviour details, and serve a descriptive role, as they are not intended to be executed.

Core language constructs. A BPEL orchestration can be represented as a series of steps, where each step is called an activity. BPEL supports two types of activities: primitive and structure activities. On one hand, primitive activities enable users to perform common tasks, such as invoking web services (`<invoke>`), waiting for the response (`<receive>`), manipulating data variables (`<assign>`), and throwing runtime exceptions (`<throw>`). On the other hand, structure activities enable users to combine primitive activities to express a more complex logic. For instance, users can define a set of activities that will be invoked in an ordered sequence (resp. in parallel) using the `<sequence>` (resp. `<flow>`) construct. For control flow semantics, the `<while>` construct can be used to define loops, whereas `<switch>` can be used to implement switch-case branches.

Runtime environment. The execution of BPEL orchestrations requires deploying them on a BPEL-capable server. BPEL servers typically provide control over process instances that are executing and those that have finished, while also supporting long-running processes and managing intermediate process states [Louridas, 2008]. Some servers even provide control over process activities and allow their monitoring. Deploying a BPEL process requires a deployment descriptor (which is not covered by the BPEL standard) and is specific to each BPEL server. The deployment descriptor typically specifies the BPEL source file name, process name, WSDL locations of all partner link services, and other configuration properties. Some of the most popular BPEL servers are based on Java EE, and include Oracle BPEL Process Manager, IBM WebSphere Business Integration Server Foundation, BEA WebLogic Integration, ActiveBPEL Engine and Apache ODE.

BPEL extensions. In the following years, a number of contributions proposed several extensions and refinements of BPEL. Among these solutions, some tackle service composition using a goal-driven semantic approach [Klusch and Gerber, 2006; Zhao and Doshi, 2009; Mayer et al., 2014]. They rely on ontologies and on reasoning engines to dynamically select services that fulfill the user-provided requirements. The scientific workflow com-

munity also uses BPEL processes to enact workflows on computing grids [Emmerich et al., 2005].

Example

To illustrate the concepts presented earlier, we propose the example presented in Fig. 2.11. The aim is to help users in planning travel plans, by looking up several airlines (here, `Airline1` and `Airline2`) and identifying the one that offers the lowest prices for the given travel details (destination, dates, etc.). We now describe the necessary steps to perform in order to implement this scenario using BPEL. To simplify our example, we assume that both airlines offer a web service and that both services are identical (i.e. provide same port types and operations). We also forgo implementing any fault handling, which remains a crucial aspect in real-world scenarios.

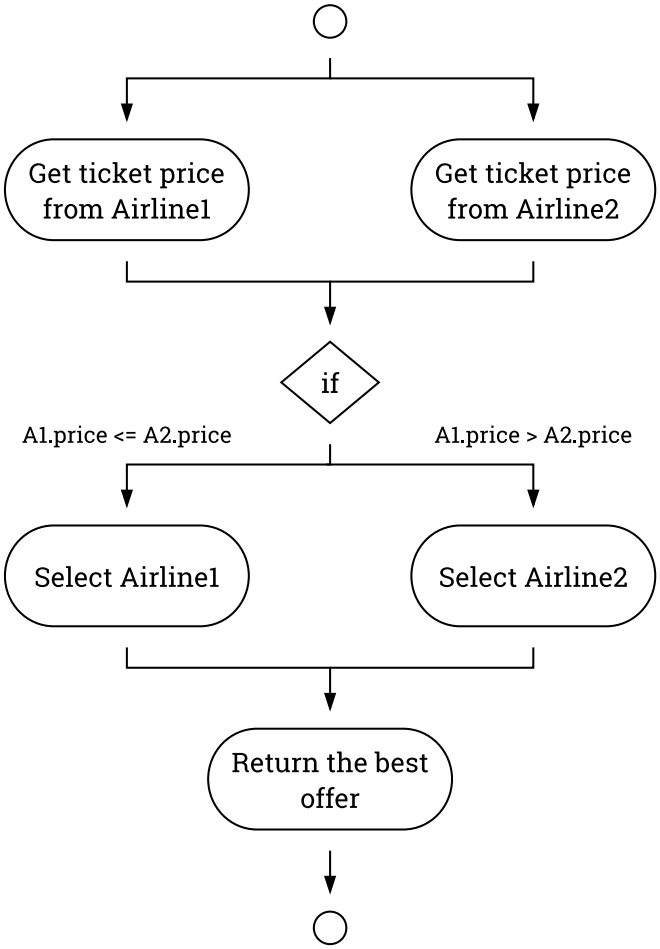


Figure 2.11 – Overview of a BPEL orchestration. It consists in looking up airline offers for the given travel details, and selecting the airline with the lowest price.

Process definition. First, we define the process by specifying its name (Fig. 2.12, line 2) and the required namespaces (lines 3-6). We define here the target namespace (line 3) and the namespaces to access the BPEL process WSDL (line 5) and the airline WSDLs (line 6). We also declare the default namespace for all BPEL activity tags (line 4).

```
1 <process
2   name="TravelProcess"
3   targetNamespace="http://example.com/bpel/travel/"
4   xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
5   xmlns:trv="http://example.com/bpel/travel/"
6   xmlns:air="http://example.com/service/airline/">
7   <!-- ... -->
8 </process>
```

Figure 2.12 – Definition of the BPEL process and its required namespaces.

Partner links. Partner links represent the interaction between the BPEL process and the involved parties. This includes all web services that will be invoked and the client of the BPEL process. Each partner link specifies up to two attributes: `myRole` indicates the role of the business process itself, while `partnerRole` indicates the role of the partner. In our example, the first partner link (Fig. 2.13, line 2) corresponds to the client that invokes the business process. The last two partner links (lines 3 and 4) correspond to the airline web services.

```
1 <partnerLinks>
2   <partnerLink name="client" partnerLinkType="trv:travelLT" myRole="travelService"
        ↪ partnerRole="travelServiceCustomer"/>
3   <partnerLink name="Airline1" partnerLinkType="air:flightLT" myRole="airlineCustomer"
        ↪ partnerRole="airlineService"/>
4   <partnerLink name="Airline2" partnerLinkType="air:flightLT" myRole="airlineCustomer"
        ↪ partnerRole="airlineService"/>
5 </partnerLinks>
```

Figure 2.13 – Definition of the process partner links.

Variables. Variables in BPEL processes enable storing, reformating, and transforming messages. A variable is typically needed for every message sent to partner services and received from them. For each variable, the type has to be specified. These types include the WSDL message type, XML Schema simple type, or an XML Schema element. In our example, we use WSDL message types for all variables (see Fig. 2.14).

```
1  <variables>
2    <!-- Input of this BPEL process -->
3    <variable name="TravelRequest" messageType="trv:TravelRequestMessage"/>
4    <!-- Input for Airline1 and Airline2 web services -->
5    <variable name="FlightDetails" messageType="air:FlightTicketRequestMessage"/>
6    <!-- Output from Airline1 -->
7    <variable name="FlightResponseA1" messageType="air:TravelResponseMessage"/>
8    <!-- Output from Airline2 -->
9    <variable name="FlightResponseA2" messageType="air:TravelResponseMessage"/>
10   <!-- Output from BPEL process -->
11   <variable name="TravelResponse" messageType="air:TravelResponseMessage"/>
12 </variables>
```

Figure 2.14 – Definition of the process variables.

Body. The process main body specifies the order in which activities are invoked. A `<sequence>` activity (Fig. 2.15, line 1) allows defining several activities that will be performed sequentially. Within the sequence, we first prepare the required input by copying the flight details from the `TravelRequest` variable (line 4) to the `FlightDetails` variable (line 5).

```
1  <sequence>
2    <assign>
3      <copy>
4        <from variable="TravelRequest" part="flightData"/>
5        <to variable="FlightDetails" part="flightData"/>
6      </copy>
7    </assign>
8    <!-- ... -->
```

Figure 2.15 – Variable assignment using the `copy` construct.

Invoking the airline services. Next, we invoke both airline web services to check for ticket prices. We use the `<flow>` activity (Fig. 2.16, line 1) to invoke both services asynchronously. For each service, we use a `<sequence>` (lines 2 and 6) to group an `<invoke>` activity (line 3) for the asynchronous invocation, and a `<receive>` activity (line 4) to wait for the callback. The resulting messages are stored in the `FlightResponseA1` and `FlightResponseA2` variables, respectively.

Selecting the cheapest offer. At this stage of the process, we have obtained two ticket offers from the invoked web services. We now use the `<switch>` activity to select the service offering the lowest price (Fig. 2.17). In lines 3 and 4, we use the BPEL function `getVariableData` to extract the value from the response message. We specify an XPath [Clark et al., 1999] query expression to locate the price element within the message part. Lines 6 to 11 assign the selected value to the output variable `TravelResponse`.

```
<flow>
  <sequence>
    <invoke partnerLink="Airline1" portType="air:FlightAvailabilityPT"
        ↪ operation="FlightAvailability" inputVariable="FlightDetails"/>
    <receive partnerLink="Airline1" portType="air:FlightCallbackPT"
        ↪ operation="FlightTicketCallback" variable="FlightResponseA1"/>
  </sequence>
  <sequence>
    <!-- Same goes for Airline2 -->
  </sequence>
</flow>
```

Figure 2.16 – Asynchronous invocation of the flight web services.

```
<switch>
  <case condition="
    bpws:getVariableData('FlightResponseA1', 'responseData', '/data/price') <=
    bpws:getVariableData('FlightResponseA2', 'responseData', '/data/price')">
    <!-- Select Airline1 -->
    <assign>
      <copy>
        <from variable="FlightResponseA1" />
        <to variable="TravelResponse" />
      </copy>
    </assign>
  </case>
  <otherwise>
    <!-- Select Airline2 -->
  </otherwise>
</switch>
```

Figure 2.17 – Selecting the cheapest airline offer.

Returning the final result. The final step of this BPEL process consists in returning a reply to the client indicating the selected airline. We use the client partner link to trigger the callback by invoking the `ClientCallback` operation on the `ClientCallbackPT` port type (Fig. 2.18, line 1). The `TravelResponse` variable holds the reply message.

```
    <invoke partnerLink="client" portType="trv:ClientCallbackPT"
        ↪ operation="ClientCallback" inputVariable="TravelResponse"/>
  </sequence>
</process>
```

Figure 2.18 – Notifying the user about the selected airline by invoking the corresponding client callback.

2.4.3 Orchestrating REST services

Nowadays, legacy web services are rapidly decaying, in favor of the more flexible REST architectural style. Although REST became the building block for major service providers, it still lacks an official standard for describing service interfaces, thus limiting the applicability of existing orchestration techniques in practice. Therefore, there is a fundamental mismatch between the REST architectural style and SOA orchestration solutions, since these solutions are not directly applicable [Zur Muehlen et al., 2005].

Nonetheless, several efforts have been made to support the composition of RESTful services [Haupt et al., 2014]. Some approaches such as Bite [Curbera et al., 2007; Rosenberg et al., 2008] and S [Bonetta et al., 2012] define domain-specific languages to express compositions. Bite follows a workflow model while S is an extension of JavaScript. Both of them require services to be statically binded and provide limited support for error handling. Other approaches propose to extend BPEL by adding new activities to manipulate REST resources as first-class entities [Pautasso, 2008, 2009b]. However, in practice, existing BPEL orchestration engines have limited support for composing REST services.

2.4.4 Commercial integration platforms

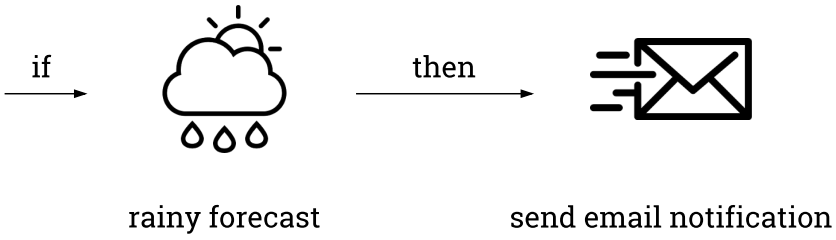


Figure 2.19 – An example of a rule-based composition, notifying the user by email if rainy weather is predicted.

In the commercial world, several SaaS (Software-as-a-Service) solutions and integration platforms have been built around the concept of composing these emerging services. These rule-based platforms provide user-friendly web applications in which users can describe simple orchestration scenarios between a multitude of web services such as Facebook and Twitter [Liu et al., 2000; Pandey et al., 2004]. For instance, they enable users to define a composition that automatically notifies the user by email if the forecast predicts rainy weather (Fig. 2.19). We provide here a quick overview of these commercial integration platforms.

IFTTT

Launched in 2011, IFTTT [10] (If This Then That) is a free service that allows end-users to describe simple compositions between a large number of web applications, with a strong emphasis on IoT devices and smart home automation [Ovadia, 2014]. Using a trigger/action paradigm, users can describe which actions to trigger when a custom event occurs on a given web service [Ur et al., 2016]. In other terms, an IFTTT composition is expressed as a pair of <*trigger, action*>, such as *"on <trigger> do <action>"*. IFTTT also provides a mobile application that allows users to view and manage their compositions, but also to leverage the device sensors as data sources for triggers (e.g. battery level, geolocation) and actions (e.g. ringer, SMS).

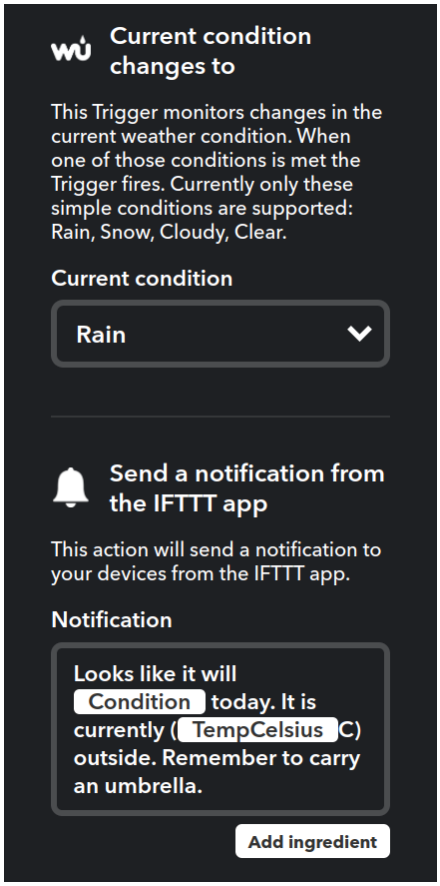


Figure 2.20 – A screenshot of an IFTTT composition. It consists in notifying the user with a custom message through the IFTTT mobile application about the weather predictions for the current day. Weather data is retrieved from the Wunderground [11] API.

10. https://ifttt.com
11. https://www.wunderground.com

Zapier

Initially released in August 2012, Zapier [12] is a web-based service that allows end users to integrate the web applications they use. If offers a wide range of possibilities, with over 750 apps supported, primarily targeting business productivity, project management and marketing automation tasks. A composition in Zapier consists in a trigger, followed by one or several actions, executed sequentially one after the other. Users can apply data filters to transform or filter the passed data between intermediary steps.

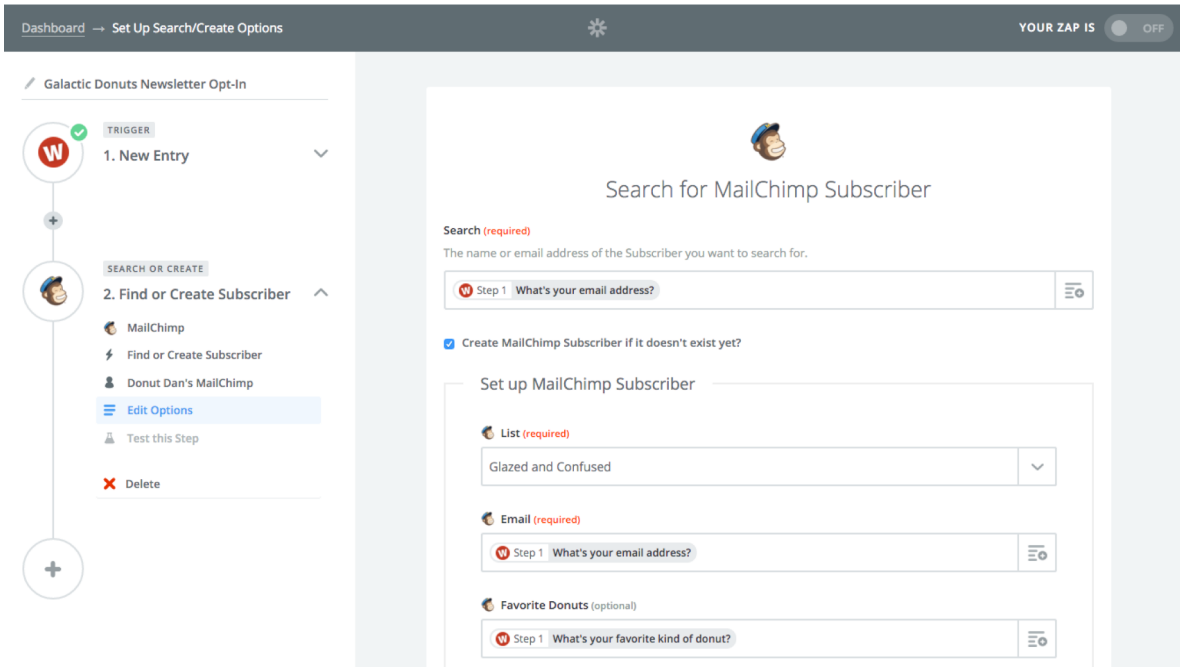


Figure 2.21 – A screenshot of a Zapier composition. It consists in automating a marketing campaign using Wufoo [13]and Mailchimp [14]. First, Wufoo is monitored for new form entries, then Mailchimp is updated accordingly with the subscriber's details.

12. https://zapier.com
14. https://www.wufoo.com
14. https://mailchimp.com

Azuqua

Founded in 2011, Azuqua [15] is a cloud-native Integration Platform-as-a-Service (IPaaS) that supports web service composition, with an emphasis on both IT governance, security and oversight [Hasija and Unger, 2015]. The Azuqua platform provides an intuitive user interface, enabling users to integrate their business applications by defining the data flow between services.

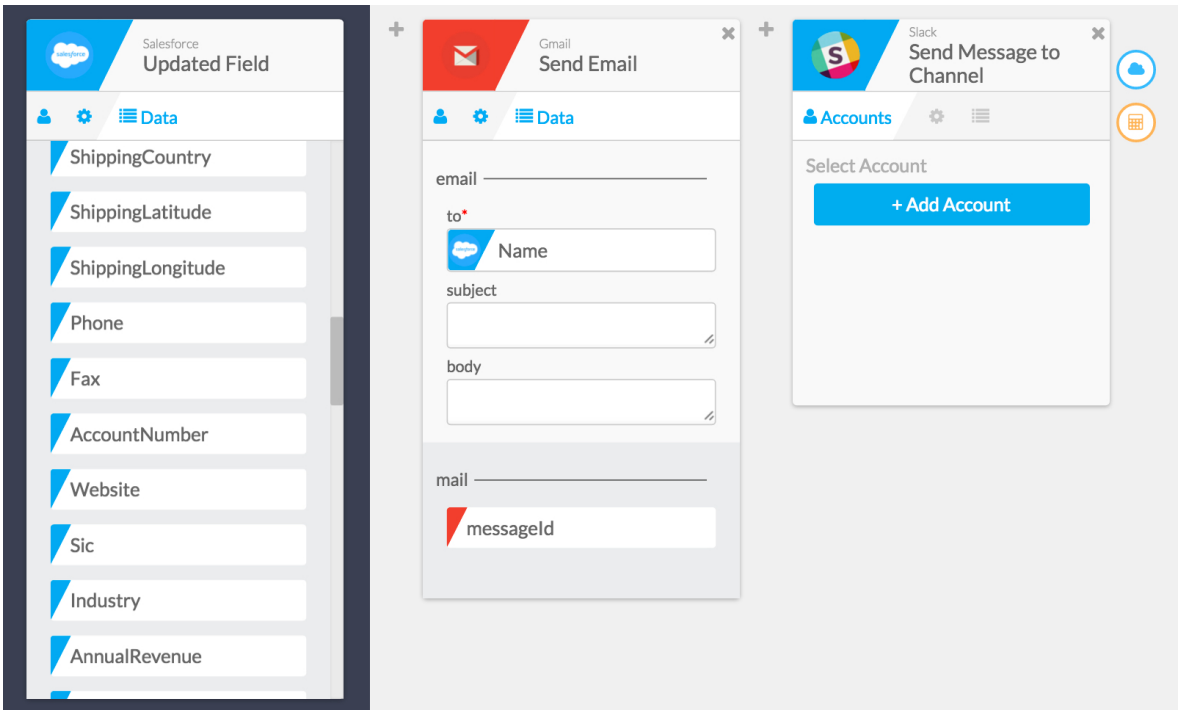


Figure 2.22 – A screenshot of an Azuqua composition. It consists in notifying the user about updates or changes in Salesforce [16], a customer relation management tool. First, Salesforce is monitored for changes in specific form fields. Then, for every change detected, the user is notified by email (Gmail [17]) and chat (Slack [18]).

15. http://azuqua.com
18. https://www.salesforce.com
18. https://www.google.com/gmail
18. https://slack.com

Workato

Launched in 2013, Workato [19] is an enterprise-oriented IPaaS. Trusted by over 21,000 organizations, it focuses on intelligent automations, enterprise integrations and process automation. Workato enables business users and IT to collaborate in order to build, operate and rollout automations while ensuring security and governance policies.

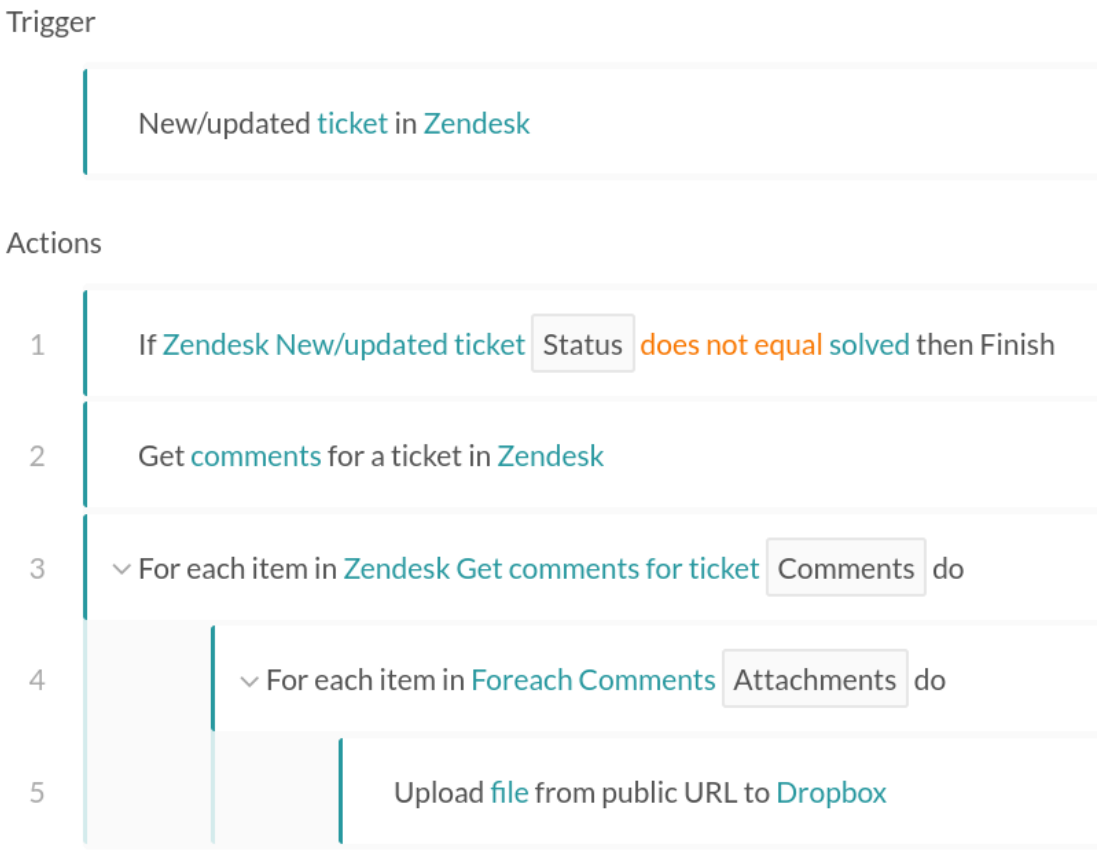


Figure 2.23 – A screenshot of a Workato composition. It consists in detecting new tickets that have been closed on Zendesk [20] (a customer support tool). Whenever a ticket is closed, the attachment documents are extracted from the comments section and uploaded on Dropbox [21] (cloud storage) for archival purposes.

19. https://www.workato.com
21. https://www.zendesk.com
21. https://www.dropbox.com

Microsoft Flow

Made publicly available in November 2016, Microsoft Flow[22] is a workflow management tool, offering an interface for connecting two or more cloud services in order to create business workflows, such as automating file synchronization, alerting, data organization, etc. It is particularly focused on integrations with Microsoft's own business tools, such as Office 365, Dynamics CRM, PowerApps, and Yammer, as well as those that are commonly used in organizations, like MailChip, GitHub, Salesforce, and Slack. Microsoft Flow also provides a mobile application for managing compositions and receiving alerts when an error occurs while running a composition.

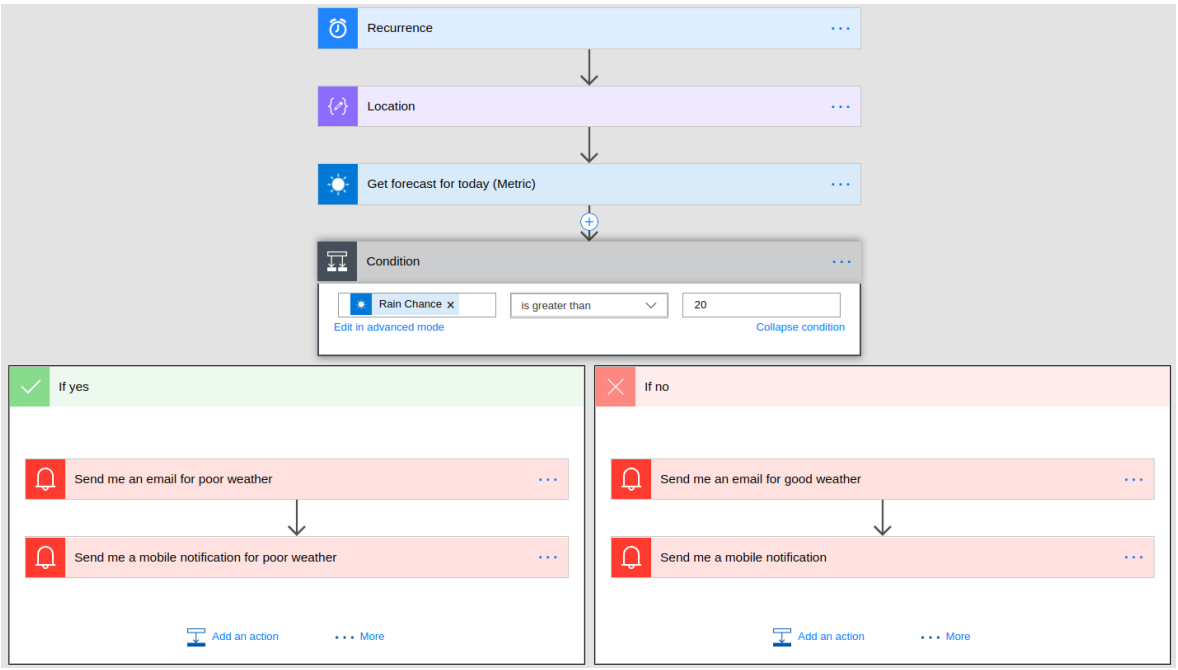


Figure 2.24 – A screenshot of a Microsoft Flow composition. It consists in using the MSN Weather[23] service to fetch the current weather at a given location. If there is more than 20% chance of rain, the user is notified about the poor weather by email and mobile notification. Otherwise, the user is notified about the good weather.

22. https://flow.microsoft.com
23. https://www.msn.com/en-us/weather

Summary

As the number of web services keeps on growing, many commercial platforms for service composition have been proposed over the past few years. We present here a brief overview of the leading commercial platforms presented in this section, and show how they compare to each other.

Expressivity. The platforms and solutions presented earlier do not provide the same level of expressivity. For instance, the IFTTT model has limited expressivity, as IFTTT compositions are limited to a single action per trigger, thus hindering the expression of more complex scenarios. Although other composition platforms do not have this restriction and enable users to express more complex compositions, they do not necessarily provide more advanced control-flow mechanisms, such as asynchronous/parallel invocations of services. Most of them also do not provide any looping constructs (except Workato and Microsoft Flow).

Extensibility. To fully benefit from a composition platform, users must be able to easily add support for any of their services. This translates into extending the platform by crafting an integration for the required service, in order to make it compatible and supported by the platform. However, the presented platforms offer many different ways to do so. For instance, IFTTT requires a premium partnership model in order to enable providers to add a new service. It enforces technical requirements that partners have to follow in order to integrate their services. On the other hand, Zapier, Azuqua, Workato and Microsoft Flow offer a more hands-on approach where developers have to configure dedicated connectors with the targeted platform. This is done either through a developer platform where developers integrate their APIs by specifying a form-based configuration, or by providing them with an SDK (Software Development Kit) to manually implement their own connectors. Likewise, they all impose specific requirements to enable the integration (description format, authentication protocol, etc.).

Offering. All commercial platforms presented above are provided as a hosted web application. Based on a freemium model, they offer different subscription plans, with varying features, services and customer support. However, due to the hosted nature of these solutions, they may not be suitable for large businesses or organizations which handle sensitive and business-critical data, as they have to compromise and expose at least a part of their internal network. In this context, an on-premise deployment of the composition platform is required to contain it within the bounds of the private network. By restricting its access to the local network of the organization, the privacy of confidential data and business processes is ensured [Na et al., 2010].

Error handling. As compositions interact with third-party services, it is inherently inevitable to encounter errors at some point. Being able to reason about such errors and to express the corresponding error handling logic is often critical to business-oriented tasks [Guidi et al., 2009]. For instance, users may require executing a different logic when a specific error occurs in a given service. However, with the exception of Workato, these platforms do not provide any error handling mechanisms to the users when expressing their composition logic. Instead, they resort to automatically retrying failed requests, logging the errors encountered and notifying the users about them.

2.5 Change detection in web resources

As presented in the previous section, composition platforms typically allow users to define and deploy service compositions, then triggering them whenever a particular event occurs on a given monitored third-party service. In the case of Fig. 2.19, the trigger consists in detecting if the forecast predicts rainy weather, in which case the rest of the composition is executed. In other words, this trigger consists in repeatedly polling the weather service and verifying if the response data changes between subsequent polls from *sunny* to *rainy*, for example. Thus, it is important to support a wide range of trigger events in order to meet the client's needs, scaling accordingly for all the services supported by the platform. The monitoring of web resources raises many challenges involving data collection and change detection [Abiteboul, 2002].

2.5.1 Data collection

Due to its distributed nature, the web is not a centrally managed repository of information. Rather, it consists of billions of independent content providers, each providing their own data and services across the web [Brewington and Cybenko, 2000]. As such, it becomes increasingly important to investigate techniques for exploring and gathering these resources [Douglis et al., 1997]. A number of research contributions focused on the challenges related to collecting data from the web. This led to the emergence of web crawlers. A web crawler is an automated system for exploring web resources (typically web pages) for different purposes [Olston et al., 2010]. Typically, web crawlers are notoriously used by seach engines to assemble and index large corpuses of resources [Fetterly et al., 2003]. This allows clients to issue queries against these indexes and find the matching resources rapidly and efficiently. Web crawlers are also used for collecting large sets of web pages and resources for archival[24] purposes, as well as for data mining, where web resources are collected and analyzed for statistical properties and data analytics [Baeza-Yates et al., 2007].

24. https://archive.org

Other solutions have been built around web crawlers to provide valuable services to their clients. For instance, web monitoring services such as Streamdata.io[25] and Giga Alert[26] allow their clients to submit standing queries, continuously monitoring the specified web resources, and notifying them about changes matching those queries. Such services lift the burden of repeated polling off of their clients, notifying them only about relevant changes. This is particularly important, considering today's growing use of mobile devices, as a particular focus needs to be given to energy efficiency and bandwidth usage [Dinh and Boonkrong, 2013].

2.5.2 Differencing algorithms

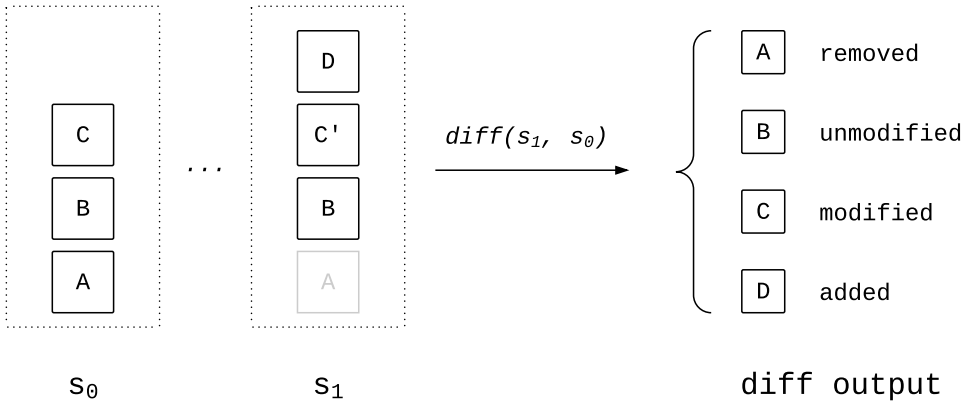


Figure 2.25 – An example of a diff between two different states of a given collection. The figure shows the states S_0 and S_1 of a collection captured at different times, and the resulting changes detected when comparing these two states.

In today's fast-paced web, data is continuously churning to reflect the latest state. Change detection consists in computing a diff between two documents, and identifying any relevant changes (see Fig. 2.25). Several existing contributions focus on improving the differencing process.

Although previous works focused on providing a framework for automatic detection of relevant changes on websites [Borgolte et al., 2014], they do not directly address change detection in REST APIs data, nor do they allow clients to specify what constitutes a relevant change. Their approach consists in representing documents as ordered or unordered labeled trees, and aim for optimizing the tree edit distance [Zhang and Shasha, 1989; Buttler, 2004; Bille, 2005]. Nonetheless, the problem of finding a minimal patch is $O(n^3)$ to NP-hard for ordered trees (depending on the set of operations considered), and NP-hard

25. https://streamdata.io
26. http://www.gigaalert.com

for unordered trees [Zhang et al., 1992; Pawlik and Augsten, 2011; Higuchi et al., 2012]. This leads to the use of practical heuristics that rely on the syntactical properties of the documents in order to provide reasonably good results [Lempsink et al., 2009]. As such, additional algorithms have been designed specifically for detecting changes in structured documents. For instance, several contributions enable change detection in XML by representing XML documents as ordered trees, then relying on the Longest Common Subsequence (LCS) algorithm [Hirschberg, 1977] to perform greedy heuristics for computing a minimal change set [Cobena et al., 2002; Lindholm et al., 2006]. Other alternatives consider XML documents as unordered trees, yielding higher quality results, although at the expense of greater runtime cost [Wang et al., 2003]. More recently, other algorithms have been proposed for JSON documents [Cao et al., 2016], which are a combination of unordered and ordered labeled trees, producing patches that are compatible with the JSON Patch RFC [Bryan and Nottingham, 2013]. Lastly, with today's growing use of mobile devices, a particular focus is given to energy efficiency. Producing minimal diffs becomes particularly important when dealing with mobile clients, as it helps reducing the bandwidth usage [Simon et al., 2014].

2.6 Summary

The literature presented in this state of the art mainly focuses on either formal concurreny models, or the composition of well-defined web services using BPEL and WSDL. However, none of these properly address the challenges raised by the composition of REST web APIs and microservices. Furthermore, the proposed solution needs to satisfy the requirements of CProDirect in terms of expressivity, reliability, scalability and performance. The design of the Medley platform draws its inspiration from existing solutions, while proposing a new approach for the composition of modern web services. To this extent, the underlying architectural framework supporting the Medley platform relies on:

(i) Polly, a high-level domain-specific language for describing change detection strategies in web service data (presented in Section 3.2),

(ii) Aria, an expressive domain-specific language for describing compositions of web services (presented in Section 3.3),

(iii) an event-driven, lightweight runtime supporting the execution of compositions specified using the Aria language, and triggered by events detected using Polly for custom change detectors (presented in Sections 4.1 and 4.2),

(iv) an efficient approach for scheduling composition executions in a distributed context, ensuring the scalability of the platform in the face of a growing userbase and third-party API rate limits (presented in Section 4.3).

CHAPTER

3 Domain-specific languages for service composition

We start this chapter by identifying several key challenges faced when composing several heterogeneous web services. To overcome these challenges, we propose two domain-specific languages (DSLs) that address the highlighted issues. First, we present POLLY, a high-level DSL for describing change detection strategies in web service data. POLLY simplifies the development of custom change detectors in order to trigger the execution of service compositions when a change occurs in web service data. We provide an in-depth description of the POLLY language constructs, and illustrate them through real use cases specified by CPRODIRECT. Then, we present ARIA, an expressive DSL for easily describing compositions of web services. We give an overview of the language architecture and semantics, and explain how ARIA provides the necessary abstractions to address the issues raised above in a simple and expressive way.

Contents

3.1 Challenges in web service composition

Distributed web applications are evolving at an increasingly high velocity, extensively leveraging existing services in order to offer a wide array of new features and functionalities. The emergence of the service-oriented paradigm has made it possible to build complex applications as a set of self-contained and loosely coupled services that work altogether in concert. Several languages, including BPEL, have been proposed to ease the orchestration of service compositions. However, they all fail in the context of modern web practices and microservice architectures, which are adopted by many major service providers. Therefore, existing approaches for orchestrating the composition of various services become unusable in practice. We illustrate issues that developers have to face in the remainder of this section.

3.1.1 Complexity of orchestrations

An orchestration may be triggered either manually (e.g. on demand), or automatically according to a given set of events, according to the user requirements (e.g. *"a new issue was created"*). To automate the execution of an orchestration, one typically needs to monitor a given service for new events or state changes, triggering the orchestration whenever specific events occur. This monitoring can be performed either synchronously by repeatedly polling the endpoint (pull mode) or by registering a callback for an asynchronous notification (push mode). When services only support polling, clients have to initiate a request to the server to retrieve the current state of the service. Then, the client compares this state with the previous one to detect any changes. Despite the advantages of push mode, developing applications based on the asynchronous paradigm is known to be challenging for many developers. When data needs to be propagated between subsequent asynchronous actions, the corresponding information has to be stored by the runtime system at the point of the asynchronous call. The runtime system then passes it back to the stored continuation function when the corresponding response is received. Integrating services based on active polling may also be challenging for the developer. He needs to set up a reasonable frequency for polling to avoid resources waste while preserving good responsiveness. When the same service is used several times, its invocations could be factorized among several clients. However, identifying such global optimization opportunities is difficult when the orchestration code is hard-written and each composition is developed independently from each other.

Example

To outline the multiple challenges involved when trying to detect changes in service data, we explain in details the scenario described in Section 1.2: *detecting new photos of a given Facebook album where Alice is tagged.*

```
{
  "data": [
    {
      "created_time":
          ↪ "2016-05-20T12:28:57+0000",
      "updated_time":
          ↪ "2016-05-20T12:26:57+0000",
      "id": "1106290499393017"
    }
  ],
  "paging": {
    "next":
        ↪ "https://graph.facebook.com/..."
  }
}
```

(a) Excerpt of a list of photos of a Facebook album.

```
{
  "data": [
    {
      "id": "10203528656797589",
      "name": "Bob",
      "created_time":
          ↪ "2016-05-20T12:39:01+0000",
      "x": 73.684210526316,
      "y": 74.865350089767
    }
  ],
  "paging": {
    "next":
        ↪ "https://graph.facebook.com/..."
  }
}
```

(b) Excerpt of a list of tags of a Facebook photo.

Figure 3.1 – Excerpt of photos and tags from the Facebook service.

In order to detect the new photos, one first needs to gather the complete list of photos of the Facebook album. This can be done by issuing a request on the https://graph.facebook.com/v2.9/:albumId/photos URL, where `:albumId` is the identifier of the photo album of interest. The Facebook service returns a response as a JSON document as illustrated in Fig. 3.1a. However, additional processing is needed to bridge the gap between the expected information and what is available in the returned document.

Firstly, the whole list of photos is not received at once, because the response is paginated (i.e. split in several lists of a fixed size). The `paging.next` attribute gives the URL to query to receive the next batch of photos. Additionally, the tags present on the photos are not part of this response. An additional request per photo is required to gather this information. This request can be made on the endpoint https://graph.facebook.com/v2.9/:photoId/tags where `:photoId` is the identifier of the photo of interest (received in response of the previous request). A request on the tags endpoint yields the result shown in Fig. 3.1b.

As we can see, this response is paginated as well. One can notice that the requests to gather the tags of each photo can be performed in an asynchronous manner, to improve performance. Finally, the tagged person names are available in these responses. To gather all the required information, the developer has then to manually construct a list that combines the photos and the tags data, as shown in Fig. 3.2a.

Performing a new polling operation using the same process would produce a new list of photos, as shown in Fig. 3.2b. By using an off-the-shelf differencing tool, the developer can compute the patch shown in Fig. 3.3. As it can be noticed, this patch contains two irrelevant changes: the `x` coordinate of the tag of the first photo and the last update time of the first photo. The only relevant change is the third one, where we can see a newly created photo

```
[
  {
    "created_time":
        ↪ "2016-05-20T12:26:57+0000",
    "updated_time":
        ↪ "2016-05-20T12:28:57+0000",
    "id": "1106290499393017",
    "data": [
      {
        "id": "10203528656797589",
        "name": "Bob",
        "created_time":
            ↪ "2016-05-20T12:39:01+0000",
        "x": 73.684210526316,
        "y": 74.865350089767
      }
    ]
  }
]
```


(a) Initial version.


```
[
  {
    "created_time":
        ↪ "2016-05-20T12:26:57+0000",
    "updated_time":
        ↪ "2016-05-20T12:29:57+0000",
    "id": "1106290499393017",
    "data": [
      {
        "id": "10203528656797589",
        "name": "Bob",
        "created_time":
            ↪ "2016-05-20T12:39:01+0000",
        "x": 76.684210526316,
        "y": 74.865350089767
      }
    ]
  },
  {
    "created_time":
        ↪ "2016-05-20T12:35:57+0000",
    "id": "2206280499393006",
    "data": [
      {
        "id": "20406528656797578",
        "name": "Alice",
        "created_time":
            ↪ "2016-05-20T12:45:57+0000",
        "x": 63.684210526316,
        "y": 62.865350089767
      }
    ]
  }
]
```


(b) Updated version.

Figure 3.2 – Initial and updated version.

containing a tag referring to user Alice. Therefore, the developer needs to post-process the patch produced by the differencing tool in order to construct the notification relevant to the scenario.

In this example we clearly show that detecting changes in service data is a tedious operation. It requires navigating across several endpoints, possibly chaining response elements into query parameters, and handling the problem of pagination at each step. When the data is gathered, an off-the-shelf differencing tool may produce irrelevant changes thus requiring either post-processing of the output or developing an ad-hoc differencing algorithm.

```
[
  {
    "op": "replace",
    "path": "/0/data/0/x",
    "value": 76.684210526316
  },
  {
    "op": "replace",
    "path": "/0/updated_time",
    "value": "2016-05-20T12:29:57+0000"
  },
  {
    "op": "add",
    "path": "/1",
    "value": {
      "created_time":
          ↪ "2016-05-20T12:35:57+0000",
      "id": "2206280499393006",
      "data": [
        {
          "id": "0406528656797578",
          "name": "Alice",
          "created_time":
              ↪ "2016-05-20T12:45:57+0000",
          "x": 63.684210526316,
          "y": 62.865350089767
        }
      ]
    }
  }
]
```

Figure 3.3 – JSON diff between the two versions of Figure 3.2.

3.1.2 Heterogeneity of unspecified interfaces

Existing orchestration languages such as BPEL require strongly-typed and well-defined interfaces from composed services. They typically enable orchestration of services by leveraging their static descriptions, thus expressing business processes as a set of operations and message exchanges between a number of services. These orchestration languages rely on description languages like WSDL that have been extensively used for many years. For instance, WSDL formally describes the service, specifying its location, its provided methods, how to bind to it, and how incoming and outgoing messages should be structured. Figure 3.4 presents an example of a WSDL description.

However, the current trend of microservice architectures promotes the use of RESTful services for which such formal service descriptions do not necessarily exist. Instead, service providers tend to simply provide human-readable documentation for the service,

```
<definitions>
  <types>
    <!-- Defines the data types used by the web service -->
  </types>
  <message>
    <!-- Defines the data elements being exchanged for each operation -->
  </message>
  <portType>
    <!-- Describes the operations that can be performed and the messages involved -->
  </portType>
  <binding>
    <!-- Defines the protocol and data format for each port type -->
  </binding>
</definitions>
```

Figure 3.4 – An example showing the XML structure of a WSDL description.

making it difficult to leverage exisiting solutions. Therefore, off-the-shelf tools are impractical in this context. In addition, services that provide similar content are often heterogeneous both in the format of data they provide and in the communication paradigm they rely on (synchronous *vs.* asynchronous). The developer has to account for all these details when building a service orchestration.

Example

To illustrate this issue, consider a custom daily news digest where a user receives an email containing information formatted to his liking about his favorite news from different sites (see Fig. 3.5). The developer has to manually specify how to interact with these news providers, what information to retrieve and how to aggregate data to produce a curated digest, and finally email the result. As the number of services increases, this task becomes laborious.

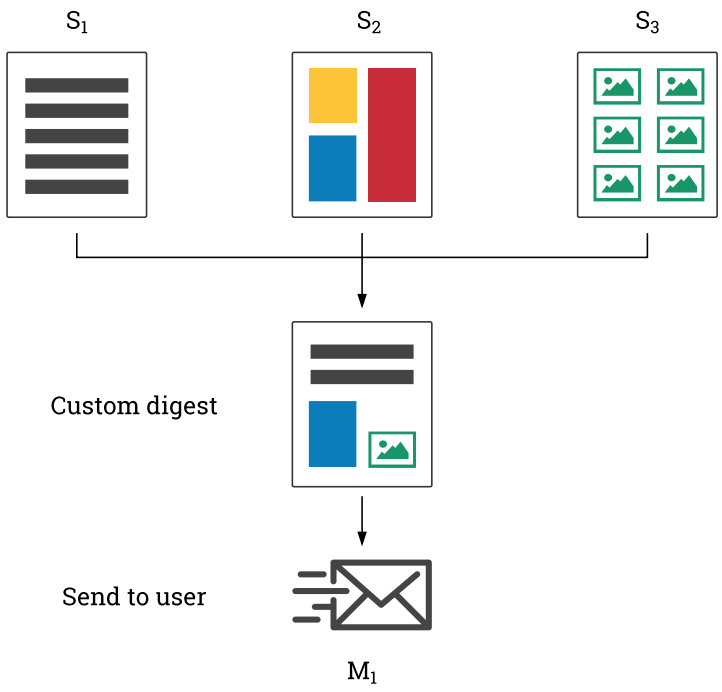


Figure 3.5 – A composition for aggregating data from different services (S_1, S_2 and S_3), then building a curated digest by extracting relevant data, and finally sending it to the client using an emailing service (M_1).

3.1.3 Dynamicity of service composition

Compositions of services are usually statically specified and make explicit the connections between the interacting composed services. This design-time coupling prevents an orchestration from dynamically adapting its behavior when new services are deployed, undeployed or upgraded. Although the microservices architecture promotes dynamicity, it does not provide any insights on how to achieve it in practice. Supporting adaptation at runtime is known to complexify the task of the developer as he needs not only to focus on the orchestration of several services, but also on how to smoothly react to service changes.

Example

As an example, consider the custom daily news digest orchestration scenario. To prevent failure in case the mail service becomes unavailable for some time, the user should ideally be able to specify a pool of mail services that can be used interchangeably (see Fig. 3.6). However, defining such dynamic service selection policies in existing orchestration languages is limited, and requires explicit handling of all errors and edge cases by the user, making it tedious to maintain.

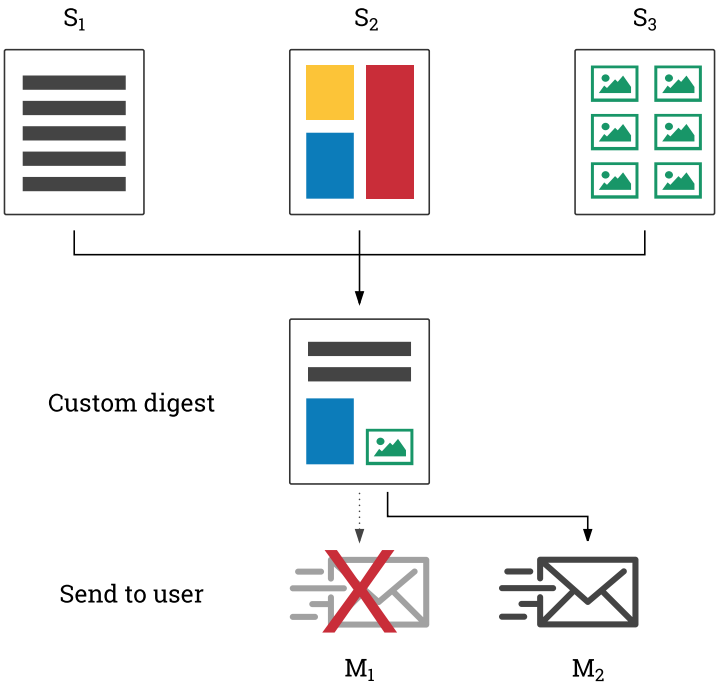


Figure 3.6 – The orchestration should be able to adapt in the face of service outages (M_1), and failover to other compatible services (M_2) in order to ensure its complete execution.

3.2 POLLY: a DSL for custom change detection of web service data

An ever-growing number of web service providers expose data that is continuously changing. Use cases arise where being notified about changes made to the data is essential to the client, for instance to know when a user has a new follower on Twitter. Monitoring changes on web services data consists in polling services for the required data, detecting any changes in the targeted data subset, and notifying the user only about the relevant changes. However, each step of this process can be relatively complex, leading to a tedious and challenging implementation for developers.

In our context, CPRODIRECT, wishes to compete with traditional platforms by enabling fast integration of new service providers and events in its own platform [Ben Hadj Yahia et al., 2016b]. To reduce time to market, we investigate the challenges of detecting changes in web service data. We focus on modern web services that follow the REST architectural style and exchange data with their consumers in JSON. In this section, we introduce POLLY, a domain-specific language for defining custom change detection strategies in web service data. By leveraging the domain knowledge of the user, POLLY offers declarative, concise yet highly-expressive constructs for specifying custom change detectors. We present the language constructs and illustrate our approach using several user-driven scenarios provided by CPRODIRECT.

3.2.1 Overview of the POLLY language

The POLLY language is based on the YAML [Ben-Kiki et al., 2005] syntax and is implemented as a Node.js module. Inspired by dataflow architectures, it enables users to express and define custom change detectors in the form of processing pipelines. A pipeline is expressed as a series of operations that are applied on successive sets of data, where data and operations on it are independent from each other.

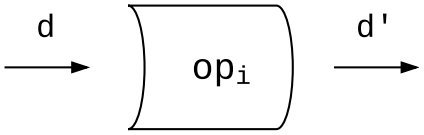


Figure 3.7 – In POLLY, an operation accepts a JSON document as input, processes it according to a given logic, and produces a new JSON document as output.

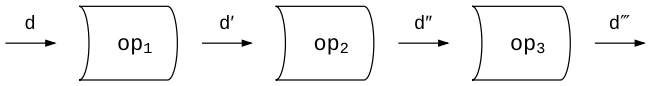
To build such pipelines, POLLY provides an extensible set of operations, where each operation performs a specific task. As Fig. 3.7 demonstrates, an operation accepts a JSON document as input, processes the input document according to a specific logic, and finally produces an output document that is passed as input for the following operation. Users

can specify additional parameters to fine-tune the operation, and refine the produced outcome. These parameters are specific to each operation type. Provided operation types are further detailed in the following section. Furthermore, POLLY provides mechanisms to extract, transform and template data using JSONPath and standard dot notation, allowing users to customize the manipulated data, keeping only relevant parts and discarding the rest.

Using the provided operations, users can build custom change detectors by chaining multiple operations together, in a way that is conceptually similar to piping Unix processes (see Fig. 3.8). POLLY allows the user to specify how to compute a state by fetching a set of API resources, how to detect custom changes that are relevant to his requirements, and how to build a custom output to match the expected outcome. The provided language operators and constructs are described at greater length in the following section.

```
1 pipeline:
2   - operation: op1
3     definition: # ...
4
5   - operation: op2
6     definition: # ...
7
8   - operation: op3
9     definition: # ...
```

(a) POLLY specification for defining a pipeline.



(b) Visual representation of the POLLY pipeline presented in Fig. 3.8a.

Figure 3.8 – An example of a POLLY processing pipeline.

3.2.2 Specification of the POLLY language

In this section, we introduce POLLY, a declarative language-based approach that raises the level of abstraction by providing dedicated operators to express state construction, change detection, and output construction within a pipeline of operations. We describe here how our approach enables one to simply design efficient custom change detectors for web service data, allowing developers to only focus on their domain knowledge of the manipulated services. Figure 3.11 gives the BNF specification of the POLLY language grammar.

Language constructs. By design, each operation processes an input value (represented by the "_" symbol), and produces an output value (represented by the "&" symbol). These default values can be overridden using the *input* and *output* keywords at the operation level. Furthermore, POLLY introduces three additional notations. The "~" symbol refers to the response body of a request (Fig. 3.9a, lines 11 and 13), while the "%" symbol refers to

```
1  - operation: fetch
2    definition:
3      request:
4        url: https://graph.facebook.com
             ↪ /v2.9/:albumId/photos
5        params:
6          albumId: 465607303461343
7        query:
8          access_token: XXXX
9        headers:
10         Accept: application/json
11       template: ~.data
12       pagination:
13         next: ~.paging.next
14     # Or, instead of using template:
15     # output: &:$..data
```

(a) POLLY specification for fetching a list of photos for a given album.

```
1  - operation: fetch
2    definition:
3      repeat:
4        forEach: _
5        placeholders:
6          photoId: ^.id
7      request:
8        url: https://graph.facebook.com
             ↪ /v2.9/:photoId/tags
9        query:
10         access_token: XXXX
11       headers:
12         Accept: application/json
13       pagination:
14         next: ~.paging.next
15       template:
16         photoId: ^.id
17         tags: ~.data
```

(b) POLLY specification for fetching a list of tags for each album photo.

Figure 3.9 – A minimal example showcasing how to retrieve all photo tags of a Facebook album using POLLY.

the response headers. The "^" symbol represents the loop iteration cursor (Fig. 3.9b, lines 6 and 16). This cursor represents the current element being iterated on. All five notations presented in this paragraph support the dot notation for accessing child properties. For example, `~.data` references the `data` attribute at the root of the response document.

Evaluating JSONPath expressions. POLLY relies on the JSONPath specification [Goessner, 2007] to describe the selection of a sub-document, as illustrated in line 15 of Fig. 3.9a. This enables users to easily extract the sub-documents of interest. Thus, a JSONPath expression [1] can be applied on any of the previous symbols, using the following notation: `[symbol]:[jsonpath_expr]`. For instance, the evaluation of the expression `&:$..id` is equivalent to evaluating `$..id` on the output document (`&`), thus producing all the `id` fields present in the output document.

3.2.3 State construction

The *fetch* operator enables the user to specify how to collect data from a set of API endpoints. These details are specified within the *request* block (Fig. 3.9a, line 3). Here, the user defines the resource URL using the *url* keyword (line 4). The URL can have parameter placeholders (prefixed by a colon), which are substituted with the matching key from the

1. The $ symbol represents the root of the current document in JSONPath.

params block (line 5). Furthermore, the DSL offers the ability to specify query parameters (*query*, line 7) as well as HTTP headers (*headers*, line 9) as key-value pairs.

Templating. In the majority of use cases, the user only requires gathering a subset of the collected data. Furthermore, he might also need to include extra information along with the response. The *template* keyword allows specifying a transformation template. This can be expressed directly as an expression, or as a new set of keys where each corresponding value is an expression. For example, line 11 of Fig. 3.9a shows how to extract the `data` object from the API response (Fig. 3.1a, line 2). Another example occurs in line 15 of Fig. 3.9b where we fetch photo tags. Here, we define a new template containing the original photo ID and its tags. This transformation is necessary in order to manually include the photo ID (which is not part of the API response) in the final state.

Pagination. The *pagination* keyword enables the user to indicate how to fetch subsequent pages when the response is paginated (Fig. 3.9a, line 12). Information about pagination is typically present in an HTTP header or in the body of the response. For example, GitHub returns the full URL of the next page in the `Link` header, while Twitter provides just a cursor for the next page in the body of the response. Other APIs such as Stack Exchange require the user to manually specify the page number as a query parameter when requesting a resource, but do not provide any information about the current or next page number in the body of the response. Instead, they just indicate if there are subsequent pages using a boolean value in the body of the response. To support all these pagination methods, POLLY enables the user to specify how to navigate to the following page using the *next* keyword (line 13). This keyword accepts either an expression containing the full URL of the next page, or key-value pairs specifying the name and value of the query parameter used for pagination (*queryParam*, defaults to the value `page` and auto-incremented by default). After collecting all subsequent pages, the results are flattened in a single array and returned as the output of the operation.

Parallel fetch. In the Facebook example presented in Section 3.1.1, the user has to first retrieve a list of photo IDs for a given album, then retrieve the tags for each photo. To enable this scenario, POLLY provides the *repeat* keyword (Fig. 3.9b, line 3). This keyword allows specifying an iteration set from the output of the previous operation (*forEach*, line 4), and corresponding placeholder labels (*placeholders*, line 5). These placeholders are substituted in the URL by their value, thus executing a *request* for each constructed URL. In the Facebook example, this corresponds to fetching the tags for each album photo. By default, all requests are asynchronous and performed in parallel. The output of this operation contains a list of templated objects (line 15), where each object includes the current photo ID and the list of tags for a given photo (e.g. Fig. 3.1b).

```
1 - operation: filterArray
2   definition:
3     input: _
4     identifiers:
5       - ^.photoId
6     find:
7       - addedItems
8     output: &.addedItems
```

(a) Specification for detecting new photos.

```
1  - operation: filterCustom
2    definition:
3      function: !!js/function >
4        function (existing, input) {
5          const result = [];
6          input.forEach((item) => {
7            const isTagged = item.tags.some((value) =>
                 ↪ {
8              return value.name === 'Alice';
9            });
10           if (isTagged) { result.push(item.photoId);
                 ↪ }
11         });
12         return { type: "addedTags", items: result };
13       }
```

(b) Custom change detection specification.

Figure 3.10 – Detecting new photos where Alice is tagged using POLLY.

3.2.4 Change detection

After computing the state in the previous step, the user can now proceed to specifying a change detection strategy. Our preliminary case studies showed that changes to a JSON document can occur on objects or arrays, and range from additions and deletions, to value modifications and order changes. In light of these results, the POLLY DSL provides several filtering operators for change detection: *filterObject*, *filterArray* and *filterCustom*. The *filterObject* (resp. *filterArray*) operator accepts an expression of object (resp. array) type as an input. The *filterCustom* operator enables the user to define custom filtering logic.

Change types. The *find* keyword enables defining a list of change types to detect in the input of the operation (Fig. 3.10a, line 6). The list of supported change types is presented in Table 3.1. For each change type listed in the *find* block, a matching object is included in the output of the operation, containing the corresponding data. For instance, listing *addedItems* and *removedItems* in the *find* block would produce as output an array of two objects, each having *addedItems* (resp. *removedItems*) as types, and each having a list of the items that have been detected as recently-added (resp. recently-removed).

Per-change type templating. Although the *template* keyword presented in Section 3.2.3 is also supported in this operation, one might need to specify different templates for different change types. To meet this requirement, POLLY supports an additional keyword *templates* (mutually exclusive with *template*). This keyword allows specifying the change type (e.g. *addedItems*) as key, and the associated template as value.

Table 3.1 – List of supported change types.

	`filterObject`	`filterArray`
Change types	*addedKeys* *removedKeys* *modifiedKeys* *unmodifiedKeys*	*addedItems* *removedItems* *modifiedItems* *unmodifiedItems* *movedItems*

Targeted monitoring. By default, all keys of the input document are watched for modifications, and any change would mark the document as modified. The optional keyword *watch* can be used to restrict the set of keys to watch for modifications. This enables the user to define what actually constitutes a relevant change. Note that for objects, a key is marked as modified (resp. unmodified) if the value corresponding to the key specified in the *watch* block is modified (resp. unmodified). For arrays, an item is marked as modified (resp. unmodified) if **any** (resp. **all**) of the values corresponding to the keys specified in the *watch* block are modified (resp. unmodified).

Custom item identification. Additionally, when dealing with array items, it is necessary to uniquely identify the items throughout subsequent polls. This allows us to know for example if a given item has been added or removed during the polling interval. However, not all APIs provide unique identifiers on all of their resources. Moreover, these identifiers can be present under different key labels. For this reason, we provide an additional keyword called *identifiers*, which allows the user to specify how to uniquely identify an item within a collection (line 4). This can be as simple as providing the path to the `id` field of an item, a list of fields (e.g. first and last names of a user), or a wildcard to hash the entire item and use it as its own identifier.

Custom filtering. When none of the previous operators are adequate, the *filterCustom* operator can be used to implement one's own custom filtering logic. Figure 3.10b shows an example of how to filter a list of photos by only selecting those where Alice is tagged. This operator provides a hook function with the previous and current states as parameters (line 4). The user can implement this hook in JavaScript, returning a custom output. In this example, the user iterates on the input array of photos (line 6) and checks whether if Alice is tagged on the current photo (lines 7-9), in which case he retrieves the photo ID (line 10). To avoid any security issues when running user-provided code, this function is executed within an isolated sandbox at runtime.

```
start       ::= pipeline:  operation+
operation   ::= (fetch | filterArray | filterObject | filterCustom)

fetch       ::= operation: fetch NL fetchDef (NL output)?
fetchDef    ::= definition: NL INDENT
                 (repeat NL)?
                 req
                 (NL template)?

filterArray ::= operation: filterArray NL fArrayDef (NL output)?
fArrayDef   ::= definition: NL INDENT
                 findInArr
                 (NL input)?
                 (NL watch)?
                 (NL identifiers)?
                 (NL template)?
                 (NL templates)?

filterObject ::= operation: filterObject NL fObjectDef (NL output)?
fObjectDef   ::= definition: NL INDENT
                 findInObj
                 (NL input)?
                 (NL watch)?
                 (NL template)?
                 (NL templates)?

filterCustom ::= operation: filterCustom NL fCustomDef (NL output)?
fCustomDef   ::= definition: NL INDENT
                 function: fn
                 (NL input)?
```

Figure 3.11 – BNF specification of the POLLY language grammar (continued on next page).

```
repeat      ::= repeat: NL INDENT
                 forEach: string
                 (NL placeholders: object)?
req         ::= request: NL INDENT
                 url: string
                 (NL params: object)?
                 (NL query: object)?
                 (NL headers: object)?
                 (NL pagination: pagingOpts)?
pagingOpts  ::= pagination: string
              | pagination: NL INDENT
                 next: string
                 (queryParam: string)?
template    ::= template: (object | string)
templates   ::= templates: object
findInArr   ::= find: (addedItems | removedItems | modifiedItems |
                       unmodifiedItems | movedItems)+
findInObj   ::= find: (addedKeys | removedKeys | modifiedKeys |
                       unmodifiedKeys)+
watch       ::= watch: strlist
identifiers ::= identifiers: strlist
fn          ::= function: !!js/function > jsFunction
input       ::= input: string
output      ::= output: (object | string)
```

Figure 3.11 – BNF specification of the POLLY language grammar (continued from previous page).

3.3 ARIA: a DSL for web service composition

To abstract away the low-level details from the users when composing heterogeneous services, we introduce in this section ARIA, a highly-expressive DSL that enables users to express service compositions from a higher abstraction level as opposed to several other orchestration languages. We present an overview of the language architecture, and show how ARIA enables the development of various compositions, involving a large number of existing services.

3.3.1 Overview of the ARIA language

Inspired by flow-based programming and the event-driven communication paradigm, ARIA enables users to reason and to focus on business logic rather than be disrupted by low-level technical implementation details and intricacies. To create a composition in ARIA, users mainly have to specify the set of services they wish to use, and the composition logic. This consists in defining the control flow that reflects which and when services should be invoked, and how runtime errors and exceptions should be handled. Furthermore, users can easily express data flow by extracting, transforming and passing data between services.

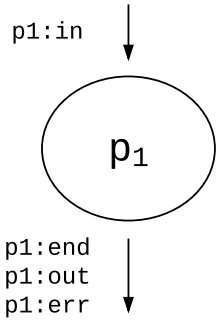


Figure 3.12 – An ARIA process listens on its input stream, and produces events on its output stream.

To enable these features, the ARIA language provides an abstraction layer to facilitate these tasks. As such, third-party web services are wrapped and exposed to the user as processes. The role of a process is twofold. First, it provides a wrapper to hide the low-level technical details for invoking the service, such as the boilerplate code that needs to be written to properly construct the request, passing in any optional or required arguments, specifying HTTP headers and authentication tokens, etc. These are hidden and abstracted away from the users. Once defined, a process can be used from any composition, thus avoiding code duplication across compositions and facilitating maintenance should a process needs updating or debugging. Second, an ARIA process exposes to the users event-based

control flow mechanisms for invoking services. As demonstrated in Fig. 3.12, a process `p1` listens for its input event, labeled `p1:in`. Whenever such an event is received, the process invokes the wrapped service. It then emits on its output stream either an event carrying the response of the invoked service if the request was successful (event of type `p1:out`), or emits an error event if the request failed or if a runtime exception is encountered (event of type `p1:err`). Finally, the process emits an event of type `p1:end` to signal the end of the processing.

Furthermore, the ARIA language natively supports JSON and JSONPath expressions, to simplify data manipulation. It also provides a way to invoke arbitrary JavaScript code at runtime. These mechanisms are explained at greater length in Section 3.3.2.

Example

To illustrate these concepts, consider the following scenario. A user needs to be automatically notified about new high-priority issues on a given GitHub [2] repository. He also needs to be notified if the composition fails at retrieving these issues. Figure 3.13 gives an overview of this composition, and shows how ARIA processes can be assembled together to achieve this.

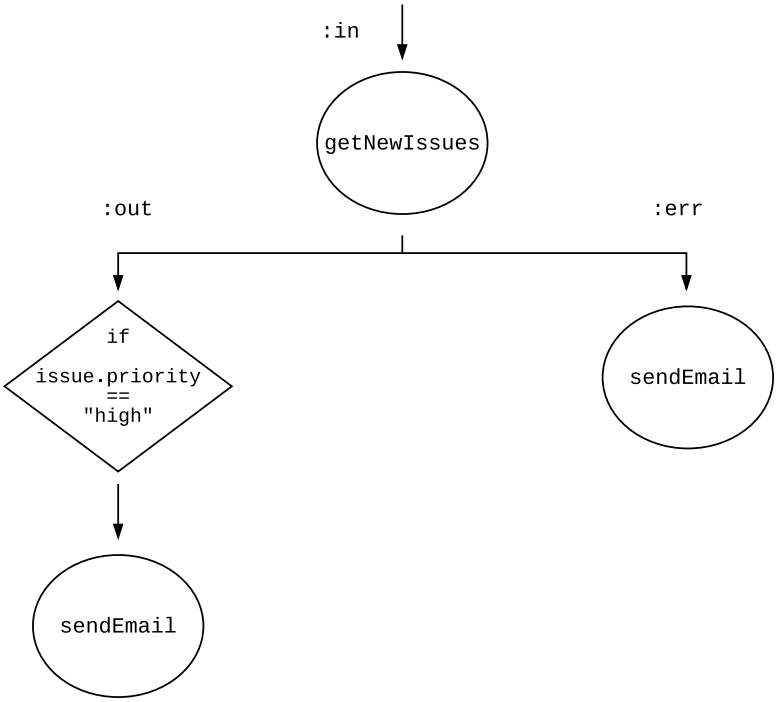


Figure 3.13 – An example of an ARIA composition.

2. A Git repository hosting service

```
1  composition {
2    process getNewIssues = require("Github/GetNewIssues");
3    getNewIssues.init({ "credentials": "<label>" });
4    process gmail = require("Gmail/SendEmail");
5    process outlook = require("Outlook/SendEmail");
6    // ...
7    pool process sendEmail = require("Medley/Pool");
8    sendEmail.addToPool([gmail, outlook]);
9    sendEmail.init({ "strategy": "round-robin" });
10
11   stream issues = getNewIssues.invoke({ "repository": "medley/hello-world" });
12   on (issues:out as issue) do {
13     if (issue.priority == "high") {
14       sendEmail.invoke({
15         "to": "john@doe.com",
16         "body": "New issue: " + jp.value(issue, "$.url")
17       });
18     }
19   }
20   on (issues:err as error) do {
21     sendEmail.invoke({
22       "to": "john@doe.com",
23       "body": "Error encountered while fetching new issues: {{error.message}}"
24     });
25   }
26 }
```

Figure 3.14 – A composition example using ARIA DSL.

3.3.2 Specification of the ARIA language

In the remainder of this section, we rely on the example scenario presented in the previous section, and provide its specification using the ARIA language in Fig. 3.14. This scenario serves as a running example to illustrate the ARIA DSL constructs, with the help of Fig. 3.18 that gives the BNF specification of the language grammar.

Figure. 3.14 describes a composition that checks for new high-priority issues created on a specific GitHub repository (line 11). If a new issue is detected, it notifies the user by sending her an email containing the issue's URL (lines 12 to 19). The email service is selected from a pool of interchangeable services, enabling fault-tolerance on service unavailability (lines 7 to 9). It also notifies the user if an error is encountered with the GitHub service when polling for new issues (lines 20 to 25).

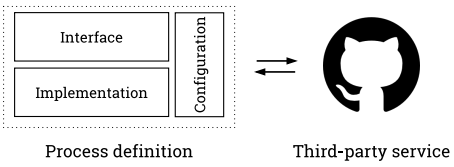
Furthermore, this example enables us to highlight some key language operators of the ARIA DSL, which are described at greater length hereafter.

Composition definition

In ARIA, services are mapped to processes. The DSL allows users to configure processes to use and express how to compose them altogether according to the events that can occur on their respective output streams. The `composition` keyword (Fig. 3.14, line 1) enables

```
1 process getNewIssues =
      ↪ require("Github/GetNewIssues");
```

(a) Requiring a process in ARIA.



(b) Underlying components of an ARIA process.

Figure 3.15 – Requiring ad-hoc processes in ARIA.

defining the body of the composition as a set of instructions. The `process` keyword (lines 2, 4, 5, 7) enables declaring a new process variable. Processes support the `init` method (lines 3, 9), allowing the user to configure the process with initialization parameters. These parameters persist throughout the lifecycle of the process instance.

Requiring ad-hoc processes

To ease the use of the ARIA language and favor code reusability, requests to third-party services are not explicitly defined in the composition specification. Instead, the request logic and configuration is implemented in ad-hoc processes by the service providers. Service providers are in charge of defining and maintaining reusable black-boxes which implement the interaction logic with the desired services (see Fig. 3.15). The process interface statically describes it, providing metadata about the service as well as type information about the expected input and the produced output of the process, using the JSON Schema specification [Galiegue et al., 2013; Pezoa et al., 2016].

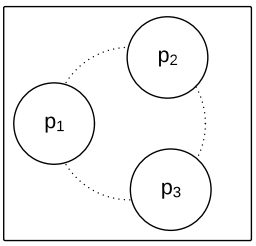
Once implemented, processes are deployed to an internal process repository, in a plugin-like fashion. The processes are then indexed and become available for use on the underlying execution platform. To enable loading an existing process, the `require` function (lines 2, 4, 5, 7) is provided globally and serves as an import mechanism for instanciating processes. `require` returns a new instance of the specified process. Processes are looked up by name and loaded from the internal process repository.

Stream processing

The `invoke` method (lines 11, 14, 21) allows the user to invoke a process with a set of arguments. When invoked, the process returns a reference to its output stream (line 11). The events of an output stream are tagged according to their types: `out` for successful executions (line 12), `err` for erroneous executions (line 20), and `end` to signal the end of stream. Thus, users can listen to these event types using the `on` construct (lines 12, 20), then react according to the event type by specifying the corresponding handler. Each event carries along a payload (response output data or error message), and can be labeled using the `as` keyword (lines 12, 20). A process invocation can yield 0, 1 or n events, according to its implementation logic.

```
1 pool process notify = require("Medley/Pool");
2 process p1 = require("P1/SendMessage");
3 process p2 = require("P2/SendMessage");
4 process p3 = require("P3/SendMessage");
5 notify.addToPool([p1, p2, p3]);
6 notify.init({ "strategy": "round-robin" });
```

(a) Defining a process pool in ARIA.



(b) Visualisation of the `notify` process pool in Fig. 3.16a

Figure 3.16 – Dynamic process pools in ARIA.

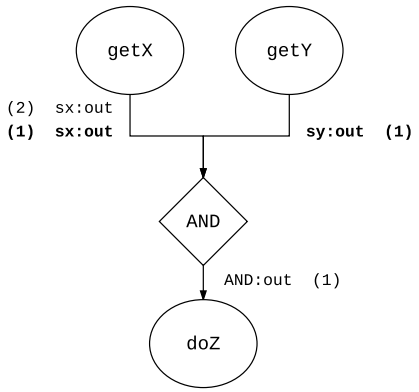
Process invocations separated by semi-colons are executed in an asynchronous manner. As such, `p1.invoke(); p2.invoke();` represents a parallel execution of both `p1` and `p2` processes. If a sequential ordering is required, `on` blocks can be nested to invoke the first process, wait for the `end` event of the first process' stream, then invoke the second process. The execution of a composition is finished when the streams of all invoked processes are closed.

Dynamic process pools

ARIA also provides a construct to specify pools of interchangeable processes, using the `pool` keyword. More specifically, it consists in a set of processes that share a common interface, and are semantically equivalent (i.e. they can fulfill the same functional need). A process pool is typically used to allow a composition to dynamically bind to a service or adapt to service outages, all while being transparent to the developer (see Fig. 3.16). For instance, emailing services such as Gmail and Outlook are considered to be interchangeable since they all provide the same base functionality and have a compatible interface (e.g. recipient address, title, message body, etc.). An example is presented in Fig. 3.14, lines 7 to 9. A verification is performed at compile time, to ensure that all processes of a pool share a common interface. Formally, two services are type compatible if there exists a bijection between their respective sets of mandatory input types such that each pair in the bijection is compatible (identical field name and field type). Process pools can be configured using different pre-defined strategies. For example, line 9 shows that a `round-robin` strategy is used to alternate between the Gmail and Outlook service providers. Other natively supported strategies include `fallback` (always use the first process unless an error occurs, in which case fallback to the next process, and so on) and `random` (randomly select a process for each invocation).

```
1 stream sx = getX.invoke();
2 stream sy = getY.invoke();
3 on (sx:out as x and sy:out as y) do {
4   doZ.invoke({ "body": "X is worth
        ↪ {{x.value}} while Y is worth
        ↪ {{y.value}}" });
5 }
```

(a) Using the AND operator in ARIA.



(b) FIFO ordering of incoming events.

Figure 3.17 – Joining two streams using the AND operator.

Control flow

Users may need to invoke several services in parallel, and join their output streams, for instance to aggregate data from different sources before performing an action. For this purpose, we introduce the `and` operator (see Fig. 3.17). It allows users to express synchronization points when invoking multiple asynchronous processes. The `and` operator is implemented as a built-in process that generates an output event only when it receives an event from both its two input streams. Incoming events are buffered in a circular FIFO memory enabling the runtime to provide load shedding by discarding events that occur more frequently from one source than the other. For each discarded event, an error event is generated on the error stream allowing the composition to react to it. If the aggregated services take too much time to respond, the memory is flushed.

The language also provides basic control flow constructs, with the `if/else` keywords. These constructs provide filtering capabilities on data from output events and can be used to conditionally execute a branch of the program. For example, line 13 of Fig. 3.14 shows how to express the invocation of the `sendEmail` process only when the value of the priority field is high.

Data flow

A crucial aspect in composing multiple web services is being able to reuse and pass data from a service to another. ARIA provides the necessary mechanisms to have fine-grain control over the data, such as on-the-fly substitution and evaluation of expressions, as well as document traversal and templating.

To enable the extraction of data from inbound events, ARIA supports the use of property accessors using the dot notation, as well as the use of JSONPath expressions [Goessner, 2007] for subdocument extraction. JSONPath is the XPath [Urpalainen, 2008] equivalent

```
comp      ::= composition { decl+ rule+ }
decl      ::= pool? process ident = require ( string );
            | ident.init ( json? );
            | ident.addToPool ([ ident (, ident)* ]);
rule      ::= on event do { action+ }
event     ::= evt | event and evt | ( event )
evt       ::= evt_kind (as ident)?
evt_kind  ::= ident : out | ident : err | ident : end
action    ::= stream ident = ident.invoke ( json? ) ;
            | ident.invoke ( json? ) ;
            | if ( expr ) action (else action)?
            | rule
expr      ::= ! expr | expr binop expr | ( expr )
            | ident | string | integer | float
            | method ( expr* (, expr)* )
method    ::= ident | jsonpath . ident
binop     ::= < | > | <= | >= | == | != | && | ||
```

Figure 3.18 – BNF specification of the ARIA language grammar.

for JSON documents. It provides a set of operators to traverse JSON documents from their root (noted as $), and selectors to match queries on document attributes. In the snippet presented in Fig. 3.14 (lines 13 and 23), we use the dot notation to access data properties of the corresponding events. In line 16, the global helper `jp` is used to evaluate a JSONPath expression against an event payload. The `jp.value` method returns the first value that matches the expression, whereas the `jp.query` method returns all matching values. In line 23, we use the double curly braces notation `{{...}}` as templating placeholders for string interpolation. These expressions are evaluated at runtime, and placeholders are replaced with their corresponding values.

Furthermore, ARIA also provides an environment for evaluating expressions on primitive types. The evaluation environment is accessed through `<@ expr @>` delimiters, where `expr` is the expression to evaluate. Valid expressions are a restricted subset of JavaScript functions. For instance, the expression `<@ Date.now() @>` evaluates to the current date. As such, users can easily manipulate and transform data through evaluated expressions. At runtime, a pre-processing phase takes place, where expressions are first substituted with their appropriate values, and then evaluation environments are resolved.

3.4 Summary

We presented in this chapter an overview of the main challenges developers face when orchestrating web services. To address these issues, we proposed two domain-specific languages to raise the level of abstraction when dealing with the composition of modern web services.

First, we identified the potential of triggering the execution of service compositions when changes occur in web service data. To this extent, we introduced POLLY, a high-level DSL for describing custom change detectors in web service data. We showed how POLLY enables users to define custom change detection strategies using high-level constructs, abstracting away the underlying complexities. Then, we also introduced ARIA, an expressive DSL for describing the orchestration of web services using an event-driven paradigm. We showed how ARIA enables users to describe complex composition scenarios in a simple yet powerful way. In the following Chapter 4, we present the implementation details of the MEDLEY platform, a runtime system that supports the execution of ARIA compositions that can be triggered by POLLY change detectors.

CHAPTER

4 Runtime system implementation

In this chapter, we present MEDLEY, an event-driven lightweight platform for service composition. MEDLEY supports the execution of compositions specified using the ARIA language, and triggered by events detected using custom POLLY change detectors. We describe the implementation details of the MEDLEY platform and show how it supports the execution of web service compositions in a scalable manner.

Contents

4.1 An event-driven lightweight platform for service composition

In this section, we rely on Fig. 4.1 as a illustration to describe the architecture of the MEDLEY platform.

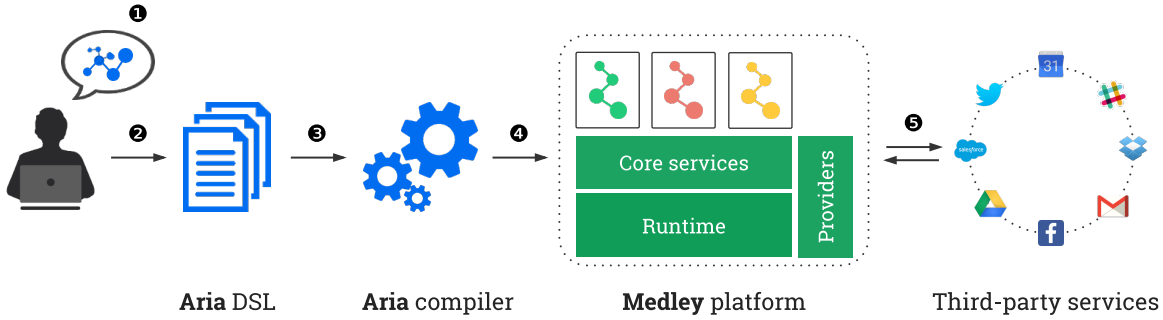


Figure 4.1 – Steps involved in running a composition on the MEDLEY platform.

Based on a particular set of services to compose (Fig. 4.1 ❶), a user specifies, via the use of the ARIA DSL presented in Section 3.3, two kinds of information: (i) how to assemble together the services, and (ii) the composition logic (Fig. 4.1 ❷). In particular, with ARIA, services are mapped to processes, and the process workflow is expressed in terms of patterns of events. Accordingly, the user is expressing in a simpler manner which processes to invoke according to events that may occur. The written specification is then given as input to the ARIA compiler (Fig. 4.1 ❸).

The compiler in turn generates the adequate low-level code enabling communications among the assembled processes. In fact, the service orchestration relies on an event-driven, process-based communication paradigm, conceptually similar to what is encountered in traditional POSIX systems (Fig. 4.1 ❹). Each orchestration is mapped to a set of processes, and runs in a sandbox isolated from other instances, enabling multi-tenancy. Hence, several users can deploy different service orchestrations without interferences among each other.

Finally, the MEDLEY platform takes charge transparently, on the behalf of the users, of the interaction with third-party services (Fig. 4.1 ❺) as expected by the users according to their ARIA specifications. Through the use of pre-defined processes that implement the interaction logic with the service providers, MEDLEY supports both the pull and push paradigms. These can be implemented as plugin modules, either using the provided MEDLEY developer API, or using POLLY to create custom change detectors to trigger compositions whenever specific events occur on targeted services.

4.2 Implementation

MEDLEY draws its inspiration from several existing concepts, such as flow-based programming, and process algebras. MEDLEY applies both of these concepts to the particular context of microservice composition. The notion of Flow-Based Programming (FBP) was first introduced by John Paul Morrison in the early 1970s [Morrison, 2010]. FBP introduces the concepts of processes, bounded buffers, information packets, named ports, and separate definition of connections. FBP views an application as a network of asynchronous processes communicating by means of streams of structured data chunks known as *information packets*. Information packets are passed between the inputs and outputs of processes. Each process may have multiple inputs and outputs, and multiple processes may be connected to a specific inport or outport. FBP encourages loose coupling of components, relying on linking black boxes in order to build microservice architectures. This approach is applied in MEDLEY, complemented by an event-driven communication layer.

The implementation of the MEDLEY platform comprises a compiler for the ARIA domain-specific language, a runtime system and a service for integrating third-party service providers. The runtime system relies on Node.js, a JavaScript runtime built on top of Chrome's V8 JavaScript engine which provides an event-driven, non-blocking I/O execution model (Fig. 4.2). Renowned in the web development ecosystem for its performance, efficiency and scalability, Node.js is an ideal target platform for our requirements. From the ARIA specification of an orchestration, the compiler generates JavaScript code that can then be linked with the runtime system. The generated code runs on devices ranging from desktop computers to resource-constrained devices such as home appliances. Excluding third-party dependencies, the runtime system defines various utility functions and amounts to about 1,200 source lines of JavaScript code. The ARIA compiler is around 600 source lines of code. In the remainder of this section, we first describe the main challenges in code generation, then present the runtime system, and finally explain how third-party services are integrated in the MEDLEY platform.

4.2.1 Code generation

The main challenges in generating code from an ARIA specification are the propagation of data throughout subsequent process invocations, and the routing of events through the use of the *publish* / *subscribe* paradigm.

Data propagation

An orchestration typically defines a hierarchy of handlers, the actions inside an `on` clause. Code inside a handler can access not only the data associated to its input event but also its inner events. Figure 4.3 shows an example of orchestration in which a handler manipulates data (line 5) associated to one of its inner events (line 2). Because each

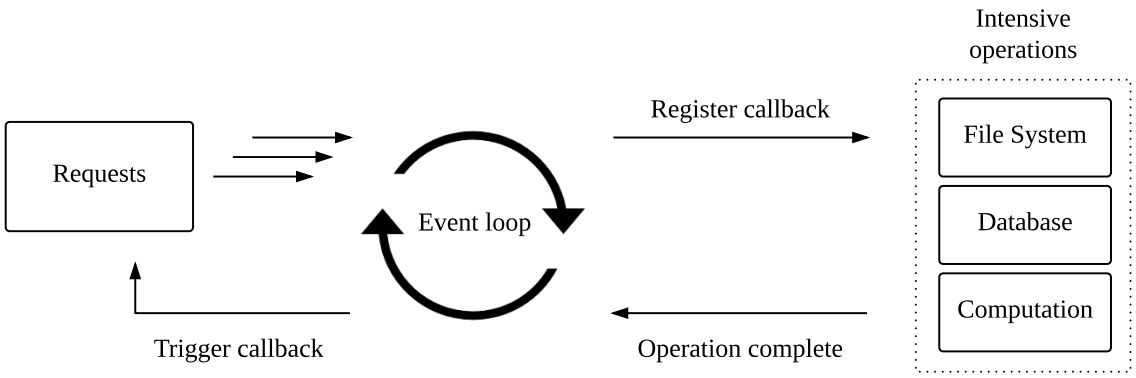


Figure 4.2 – Non-blocking asynchronous execution model of Node.js.

```
1 stream foo = getFoo.invoke();
2 on (foo:out as f) do {
3   stream bar = getBar.invoke();
4   on (bar:out) do {
5     p.invoke({ "id": "{{f.id}}" });
6   }
7 }
```

Figure 4.3 – A hierarchy of nested handlers.

process invocation is asynchronous, data associated to events must be maintained across multiple invocations, resulting into a hierarchy of data. Maintaining data hierarchy can, however, have serious performance penalties in terms of memory usage. Furthermore, propagating the whole payload of an event might not be necessary when only a subset of the data is required at a later stage.

The ARIA compiler implements a backward dataflow analysis to identify data fragments that must be maintained across multiple process invocations. These data fragments are implemented as an environment structure that is added to the event payload. Processes forward this environment from their input channel to their output channel, adding information only when it may be required at later stage. To reduce memory footprint, the environment structure contains only references to data stored inside a global environment maintained by the runtime system. The MEDLEY platform abstracts away this mechanism, as developers do not need to be aware of these details.

Event routing

In MEDLEY, each process has its own input channel for listening to events and output channel for publishing events. Events associated to a process are isolated in the namespace of the process, preventing them from interfering with other processes. To implement the logic described in the ARIA specification, the compiler generates a set of rewrite rules.

$$\frac{e = \langle l, d, \delta \rangle \qquad e \vDash p.\texttt{invoke}(j)}{e \Rightarrow \langle p_{in}, j, \delta \cup \{(l,d)\} \rangle} \tag{4.1}$$

$$\frac{e = \langle l, d, \delta \rangle \qquad e \vDash \texttt{stream}\ s = p.\texttt{invoke}(j)}{e \Rightarrow \langle p_{in}, j, \delta \cup \{(l,d)\} \rangle \qquad p_{out} = \langle l', d', \delta' \rangle \qquad p_{out} \Rightarrow \langle s, d', \delta' \rangle} \tag{4.2}$$

$$\frac{e = \langle l, d, \delta \rangle \qquad \vDash \texttt{on}\ (e)\ \texttt{do}\ \{stmt_1; \ldots; stmt_n\}}{e \vDash action_1 \quad \ldots \quad e \vDash action_n} \tag{4.3}$$

$$\frac{e_1 = \langle l_1, d_1, \delta_1 \rangle \qquad e_2 = \langle l_2, d_2, \delta_2 \rangle \qquad \vDash \texttt{on}\ (e_1\ \texttt{and}\ e_2)\ \texttt{do}\ \{stmt_1; \ldots; stmt_n\}}{\begin{array}{c} e_1 \Rightarrow \langle and_{in}, \{(l_1, d_1)\}, \delta_1 \rangle \qquad e_2 \Rightarrow \langle and_{in}, \{(l_2, d_2)\}, \delta_2 \rangle \\ and_{out} \vDash stmt_1 \quad \ldots \quad and_{out} \vDash stmt_n \end{array}} \tag{4.4}$$

Figure 4.4 – Rewrite rules for event routing.

Rewrite rules are used to intercept events, rename them, and publish them under a new event name, in order to dispatch them to the appropriate recipient processes. Rewrite rules are described as inference rules with a sequence of premises above a horizontal bar and a judgment below the bar (see Fig. 4.4).

An event e is described as $\langle l, d, \delta \rangle$, where l is the label name of the event, d the data associated to it, and δ the environment structure of the call hierarchy. A rewrite rule of the form $e_1 \Rightarrow e_2$ means that once the event e_1 occurs, the runtime system raises the event e_2. A judgment of the form $e \vDash stmt$ means that the runtime systems interprets the statement $stmt$ when the event e occurs. In other words, $stmt$ is the callback associated to e. The first and second rules are for invoking a process p. In that case, we rewrite the event e that triggers the invocation of p as p_{in}, the input event of p. The third rule shows how all instructions defined in the body of the handler are executed asynchronously. The fourth rule shows how MEDLEY implements the `and` operator by rewriting each event into the input event of the `and` process. This process is provided as a built-in process. When it receives both the events $\langle and_{in}, \{(l_1, d_1)\}, \delta_1 \rangle$ and $\langle and_{in}, \{(l_2, d_2)\}, \delta_2 \rangle$ on its input channel, it generates the event $\langle and_{out}\{(l_1, d_1), (l_2, d_2)\}, \delta_1 \cap \delta_2 \rangle$ on its output channel. This rule is generalizable for the conjunction of n events.

4.2.2 Runtime system

The runtime system relies on Node.js as the underlying execution environment. Once a composition is specified and compiled, the generated code is deployed onto the platform for execution.

Event-driven messaging model

At runtime, an ARIA compositions is translated into a set of processes and rewrite rules. The runtime system manages the lifecycle of processes by initializing, starting, stopping and destroying them as necessary. When initialized, a process p is subscribed to its input channel, listening to events of type p_{in}. Whenever it receives such an event, the process is invoked, producing on its output stream events of type p_{out} for successful executions, or events of type p_{err} when errors are encountered. When a process finishes executing its logic, it emits an event of type p_{end} to signal the end of the processing, thus closing its output stream and releasing acquired resources. To enable the routing of events to the appropriate processes, the system generates a set of rewrite rules to dynamically rename events on the fly, mapping them to the appropriate processes as defined by the composition specification. Rewrite rules are discussed at greater length in Section 4.2.1. Furthermore, the runtime system encapsulates each composition in a scoped environment by assigning it a unique namespace. Therefore, events generated within a composition are restricted to their composition scope and cannot leak over to other compositions, thus enabling multi-tenant concurrency.

Authenticating HTTP requests

Nowadays, most third-party services require some form of client authentication in order to allow the interaction with their web APIs. Such mechanisms ensure that the client is authorized to perform the requested operations. Our current implementation supports a wide range of client authentication methods, ranging from HTTP Basic Authentication [Franks et al., 1999] and API keys [Farrell, 2009], to OAuth protocols [Hammer-Lahav, 2010; Hardt, 2012]. To handle these authentication mechanisms, MEDLEY provides a dedicated user interface through which users can authorize third-party services by providing the corresponding credentials and a textual label to reference them. When editing an ARIA composition, the `process.init` method can be used to specify the user credentials in order to authenticate outgoing requests to third-party services when authentication is required. A process cannot be started unless all required parameters and credentials have been correctly set.

Error handling

During its lifecycle, a composition may encounter several kinds of errors. A process may emit an error on its output channel (events of type `err`) based on its internal implementation. An error may indicate that a request to a third-party service has failed, that authentication has failed or any other service specific errors. These errors are reported as events and thus are accessible at the language level. Therefore, users can describe in their orchestration their own error handling policies. In addition, the runtime system handles errors such as network failures. In that case, it rolls back the failed process and retries

the failed request again later, increasing the time interval between each subsequent retry. When too many errors are raised by a composition, the system may decide to kill the running instance and release corresponding resources. Furthermore, the system enforces a timeout at the process level and at the composition level, in order to prevent the execution from hanging indefinitely.

4.2.3 Integrating third-party services

Third-party services are integrated into the MEDLEY platform through the implementation of processes. The developer responsible for this task is called a process provider.

Defining processes. Since the majority of exisiting third-party service providers rarely publish any form of formal service interface description (such as WSDL or OpenAPI descriptions), the process provider needs to bridge this gap by providing the necessary information to describe the wrapped service. Typically, in MEDLEY this consists in specifying various details about the process interface in a manifest file, facilitating its discovery and usage across the platform. The manifest file contains metadata about the process (name, human-readable description, resources, classification, etc.), and includes a process interface. The interface is described using the JSON Schema standard. It gives typing information about the process input and output data, indicating which keys are required or optional, which values are allowed, etc.

Implementing processes. After defining its interface, the process provider implements the process using one of two methods. The first method consists in implementing the process as a Node.js module. This can be as simple as specifying the HTTP request to the targeted service. Various helper methods from the MEDLEY developer API are passed to the module through depencency injection. The second method consists in using the POLLY language to specify *trigger* processes. These processes are typically used to monitor changes on third-party services, and trigger the execution of a composition whenever a change is detected.

Publishing processes. Once fully implemented and defined, processes can be published on the MEDLEY process repository, making them available to use by the platform users. A discovery service enables looking up processes according to user requirements. Furthermore, the process repository also supports versioning for published processes, as it is necessary to handle third-party API evolution in a sane way.

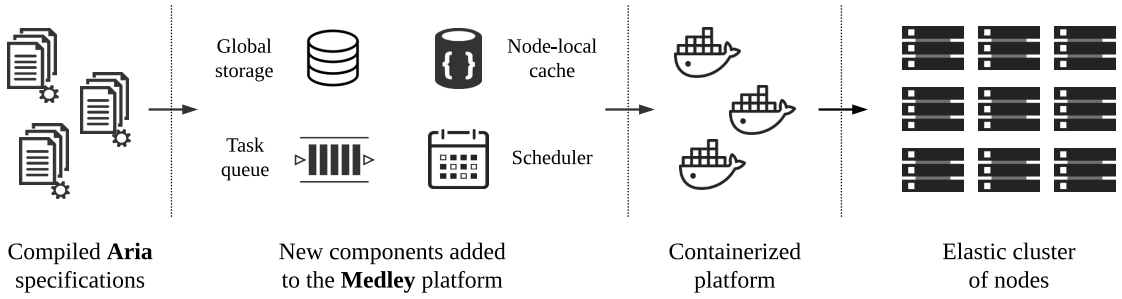


Figure 4.5 – Overview of the architectural elements to scale the MEDLEY platform.

4.3 Towards a scalable architecture

In Section 3.3, we introduced ARIA, a language-based approach to raise the level of abstraction required to express an orchestration of web services, and presented the implementation details on how ARIA compositions are executed on the MEDLEY platform in Section 4.1. However, knowing that CPRODIRECT plans on commercializing the platform, it is crucial to consider the performance and scalability of the platform in order to ensure the reliability of the product.

In this section, we describe how we improve the prototype implementation of MEDLEY and turn it into a reliable commercial product by introducing new architectural elements, supporting horizontal scaling across an elastic cluster of nodes. We propose an approach to overcome API rate limit rules of third-party services, in order to scale the number of executed composite services linearly with the number of nodes of the cluster. An evaluation is presented in Chapter 5, where we compare the performance of our approach against existing scheduling strategies.

4.3.1 Scalability challenges

To support horizontal scaling, a distributed approach must be considered in order to efficiently spread the increasing workload. In the context of MEDLEY, we consider the execution of a service composition as an individual job. Hence, scaling MEDLEY requires dispatching jobs across a cluster of nodes running the MEDLEY platform in an efficient way. To achieve this, a suitable scheduler is required to efficiently load balance the incoming workload among the multiple instances of the MEDLEY platform deployed on the cluster nodes. Furthermore, the expected scheduler must be able to perform a fine-grained dispatching policy according to several criterias in order to optimize the performance. First, it must take into account the underlying resource usage of each node. Second, it should also consider the resulting cache affinity coming from the execution of compositions to avoid reaching the API rate limits of the related third-party services. In the remainder of

this section, we introduce a refinement of the architecture of the MEDLEY platform and its new underlying elements to provide such features.

The main challenge is designing a scalable architecture capable of supporting an increasing number of clients (thus, an increasing number of executions) while minimizing the usage of third-party services in order to stay under the API rate limits for as long as possible. Figure 4.5 highlights the newly-added architectural elements to support horizontal scaling, and how they fit with our already existing MEDLEY platform. The main new components are the global storage, the task queue, the scheduler, and local caches for each node.

4.3.2 Approach

From single node to a distributed architecture

The initial MEDLEY platform, which includes components such as a runtime system, core services, and a set of service providers, is containerized using Docker [Merkel, 2014] to ensure sandboxing at the operating system level. This facilitates both deployment and administration of the platform on a number of system architectures. Following the microservices paradigm, the MEDLEY platform that runs inside a container, called a worker, processes a single composition at a time according to an ARIA specification. As such, this enables fair resource usage across compositions, while also preventing misbehaving compositions from impacting other instances running on the same node. Each node of the cluster can launch several workers depending on its hardware capabilities (CPU and memory). Additional nodes and/or workers can be dynamically provisionned to meet the increasing load.

Precompiling MEDLEY specifications

Each MEDLEY specification $\mathcal{S}$ is compiled to produce executable code. Compiled MEDLEY specifications $C(\mathcal{S})$ of $\mathcal{S}$ are saved in a global storage using the hash $H(\mathcal{S})$ of the specification as an identifier. Each specification $\mathcal{S}$ is hence uniquely identified by an identifier $\mathcal{S}_{id}$, and is further associated with metadata that defines user credentials and the frequency at which $\mathcal{S}$ needs to be executed based on the subscription plan of the user. Based on that information, a new job is created every time $\mathcal{S}$ needs to be executed. The resulting job defined as $\langle \mathcal{S}_{id}, H(\mathcal{S}) \rangle$ is then submitted to the task queue.

To avoid fetching every time the compiled version of a composition to be executed by a worker, each node of the cluster embeds a local in-memory cache. In case of a cache miss, the compiled code corresponding to the job is retrieved from the global storage and saved in the cache, resulting in potential eviction of previous lines. To avoid waste of resources, cache entries have time-to-live delays and are automatically evicted if unused for a given period.

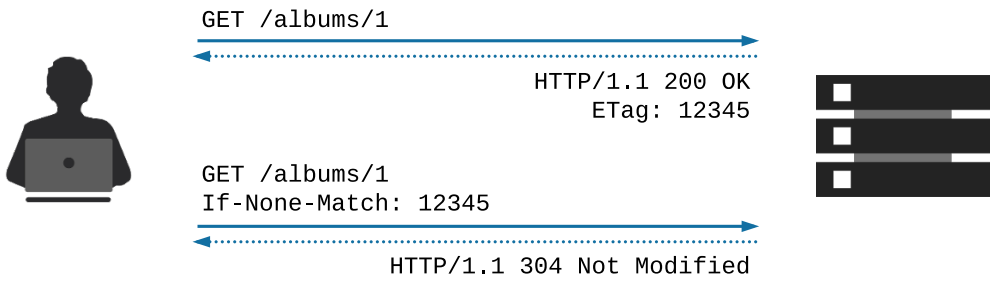


Figure 4.6 – Conditional request using the `ETag` and `If-None-Match` HTTP headers.

Job scheduling & processing

Whenever a composition is planned for execution, a corresponding job is submitted to the task queue. The scheduler consumes jobs from the task queue, and schedules them to available nodes. Using a task queue upstream allows scaling the scheduler by spawning several instances feeding off the same queue, thus distributing the load over several instances and avoiding having a single point of failure. When the execution of a composition is completed, the container is shutdown, and resources of the host node are released for future use. However, if the execution fails due to a runtime error (e.g. an internal error or an error returned by a third-party service), then the job is pushed back into the queue and retried again at a later point in time, using an exponential backoff policy. This allows to account for rate limits, which can be handled by retrying later in this case.

Caching HTTP responses

Similarly to the pre-compiled specifications, each node uses a local cache for third-party services. Whenever a worker requests a resource from a third-party service, it first checks if it is already present in the local cache, in which case its value is immediately returned. Otherwise an outgoing `GET` request is issued to the third-party service, and its response body is cached according to the response headers. Namely, the HTTP standard defines clear semantics of caching mechanisms through the use of the `Cache-Control` and `Expires` headers. These headers allow specifying the caching strategy for a given resource as well as their expiration date, beyond which the resource is considered to be stale. For instance, `Cache-Control: public, max-age=600` indicates that a resource can be cached for 10 minutes by any (public or private) cache.

Furthermore, performing conditional requests improve the efficiency of the caches while also reducing the bandwidth usage. For instance, the value of the `ETag` response header (which represents the fingerprint of a given resource) is used in subsequent requests under the `If-None-Match` request header to conditionally fetch a resource if the

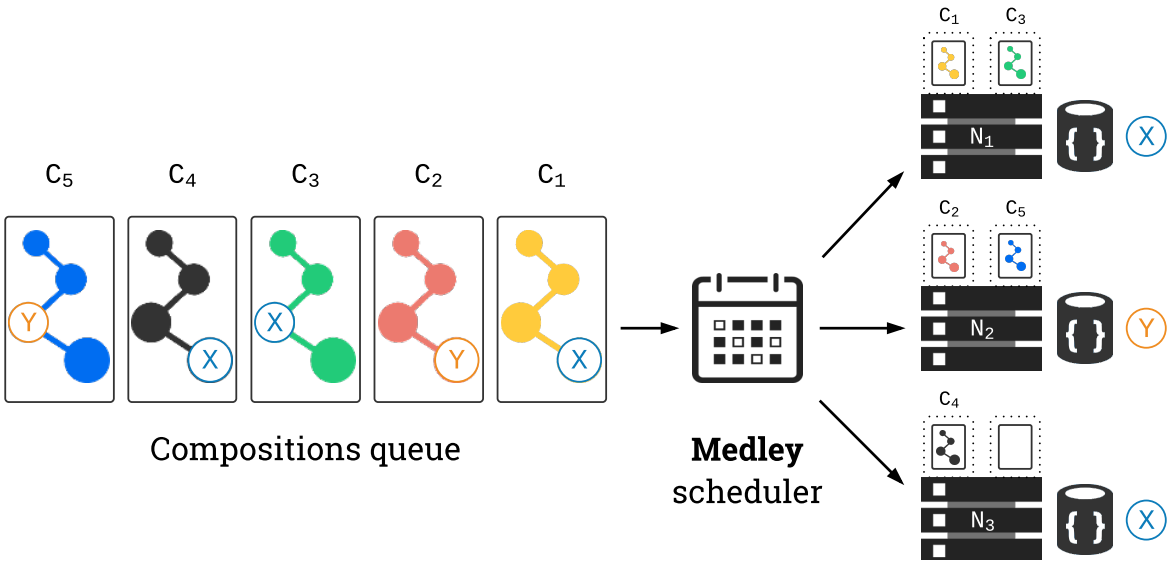


Figure 4.7 – Cache-aware scheduling of jobs across a set of cluster nodes. `X` and `Y` denote cacheable resources.

resource has been modified between polls. Otherwise, the server returns the status code `HTTP 304 Not Modified` to indicate that the resource has not been modified since, and that using the cached value is safe (see Fig. 4.6). Likewise, the same principle applies with the `Last-Modified` response header and the `If-Modified-Since` request header, but with timestamps instead of hash values.

Using this kind of information enables us to safely cache and manage responses from the queried third-party services. The advantages of cache hits are twofold: (i) the response is returned an order of magnitude faster than performing a network roundtrip to fetch it since it is locally available, thus incurring less operational costs, and (ii) it avoids consuming from the allocated API rate limit quotas, thus supporting more executions.

However, third-party services do not necessarily support caching. To overcome this issue, we extend our approach to enable the platform owner to override the service provider component. Such components are responsible for communicating with specific third-party services and act as thin wrappers to enable their integration with the MEDLEY platform. Depending to which extent an information needs to be accurate, the service provider component can override the headers of a response to make it cacheable for a given period of time. For example, if many users (and thus compositions) use a weather service to fetch the temperature of the same city at almost the same time, responses to that service can be safely cached for a given amount of time.

$$\Theta(x,n) = \begin{cases} 1 & \text{if } x \text{ is cached locally on node } n \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

$$\Theta_{src}(C,n) = \alpha \cdot \Theta(C,n) \tag{4.6}$$

$$\Theta_{srv}(C,n) = \beta \cdot \sum_{s=1}^{|services(C)|} \gamma \cdot \Theta(s,n) \tag{4.7}$$

$$score(C,n) = f(\Theta_{src}(C,n),\ \Theta_{srv}(C,n),\ res(n)) \tag{4.8}$$

Figure 4.8 – Job placement heuristics.

Optimizing cache affinity

The MEDLEY scheduler is responsible for efficiently load balancing jobs among nodes of the cluster. The purpose of job placement is to optimize cache affinity while taking into account the underlying resource usage of each node. More specifically, to do that, we rely on a heuristics-based approach which calculates, for a given job, the score of each node. The highest ranking node is then selected and a worker is created on that node to execute the job. To avoid launching too many workers on nodes of the cluster, thus degrading the overall performance, the scheduler can postpone the processing of jobs in the task queue. When the size of the task queue increases too much, the system may decide to dynamically extend the size of the cluster by provisioning new nodes. Figure 4.7 gives an illustration of cache-aware scheduling across a cluster of 3 nodes.

Our scoring function depends on several criteria (see Fig. 4.8). First, we define a helper function Θ that checks if a given resource is locally cached (Equation 4.5). The functions in Equations 4.6 and 4.7 calculate a partial score reflecting if a composition C is cached locally on node n, and a partial score for the number of services used in composition C that are cached on node n, respectively. A node ranks higher if it has cache entries for the compiled code of the composition as well as entries for the third-party services used by that composition. The weight parameters α, β and γ enable fine-tuning the equations according to the cost or importance of the variables. For instance, we can assign a higher γ value for services that have more restricted rate limit quotas than others. Finally, Equation 4.8 shows how the final score depends on the partial scores calculated previously, as well as the resources of node n: CPU usage, available memory, number of jobs in progress, etc. The score of a node increases when available resources increase. Note that the scheduler also does a health check of each node of the cluster to prevent scheduling a job on an unavailable node.

A walkthrough example

To better illustrate this aspect, consider the setup presented in Fig. 4.7 as an example. Initially, all nodes (N_1, N_2 and N_3) are idle, and their corresponding caches are empty. The composition queue contains five compositions (C_1 to C_5), each them requiring either the cacheable resources X or Y.

Step 1. First, the scheduler pulls C_1 from the queue, and assigns it to the highest ranking node. Since initially, all nodes have an equal score, any node can be chosen (in this case, N_1 was selected). During its execution, composition C_1 requests the resource X, which is subsequently cached on the same node.

Step 2. Next, C_2 is considered. The highest ranking nodes for C_2 are nodes N_2 and N_3. In this case, N_2 was selected. Likewise, composition C_2 is executed, and resource Y is subsequently cached locally.

Step 3. The scheduler considers now the placement of composition C_3. Seeing as this composition requires the resource X, which is already cached on node N_1, the score of this node increases considerably, leading to the placement of C_3 on node N_1. Leveraging the availability of the cached resource X improves overall performance by reducing the overall number of outgoing requests, and does not consume from the corresponding alloted API requests quota.

Step 4. Composition C_4 is now considered. Although node N_1 would be ideal in terms of cache affinity, it however can no longer process more compositions due to lack of resources. Thus, the highest ranking node in this case is node N_3.

Step 5. Finally, the scheduler considers composition C_5. Similarly to step 3, this composition requires the resource Y, which is already cached on node N_2. Following our heuristics for optimizing cache affinity, the highest ranking node in this case is N_2.

4.4 Summary

We presented in this chapter the implementation details of the MEDLEY platform. We described the compiler internals, and showed how the underlying runtime system relies on a lightweight, event-driven model to support the execution of MEDLEY compositions. Furthermore, we identified the challenges in scaling such a platform, and proposed a novel approach to efficiently distribute the load across a cluster of nodes. In the following Chapter 5, we present a thorough evaluation of the domain-specific languages presented in

Chapter 3, followed by a performance evaluation of the MEDLEY runtime system to evaluate its performance and scalability.

CHAPTER

5 Evaluation and analysis

In this chapter, we present several evaluations to assess the proposed contributions. First, we show the applicability of POLLY by using it to automatically generate a number of change detectors for widely used web services such as Twitter, Facebook, and GitHub. We demonstrate that POLLY's code is more concise that a manual implementation, and that it outperforms a state-of-the-art, off-the-shelf differencing technique. Second, we propose a comparative study of the supported language features of ARIA and the abstractions provided compared to existing approaches. We also evaluate the runtime performance of the code generated by compiling ARIA specifications. Using various compositions involving a large number of services, we show how ARIA consumes a reasonably low amount of resources, both on a standard server and on an embedded device. Finally, we show how the MEDLEY platform scales well in a distributed context, while also taking into consideration the API rate limits of third-party services. We demonstrate how our approach outperforms existing solutions.

Contents

5.1 Evaluating the POLLY language

We evaluate our approach using six scenarios provided by our industrial partner CPRODIRECT. We first compare the level of abstractions provided by POLLY (in terms of verbosity) compared to its handwritten counterpart. We then assess some runtime metrics (such as the differencing time, output size and maximum memory usage) of our solution compared to a state-of-the-art differencing tool.

5.1.1 Test scenarios

Our industrial partner CPRODIRECT has defined the six following scenarios to be used in our evaluation. They illustrate the diversity of possible use cases ranging from being notified about new objects to changes of attributes values or order in a ranking.

— **ElasticSearch (ES):** Developer Alice uses an instance of ElasticSearch as a search engine for her e-commerce platform, and wants to be notified when the top 5 best-selling products change in ranking order.

— **Facebook (FB):** Developer Alice wants to monitor a Facebook album where her friends Dan and Dave are participating. Alice would like to be notified only about pictures where Dan and Dave are tagged together.

— **GitHub (GH):** Developer Alice is interested in monitoring GitHub for new projects written in the Go language with over 2,000 stars.

— **Stack Overflow (SO):** Developer Alice wants to monitor StackOverflow for new JavaScript questions where there is an active bounty of over 100 reputation points.

— **Transport for London (TL):** Developer Alice wants to be notified whenever the status of the Victoria subway line changes (e.g. from healthy to faulty).

— **Twitter (TW):** Developer Alice wants to be notified whenever the official *Bordeaux* account has new followers on Twitter.

5.1.2 Language verbosity

Experimental setup. All scenarios described in Section 5.1.1 have been implemented twice by the main contributor of POLLY: once using the JavaScript language on top of the Node.js platform, and once using our domain-specific language POLLY. Note that the JavaScript version was implemented before any research work was done on POLLY, in order to avoid any bias, and to serve as a reference point in subsequent evaluations.

Experimental protocol. Figure 5.1 shows the number of lexical tokens used in the Node.js version versus the POLLY version. One can notice that POLLY results in a much smaller program, ranging from 5.5 to 8 times smaller. Furthermore, the figure shows the distribution

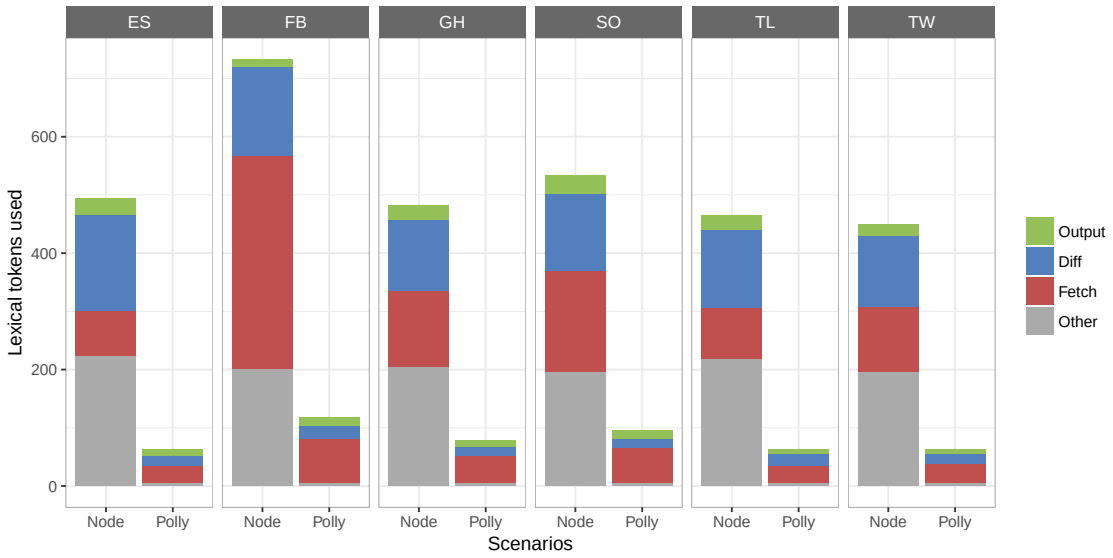


Figure 5.1 – Lexical tokens used to specify each scenario, using Node.js code *vs.* POLLY.

of tokens across different categories (*fetch*, *diff* and *output*). Other tokens that are not directly related to these (such as module imports and configuration) are assigned to the *other* category. First, we notice that the Node.js implementation requires a lot more boilerplate code than POLLY, with around 200 tokens in the *other* category, compared to 5 for POLLY. Second, we notice that the *output* construction requires more or less the same number of tokens for both approaches, while it requires significantly less tokens for the *fetch* and *diff* categories using the POLLY approach.

5.1.3 Performance metrics

Since one of the main benefits of using our approach is to be able to perform a custom differencing based upon domain knowledge of the data returned by the REST APIs, we wanted to evaluate in greater details the advantages of using such a strategy. We compare in this experiment the performance of POLLY against a state-of-the-art generic differencing technique for JSON documents (JDR). We selected JDR as a candidate since prior benchmarks show it outperforms all other JavaScript differencing libraries [Cao et al., 2016].

Experimental setup. Since we are only focusing on the performance of the differencing and output construction stages for this benchmark, we can prefetch all required resources for better reproducibility. We thus proceed to collecting real data from the six service providers presented above. This is achieved by polling the services for the required resources over a period of 48 hours with an interval of 5 minutes, yielding 576 snapshots per service. We then serve this collected data through a mock server in the following exper-

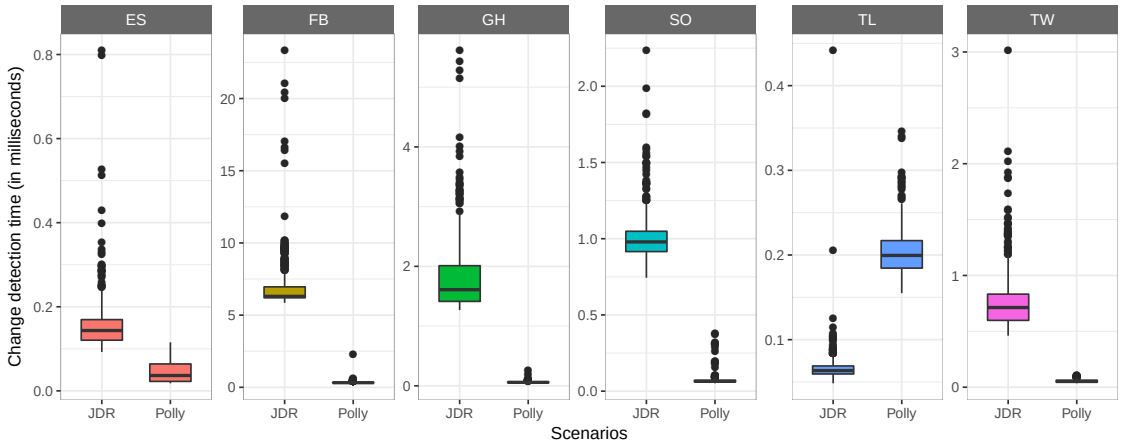


Figure 5.2 – Change detection time using JDR *vs.* POLLY.

iments. All experiments were performed on a single machine powered by an Intel Core i7-6500U CPU @ 2.50 GHz x 4 and 8 GB of memory.

Experimental protocol. We designed an experiment which consists in running each scenario 576 times (once for each snapshot) using JDR and POLLY as change detection methods. At each step, we measure the differencing time as well as the output size. This process is repeated for 10 iterations for better precision. Throughout the process, the memory usage is monitored in order to register the peak memory consumption. The results of this experiment are shown in Fig. 5.2, Fig. 5.3 and Fig. 5.4. One can notice that the POLLY approach produces lower differencing times and output sizes compared to the JDR approach, apart from the output size for the Facebook (FB) scenario, where the output size is equal to 0 for every polling step for both approaches. This is because no modifications occurred during the monitoring period. The difference in output sizes is explained by the fact that JDR produces a JSON Patch [Bryan and Nottingham, 2013] (an intermediary document expressing a sequence of operations to apply to a JSON document in order to obtain the final outcome), whereas POLLY directly produces the minimal set of required data as specified in the DSL, which generally tends to be much smaller in size. Furthermore, we can see in Fig. 5.4 that the maximum memory usage for POLLY is always lesser than for JDR. This is because POLLY loads smaller objects in memory during the differencing stage, thanks to the templating directives of the DSL.

Statistical testing. To have a finer-grained analysis of these results, we subject our results to a statistical testing. Our three null hypotheses are that:

— H_0^1 output size is the same for POLLY and JDR.

— H_0^2 differencing time is the same for POLLY and JDR.

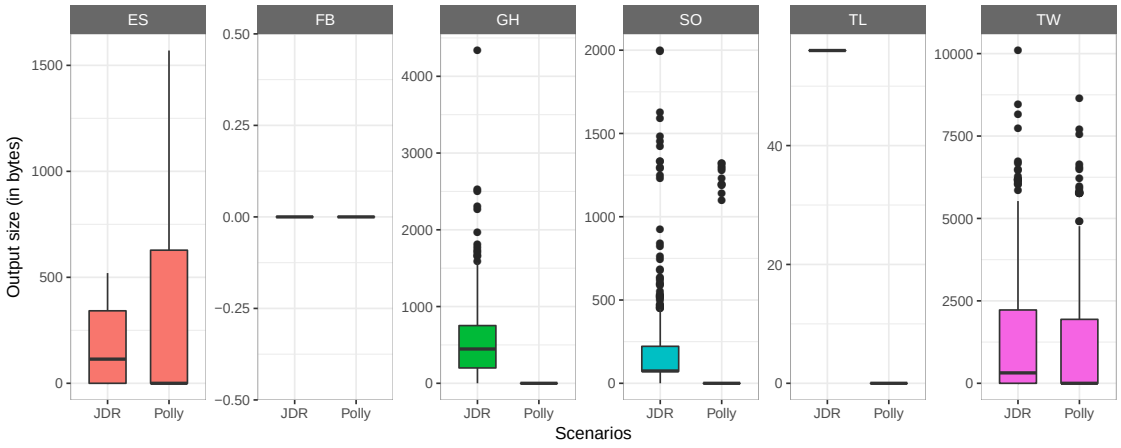


Figure 5.3 – Output sizes using JDR *vs.* POLLY.

— H_0^3 maximum memory usage is the same for POLLY and JDR.

Our three alternative hypotheses are:

— H_a^1 output size is lesser for POLLY than JDR.

— H_a^2 differencing time is lesser for POLLY than JDR.

— H_a^3 maximum memory usage is lesser for POLLY than JDR.

To test these three hypotheses, we used a one-tailed paired Wilcoxon rank test, since it bears no assumptions on the underlying distribution of the dataset values. To assess the magnitude of the difference between differencing time, output size, and maximum memory usage between the two approaches, we use Cohen's *d* and report its corresponding level on Cohen's standard scale. The results of this statistical testing are shown in Table 5.1.

One can notice that most tests are significant (*p-value* under the 0.05 threshold), meaning that POLLY produces significantly smaller outputs in a significantly reduced time compared to the JDR generic differencing approach. The only non-significant test is for the

Table 5.1 – P-values of our statistical testing and size effect.

	Detection time		Output size		Max memory	
Scenario	P-value	Effect size (lvl)	P-value	Effect size (lvl)	P-value	Effect size (lvl)
ES	5.2403e-96	2.0909 (large)	4.4477e-42	0.6854 (med)	8.0638e-31	0.5821 (med)
FB	5.2548e-96	3.7704 (large)	1.0000e+00	NaN (NA)	4.3350e-06	0.1922 (negl)
GH	5.2550e-96	2.9894 (large)	1.3620e-84	1.1863 (large)	1.1887e-63	0.8794 (large)
SO	5.2543e-96	6.1502 (large)	1.0484e-74	0.7705 (med)	9.8855e-16	0.3372 (small)
TL	1.0000e+00	-4.8853 (large)	6.3619e-99	88.6262 (large)	2.2512e-02	0.0908 (negl)
TW	5.2547e-96	2.8470 (large)	2.4653e-72	0.8081 (large)	3.9153e-39	0.6092 (med)

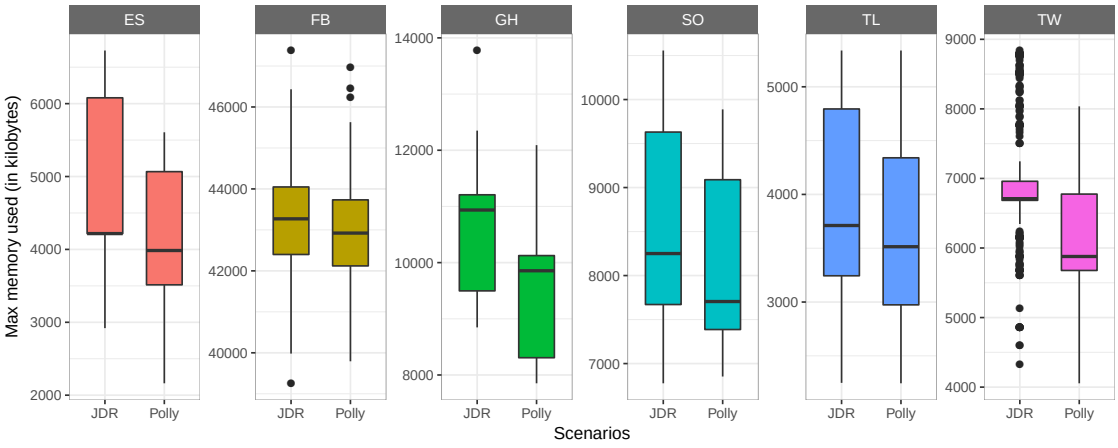


Figure 5.4 – Maximum memory usage using JDR *vs.* POLLY.

output size of the FB scenario. This is because in this scenario the output size is equal to 0 for every polling step for both approaches.

For the magnitude of the difference, the values range from medium to large, large being by far the most common value (10 times out of 17 values), followed by medium (4 times), small (1 time) and negligible (2 times). This means that POLLY results in a highly improved outcome in terms of output size, differencing time and memory usage compared to the JDR approach.

5.2 Evaluating the ARIA language

In this section, we propose a number of evaluations to assess the benefits of our approach. First, we present a comparison of the language features provided by ARIA and other existing languages and solutions. We rely on well-established benchmarks to show how ARIA is more expressive than the other alternatives. Then, we conduct a series of experiments to evaluate the performance of the generated code from ARIA specifications, and show its efficiency when executed on different setups.

5.2.1 Language expressivity

Experimental protocol

To assess our approach in providing a simple yet highly expressive language for service composition, we conduct a comparative study of the features supported by ARIA compared to Bite [Rosenberg et al., 2008], S [Bonetta et al., 2012] and the WS-BPEL standard [Andrews et al., 2003]. We select these solutions because they address the problem of composing web services and provide a language to describe such compositions. We rely on the work of Sheng et al. [Sheng et al., 2014] to identify the following features:

— *Dynamic typing*: the ability to manipulate arbitrarily-typed data structures.

— *Dynamic service selection*: the ability to select and bind services at runtime.

— *Exception handling*: the ability to handle and respond to runtime errors.

— *Hybrid service support*: the ability to compose services of different types (REST, SOAP, etc.).

— *Language extensibility*: the ability to extend the language and provide new features.

— *Scoping*: the ability to define and use nested blocks and localized variables.

Table 5.2 – Comparison of language features in ARIA *vs.* Bite, S and WS-BPEL.

	ARIA	Bite	S	WS-BPEL
Dynamic typing	+	+	+	-
Dynamic service selection	+	-	-	-
Exception handling	+	~	~	+
Hybrid service support	+	-	-	~
Language extensibility	+	+	-	-
Scoping	+	-	+	+

(+) Supported, (-) Not supported, (~) Partial support.

Results

Table 5.2 summarizes the results of our comparative study.

— All approaches support dynamic data typing except for BPEL, where data types are defined by their corresponding WSDL interface.

— Furthermore, even though all solutions enable static binding of services, ARIA also provides a construct to handle pools of services at runtime, enabling dynamic binding based on user-defined strategies.

— All four solutions also support handling runtime exceptions, although at different levels. For instance, Bite enables defining exception handlers at the activity and composition levels, while S just relies on standard error handlers provided by the JavaScript language. On the other hand, ARIA enables reacting to error events from the output streams of the invoked processes.

— As for the supported types of web services, they all enable composing RESTful services except BPEL, even though recent works aim to address this aspect by proposing extensions to BPEL. However, in practice, these extensions have limited support for most major enterprise BPEL engines. Moreover, since services are wrapped and exposed as processes in ARIA, we can easily integrate other types of web services such as SOAP. Since the adaptation is handled at the process level (by the process provider), it is transparent at the language level, enabling the composition of heterogeneous services.

— Regarding language extensibility, ARIA can be easily extended by implementing new processes, whereas the same can be achieved in Bite by implementing new activity types, allowing further customization of these languages. This aspect is not covered in S and BPEL.

— Table 5.2 also shows that scoping is supported by all solutions except Bite, since it relies on a lightweight composition model.

5.2.2 Performance metrics

To assess the runtime behaviour of the code generated from compiling ARIA specifications, we perform a series of experiments to evaluate the resource usage across different hardware architectures.

Experimental protocol

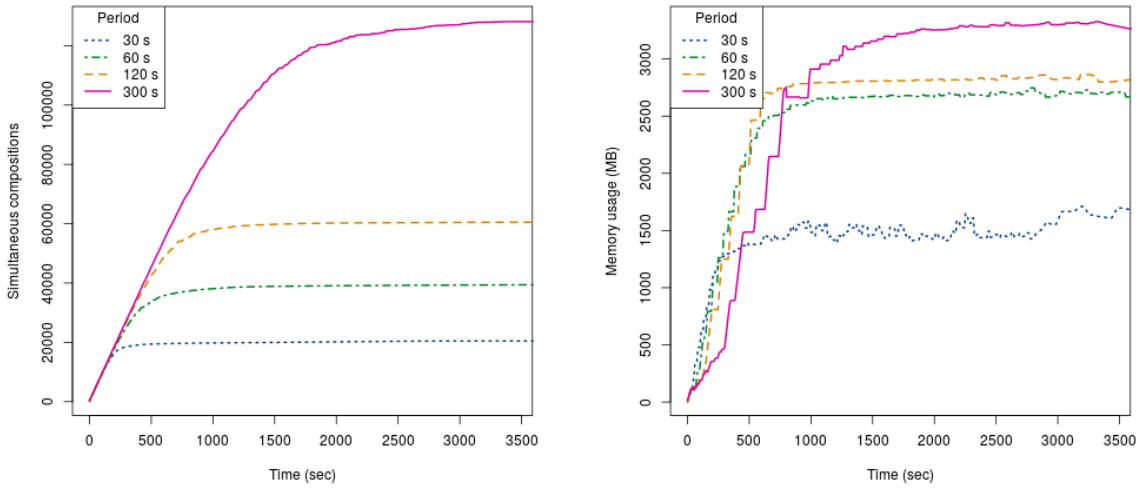
Our benchmarks measure the resource usage of ARIA compositions when gradually increasing the number of simultaneous compositions. Monitoring memory footprint is performed using Node's builtin method `process.memoryUsage()` [1]. This method returns various information about the memory consumption of the Node process including the resident set size, which is the portion of the process's memory held in RAM.

We perform a staged rollout by instanciating and starting a new composition every 10 milliseconds, and collect a snapshot of memory usage every second. The period used in our experiments vary from 30 seconds to 5 minutes (which is relatively short compared to existing commercial solutions, where the fastest cycles are of 5 minutes). A small period increases responsiveness but requires much more resources as the composition needs to be executed more often. This protocol reflects the typical workload that is expected for recurrent compositions.

```
composition {
  process getQuote = require("Mock/GetQuote");
  process sendSms = require("Mock/SendSms");
  // ...
  stream quote = getQuote.invoke({ "symbol": "MSFT" });
  on (quote:out as q) do {
    if (q.value > 100) {
      sendSms.invoke({
        "text": "Current price: {{q.value}}",
        "number": "+33601234567"
      });
    }
  }
}
```

Figure 5.5 – ARIA specification of the stock exchange composition. This composition sends a notification by SMS to the user whenever the value of MSFT shares goes over 100 USD.

1. https://nodejs.org/api/process.html#process_process_memoryusage



(a) Simultaneous compositions.

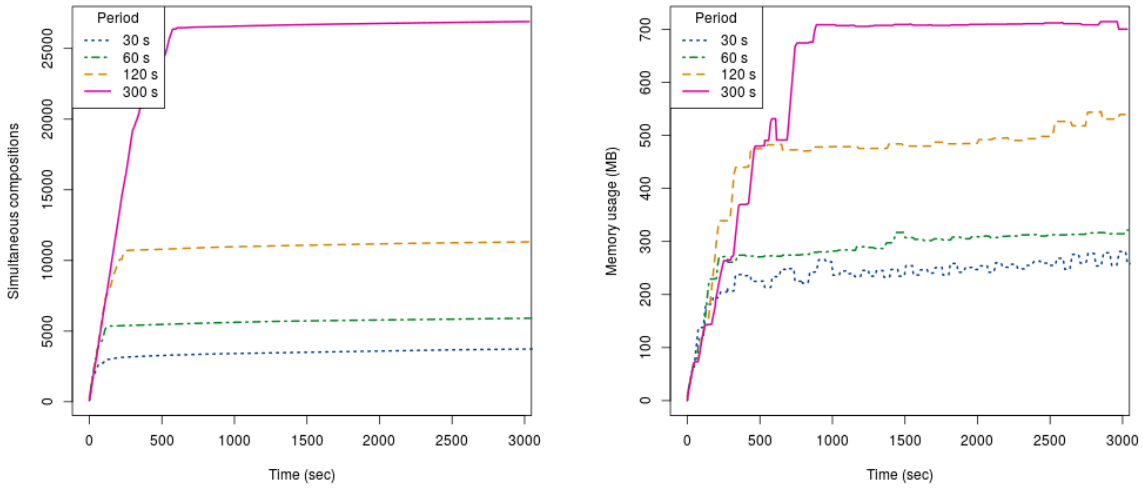
(b) Dynamic memory consumption (MB).

Figure 5.6 – Benchmark results on a server.

Experimental setup

The ARIA specification used for our experiments is depicted in Fig. 5.5. It consists in periodically polling a stock exchange service for a quote, and notifying the user by SMS if the value of the stock quote is above 100 USD. The period corresponds to the time elapsed between two successive executions of a composition. To measure the intrinsic scalability of our implementation, the processes used in our experiments do not actually communicate with third-party services. Instead, we use a mock server to simulate real-world latency by defining a randomized delay for response times between 50 and 100 milliseconds. Similarly, we mock the behavior of the stock exchange service. The returned value is randomized and varies between 80 and 120 USD.

We run our experiments on two different kinds of hardware platforms, from embedded devices to mainstream servers. The server we use is powered by 2 quadcore AMD Opteron 4386 CPUs at 3 GHz and 16 GB of memory. We configure our runtime system to use a pool of 7 working threads, and one thread for the main process. Therefore, we allocate one thread on each physical core of the server. We increase the memory limit of our underlying execution engine to 4 GB which is its current maximum on 64-bit systems. As an embedded system candidate, we use the Raspberry Pi 2 model B with 1 quadcore BCM2836 CPU and 1 GB of memory. We configure our runtime system to use a total of 4 threads, mapping each of them on a physical core. We raise the memory limit to 1 GB, which is the maximum of memory available on this device.



(a) Simultaneous compositions.

(b) Dynamic memory consumption (MB).

Figure 5.7 – Benchmark results on an embedded device.

Results

Performance results on the server are shown in Fig. 5.6 while those for the embedded device are shown in Fig. 5.7. On the server, the total number of simultaneous compositions varies from at least 22,000 with a period of 30 seconds to up to 125,000 with a period of 5 minutes. Similarly, the Raspberry Pi 2 enables at least 4,000 simultaneous compositions with a period of 30 seconds to up to 27,000 with a period of 5 minutes. When the period is too small or the number of simultaneous compositions is too high, the event queue of the runtime (Node.js) becomes full and no composition can be instantiated anymore. As illustrated in Fig. 5.6b and Fig. 5.7b, the memory consumption of ARIA follows the same growth as the number of simultaneous compositions. In the worst case, the runtime consumes up to the total of memory allocated to it. Our current implementation relies on Node.js which limits the memory of a single process to 4 GB. However, as compositions are independent from each others, it is possible to increase the number of simultaneous compositions by distributing them over a cluster of several instances of Node.js processes.

5.3 Evaluating the MEDLEY platform

We perform a series of experiments to evaluate the runtime performance and scalability of the MEDLEY platform. We show that our approach enables overcoming API rate limit rules of third-party services and scaling the number of executed composite services linearly with the number of nodes of the cluster. We then compare the performance of our cache-aware scheduler with several traditional solutions implemented by Docker Swarm.

Experimental protocol

```
composition {
  process getWeather = require("Mock/GetWeather");
  process notify = require("Mock/Notify");
  // ...
  stream weather = getWeather.invoke({ "city": "BDX" });
  on (weather:out as w) do {
    notify.invoke({
      "to": "+33612345678",
      "body": "Current weather: {{w.temperature}} Celsius."
      });
    }
  }
}
```

Figure 5.8 – ARIA specification of a composition that notifies the user about the current weather.

The workload we use is based on both CPRODIRECT's business plan and their preliminary estimations of the platform usage. We consider 50 different compositions and 50 third-party services. Each composition retrieves an information from one of the services and then triggers a notification to the user. An example of an ARIA specification used as a composition within our workload is shown in Fig. 5.8. Here, the composition consists in retrieving the current weather of a city and then sending a SMS message to the user. Note that the data returned by the weather service is cacheable for a duration between 30 to 60 seconds.

From these 50 ARIA specifications, we generate a set of 1,000 jobs based on the estimations of CPRODIRECT regarding the subscription plans and the frequency of execution for each plan. Let $\mathcal{F}$ be the minimal time interval between two consecutive executions of the same composite service for a given user. We consider that 10% of the users have subscribed a plan with a frequency of execution of $\mathcal{F}$, 30% with a frequency of $2 * \mathcal{F}$ and 60% with a frequency of $3 * \mathcal{F}$.

Our experiment consists in processing all the jobs in the task queue. Each run is repeated three times for better precision. We then compare our own strategy (MC) with two

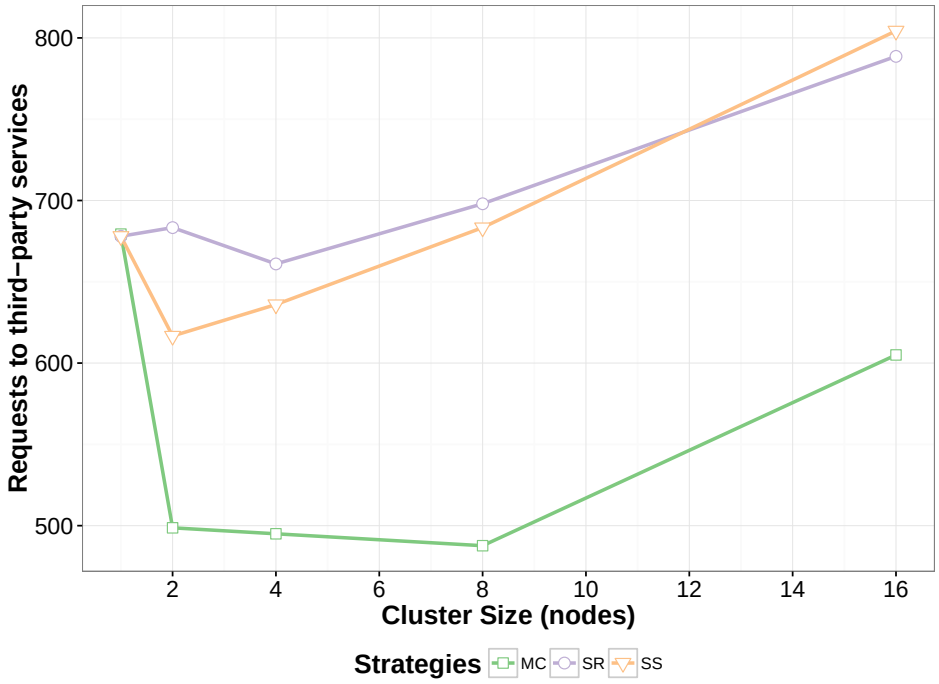


Figure 5.9 – Number of requests to third-party services.

placement strategies provided by Docker Swarm [2]. The *random* strategy (SR) randomly select a node to run a worker. The *spread* strategy (SS) schedules a job on the node with the smallest number of running workers (containers).

Experimental setup

To make our experiments reproducible and agnostic of the network, we developed several mock servers. These rely on data collected from real invocation of service providers such as *GitHub*, *Yahoo Finances*, and *OpenWeatherMap*. During the data collection phase, we gathered the HTTP headers, body and round-trip time of each response. This information enables us to mimic the behavior of real worldwide service providers. For instance, the mock server uses the response time collected in the previous step to delay its response upon a request.

For our experiments, we use one machine to host the mock servers and one machine to host the MEDLEY scheduler along with its task queue and global storage. In addition, we use a cluster of nodes for running instances of the MEDLEY execution engine. We vary the size of the cluster, ranging from 1 node, 2 nodes, 4 nodes, 8 nodes, up to 16 nodes. Each

2. https://docs.docker.com/swarm/scheduler/strategy

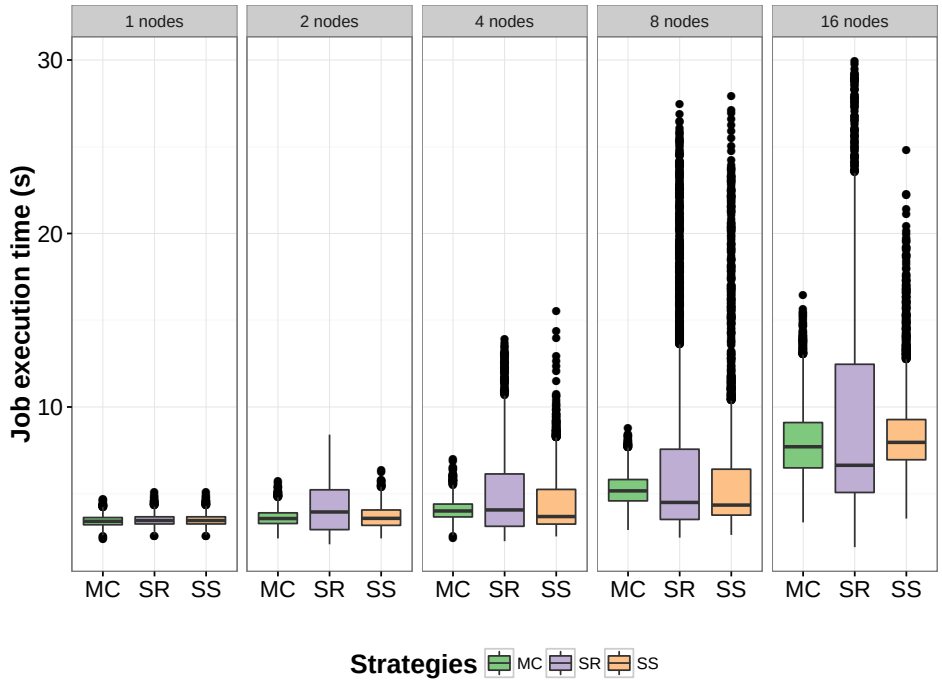


Figure 5.10 – Distribution of job execution time per cluster size and strategy.

node is powered by four Intel(R) Xeon(R) CPU E5-4620 0 @ 2.20 GHz cores and 4 GB of memory.

5.3.1 Overcoming API rate limits

Figure 5.9 depicts the number of requests to third-party services for a cluster size ranging from 1 to 16 nodes. Our strategy (*MC*) consistently outperforms both Docker Swarm *spread* and *random* scheduling strategies. When the number of nodes grows, cache locality decreases since cache entries are distributed across several nodes. This leads to an increase in the number of requests. However, this happens slower for our policy than for other Docker strategies. This behaviour can be observed in the segment from 8 nodes to 16 nodes using *MC*. It is worth noting that the worst behaviour appears with a cluster size of one node. This is due to the fact that processing all the jobs using a single node takes more time; during that time, cache entries have a higher chance of expiring before being reused by other jobs. Thus, the increasing number of cache expirations leads to a higher number of requests.

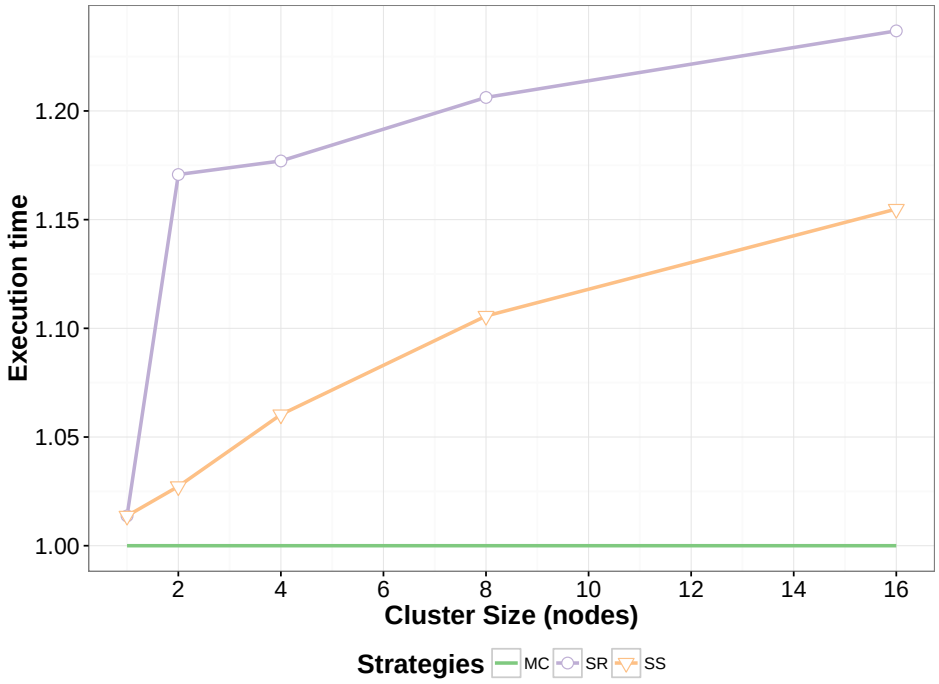


Figure 5.11 – Normalized execution time (our approach (*MC*) is the baseline).

5.3.2 Scalability performance

Job execution time is an important metric to assess the behaviour of MEDLEY. Figure 5.10 presents a boxplot with the distribution of job execution time for all strategies using different cluster sizes. As expected, the strategy selected has no impact on job execution time when only one node is used. The first thing worth noting is that our strategy has greater *stability* than both Docker *spread* and Docker *random*. In particular, we can observe that the range of values for *SS* and *SR* is larger than *MC*'s when four or more nodes are used. In addition, many outliers appear in Docker strategies for 8 and 16 nodes.

The *stability* of our scheduling strategy positively affects the number of jobs we can process in a given amount of time. Figure 5.11 shows the normalized time required to process 1,000 jobs using different cluster sizes and scheduling policies. We use our own strategy as the baseline. When using a cluster of 8 nodes, the overhead of using off-the-shelf techniques ranges from 10% with *SS* to 20% with *SR*. When the cluster has 16 nodes, the overhead increases up to 15% using *SS* and 23% using *SR*.

5.4 Summary

In this section, we presented a thorough evaluation of all the underlying elements of the MEDLEY platform. First, we used POLLY to automatically generate custom change detectors for six use cases provided by our industrial partner CPRODIRECT. Our evaluation (Sec. 5.1) shows that POLLY outperforms a handwritten implementation in terms of code verbosity, and that POLLY outperforms a state-of-the-art off-the-shelf differencing tool in terms of running time and output size. Second, we demonstrated how easy it is to develop various compositions involving a large number of existing services, using the ARIA domain-specific language. We showed how ARIA is a highly-expressive language that is richer in functionalities and language constructs compared to existing solutions (Sec. 5.2). Third, we presented a performance evaluation of the MEDLEY platform on a single host. We demonstrated how MEDLEY consumes a reasonably low amount of resources and how the platform scales well both on a mainstream server and an embedded device such as a Raspberry Pi (Sec. 5.2.2). Finally, we evaluated our approach for scaling the MEDLEY platform in a distributed context (Sec. 5.3). We proposed a new strategy for scheduling the execution of service compositions on a cluster of nodes. Our goals are reducing the number of requests to third-party services and reducing the cost of processing compositions. The proposed scheduling policy assumes that each node in the cluster has a cache for the latest HTTP responses, as well as the compiled code of the compositions to execute. Using information about the content of these caches, the scheduler places incoming jobs in a way that increases cache affinity. We showed that such a mechanism helps in achieving our goals. By conducting a set of experiments, we showed that our strategy outperforms well-established approaches in terms of minimizing the number of outgoing requests. Likewise, we also showed that we are able to reduce the cost of processing a large set of compositions.

CHAPTER

6 Conclusion

In light of the recent advances in the field of web engineering, along with the decrease of cost of cloud computing, service-oriented architectures rapidly became the leading solution in providing valuable services to clients. Typically, these solutions are provided to a broad number of clients in the form of specialized, well-defined web services. Following this trend, the composition of third-party services has become a successful paradigm for the development of robust and rich distributed applications. Although such compositions can be implemented manually, it can be a tedious, error-prone and challenging task to accomplish, especially when dealing with a large number of heterogeneous third-party services. Furthermore, distributing the load in an efficient and scalable way while also taking into account runtime constraints is not a straightforward task. In this concluding chapter, we summarize our contributions in web service composition, and present some interesting research perspectives

6.1 Context and contributions

The work presented in this dissertation lays ground for a comprehensive approach to address the underlying challenges in service composition. To meet their clients needs, CPRODIRECT aims at becoming the leading solution for service orchestration. This translates into several requirements that we address in this thesis:

— the ability to rapidly design and deploy service compositions of arbitrary complexity

— the ability to detect specific change events that occur on third-party services, in order to trigger the execution of said compositions

— the ability to provide a lightweight, low overhead runtime system to enable on-premise deployments for enterprise clients

— the ability to efficiently scale according to the resources available and dynamically adapt according to the API rate limits

Consequently, we investigated each one of these requirements, and proposed a comprehensive approach for web service composition using the MEDLEY platform. A summary of the proposed contributions is presented below.

6.1.1 ARIA: a domain-specific language for web service composition

Our first contribution, presented in Sec. 3.3, aims at identifying and addressing the multiple issues faced by developers when composing several heterogeneous web services. Furthermore, CPRODIRECT requires that the proposed solution should be of a sufficiently high level of abstraction, to ease its use and make it more accessible to a larger number of clients.

ARIA meets these requirements by providing a high-level domain-specific language to describe service compositions in a simple yet highly expressive way. Using dedicated high-level constructs and domain-specific semantics, ARIA enables users to have a fine-grain tuning of both control flow and data flow of service compositions. By relying on an event-driven paradigm, users express compositions by invoking processes that encapsulate the services logic, and react on the events emitted on their output streams. To improve the robustness and reliability of the composition, process pools can be used to dynamically select the services based on a given strategy. This contribution has been published in the *International Conference on Web Engineering (2016)* [Ben Hadj Yahia et al., 2016b].

6.1.2 POLLY: a language-based approach for custom change detection of web service data

Our second contribution, presented in Sec. 3.2, aims at identifying and addressing the underlying challenges in detecting specific changes across a multitude of web API endpoints. To reduce time to market, CPRODIRECT requires a solution to rapidly integrate new service providers and provide a way to monitor change events and react upon their occurrence.

For this purpose, POLLY provides a declarative domain-specific language to describe custom change detection strategies in web services data. By leveraging the domain knowledge of the user, POLLY offers concise, yet highly expressive constructs for specifying custom change detectors. The language provides a simplified syntax to collect data from one or several web API endpoints, supporting automatic pagination and request chaining in sequential and parallel order. Furthermore, it enables users to precisely describe what constitutes a relevant change in a given scenario, while also allowing them to template the final output to extract only the necessary data, thus improving the overall performance. We showed the applicability of POLLY using real scenarios provided by CPRODIRECT, and

demonstrated its efficiency compared to traditional approaches. This contribution has been published in the *International Conference on Service-Oriented Computing (2017)* [Ben Hadj Yahia et al., 2017].

6.1.3 MEDLEY: an event-driven, lightweight platform for service composition

The objectives of our third contribution, presented in Sec. 4.1, are twofold. First, the design of a runtime system is needed to support the execution of service compositions specified using the ARIA language. Furthermore, CPRODIRECT requires that such a system must be lightweight, efficient and extensible to enable its deployment in enterprise environments. Second, the execution platform must enable triggering the execution of compositions when change events are detected using POLLY.

In this contribution, we presented the implementation of the runtime system of the MEDLEY platform. We described the internals of the compiler, and how an event-driven messaging model is used to generate events and route them to the appropriate processes. The platform also provides authentication and error handling mechanisms, as well as a plugin-based mechanism for integrating third-party services. Through a series of experiments, we demonstrated the efficiency of the MEDLEY runtime in a single-host setup. This contribution has been published in the *International Conference on Web Engineering (2016)* [Ben Hadj Yahia et al., 2016b].

6.1.4 Towards a scalable service composition platform

Our fourth and final contribution, presented in Sec. 4.3, investigates the challenges in scaling web service composition platforms. As CPRODIRECT wishes to support a large number of clients on its MEDLEY platform, constraints such as API rate limits and runtime performance must be carefully addressed.

To this end, our contribution proposes a novel approach for efficiently scaling web service composition engines. Using a distributed architecture, composition executions are spread across a variable number of cluster nodes. Whenever they are run, compositions are precompiled and locally cached, along with HTTP responses of the invoked third-party services, to improve performance. Compositions are executed in isolated sandboxes to prevent runtime misbehaviour from affecting other instances. Furthermore, a cache-aware scheduler is used to dispatch composition executions across the cluster in an efficient way. Its placement policy aims at optimizing cache affinity, while also taking into account the node's available resources. Maximizing cache hits positively improves the overall performance, and decreases the consumption rate of the API request quotas, thus allowing to support more clients. Finally, our evaluation shows that our approach outperforms existing and well-established scheduling strategies. This contribution has been published in the *International Middleware Conference (2016)* [Ben Hadj Yahia et al., 2016a].

6.2 Perspectives

As demonstrated throughout this dissertation, web service composition is a complex domain of research, requiring expertise in several fields such as software engineering, web engineering, domain-specific languages and distributed systems. Furthermore, applying these fields in an industrial context brings forth its own set of challenges. This leaves room for a number of interesting research axes worth investigating. We propose in this section several perspectives for the MEDLEY platform.

6.2.1 A formal verification model for data privacy in ARIA

In the industrial world, data privacy is of utmost importance, especially in the context of sensitive and confidential data. Some institutions are even required by law to provide guarantees about the privacy and integrity of their clients information. For instance, banking and health institutions need to comply with regulations and compliances such as HIPAA [1], HITRUST [2], SOC 2 [3] and SOC 3 [4]. As the trend of using multiple, specialized services is becoming more prevalent, it is crucial to maintain these guarantees across all composed services.

An interesting approach would be proposing a formal verification model for ARIA to verify and enforce privacy and security policies throughout the composition platform. Such a model would enable performing dataflow analyses of orchestrations to verify if sensitive data may be compromised and exposed to untrusted services. Furthermore, these policies can be applied at the user level as well, to enable the administrators to define access control levels (ACL) and prevent unauthorized users from accessing or manipulating sensitive data in their compositions.

6.2.2 A large-scale developer survey for POLLY

In Sec. 3.2, we showed how POLLY addresses several issues in detecting custom changes in web services data. One of the initial design goals was to provide a domain-specific language that is expressive, simple to use and less verbose than handwritten implementations. In this sense, it would be insightful to conduct a large-scale developer survey to assess the benefits of using POLLY in terms of productivity, code quality and maintenance cost. The survey would help identifying possible enhancements and optimization opportunities, as well as highlighting common pitfalls, best practices, and learning curve. The study can consist in asking developers to implement predefined scenarios, using both POLLY and the programming language of their choice, followed up by a survey to rate both approaches

1. https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html
2. https://hitrustalliance.net
3. https://www.ssae-16.com/soc-2
4. https://www.ssae-16.com/soc-3

according to the criteria mentionned earlier. Analyzing the results would give us a better insight about the effectiveness of POLLY. As an effort to showcase our current implementation, an online demonstration of POLLY is freely available at the following address [5].

6.2.3 Refining job placement strategies in MEDLEY using machine-learning techniques

In Sec. 4.3, we showed how we efficiently distributed composition execution jobs across a cluster of nodes, with the aim of maximizing cache affinity whenever possible. As most compositions are recurrently executed at a given frequency, it becomes possible to progressively estimate the resources consumed for a given composition, refining it over time. As such, we envision a refinement of the scheduler where the resource usage of each job is also considered. Using machine-learning techniques, a runtime profile can be built for each composition, which can be leveraged by the scheduler to improve job placement. This approach has the added benefit of improving the overall resource usage of the cluster, which leads to better provisioning and lower operational costs.

5. https://demo.pollyapp.ml

APPENDIX

A Résumé en Français

Un service web repose sur un ensemble de standards bien définis afin de permettre la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. Grâce aux dernières avancées technologiques dans le domaine du génie logiciel, le développement de services web est désormais de plus en plus accessible, et de moins en moins coûteux [Zhang et al., 2010]. Ainsi, un vaste nombre de services à valeur ajoutée sont disponibles aujourd'hui sur le marché, témoignant d'une forte croissance au quotidien. L'abondance de ces services a suscité l'intérêt des chercheurs et des entreprises afin d'exploiter leur potentiel dans plusieurs façons possibles [Alonso et al., 2004]. Plus particulièrement, notre partenaire industriel CPRODIRECT cherche à exploiter cette panoplie de services afin de proposer des solutions pertinentes et efficaces à ses clients, grâce à l'intégration et l'orchestration de ces services. Cependant, avec la disponibilité de centaines de milliers de services et APIs différentes, ces intégrations deviennent fastidieuses quand effectuées manuellement. De plus, chaque client peut exiger des contraintes et politiques d'intégration différentes, complexifiant davantage la tâche. Enfin, concevoir et fournir une solution qui soit à la fois robuste et *scalable* est une tâche non-triviale. Au vu de cette forte croissance des architectures orientées services, il est donc nécessaire d'étudier comment coordonner de manière efficace les interactions entre des services web existants. Cette thèse a pour objectif d'étudier les problématiques sous-jacentes à la composition de services web, dans le contexte des architectures web modernes.

Au cours des dernières décennies, de nombreuses approches et solutions ont été proposées pour aborder ces problèmes. Par exemple, BPEL (Business Process Execution Language) [Andrews et al., 2003] était la solution de référence pour l'orchestration des services SOAP, et était le sujet de nombreuses études et applications commerciales. Cependant,

les services SOAP disparaissent rapidement aujourd'hui, en faveur du style architectural REST [Fielding, 2000] qui présente plus de flexibilité. En effet, il existe des différences fondamentales entre les services web hérités (SOAP) et les services web développés selon le style architectural plus moderne (REST), en terme de spécifications, d'outils et des bonnes pratiques à adopter. De nombreux autres modèles (composition sémantique [Rao and Su, 2003], composition basée sur l'ontologie [Agarwal et al., 2003]), langages (algèbres de processus [Morimoto, 2008; Aceto and Gordon, 2008]) et extensions de BPEL ont été proposés au fil du temps [Sheng et al., 2014]. Néanmoins, ceux-ci n'adressent pas les besoins du monde industriel, qui exige une grande rapidité d'intégration avec les nouveaux services émergents.

Dans le milieu commercial, de nombreuses plateformes de composition telles que IFTTT [1] et Zapier [2] permettent à leurs utilisateurs d'exprimer des compositions de services dans le but d'automatiser des tâches récurrentes [Liu et al., 2000; Pandey et al., 2004]. Ces compositions se déclenchent quand un ou plusieurs événements ont lieu sur un service donné, puis exécutent la logique de composition correspondante [Ur et al., 2016]. Cependant, ces plateformes restent très limitées en expressivité, et ne permettent pas de spécifier des compositions plus complexes. De plus, comme la grande majorité des services web ne fournissent pas de moyens pour définir des événements personnalisés, il est à la charge de la plateforme concernée de développer un système de notification, consistant à monitorer des services en les interrogeant régulièrement afin de détecter des changements au fil du temps. Dès qu'un changement est détecté, un événement peut être levé. Comme il est nécessaire d'écrire du code spécifique pour chaque service à intégrer, cette approche devient très vite limitée et risque de ne pas correspondre aux attentes des clients. Tous ces éléments soulèvent donc un grand nombre de défis à aborder dans le contexte de la composition des services web modernes.

Dans cette thèse, nous identifions les problématiques à considérer dans le contexte de la composition de services web modernes (Chapitre 2). Ici, nous adressons les besoins suivants:

— La capacité de rapidement concevoir et déployer des compositions de services d'une complexité arbitraire

— La capacité de détecter des événements de changement spécifiques qui ont lieu sur des services tiers, afin de déclencher l'exécution de ces compositions

— La capacité de fournir un environnement d'exécution léger et performant pour permettre le déploiement auprès des clients industriels

— La capacité de passage à l'échelle efficace en fonction des ressources disponibles, et l'adaptation dynamique en fonction des limites sur les taux de requêtes d'APIs

1. https://ifttt.com
2. https://zapier.com

En se basant sur une approche langage, nous proposons un cadre architectural complet qui permet la spécification et l'exécution des compositions de services web de manière *scalable*. Pour cela, nous proposons quatre contributions complémentaires.

Premièrement, nous proposons ARIA, un langage dédié pour décrire des compositions de services grâce à des constructions de langage de haut niveau, ainsi que des sémantiques spécifiques au domaine de métier (Sec. 3.3). ARIA est conçu spécifiquement pour confronter les éléments problématiques mentionnés auparavant, rencontrés lors de l'orchestration de plusieurs services web hétérogènes. En fournissant une couche d'abstraction entre l'implémentation de bas niveau et la logique métier de haut niveau, le langage permet aux utilisateurs d'exprimer des compositions à un degré fin à la fois le flux de contrôle et le flux de données. Afin d'améliorer la robustesse et la fiabilité de la composition, des *pools* (groupements) de processus peuvent être utilisés afin de sélectionner dynamiquement les services à invoquer en fonction d'une stratégie donnée.

Deuxièmement, nous proposons POLLY, une approche déclarative et orientée langage pour simplifier la construction des détecteurs de changement (Sec. 3.2). POLLY permet de décrire des stratégies de détection de changement dans les documents JSON, obtenus à travers les APIs REST. Ce langage dédié fournit des constructions déclaratives, simples et expressives pour décrire comment construire un état à partir d'une ou plusieurs routes d'APIs, comment identifier des changement dans ces états, et comment produire une sortie personnalisée en fonction des attentes du client. Le compilateur de POLLY produit automatiquement une implémentation efficace en JavaScript, qui est exécuté dans un environnement d'exécution dédié, et fait abstraction des contraintes techniques de bas niveau, telles que l'authentification HTTP et la pagination des réponses. Dans notre contexte, les détecteurs de changement développés avec POLLY permettent de générer des événements personnalisés afin de déclencher automatiquement l'exécution des compositions ARIA quand un changement a lieu dans les données d'un service web donné.

Troisièmement, nous présentons l'architecture de MEDLEY, une plateforme légère et orientée événements pour la composition de services web (Sec. 4.1). La plateforme MEDLEY consiste en un environnement d'exécution léger pour supporter l'exécution de compositions de services spécifiées avec le langage ARIA, et permet l'intégration rapide des fournisseurs de service web tiers. Une fois définies, les spécifications ARIA sont compilées vers du code bas niveau, qui est ensuite exécuté au sein de MEDLEY. L'environnement d'exécution repose sur un paradigme de communication événementiel et basé sur les processus, pour fournir un modèle d'exécution léger et à haute performance. MEDLEY supporte aussi l'intégration des détecteurs de changement spécifiés avec POLLY, permettant ainsi la mise en place de déclencheurs pour les compositions ARIA, basés sur les événements de changement détectés par POLLY.

Enfin, afin de s'assurer de la capacité de passage à l'échelle de la plateforme MEDLEY dans un environnement de production, nous proposons une nouvelle approche pour un ordonnancement efficace dans les moteurs d'orchestration de services (Sec. 4.3). Le défi principal consiste à supporter un nombre croissant d'utilisateurs tout en prenant en

compte les limites sur les taux de requêtes des APIs des services tiers invoqués par les compositions. Ainsi, nous permettons le passage à l'échelle horizontal de MEDLEY. Cela permet de distribuer la charge de la plateforme à travers plusieurs nœuds. À cet effet, de manière similaire à Docker Swarm [Merkel, 2014], nous introduisons un ordonnanceur dédié à la plateforme MEDLEY, afin de pouvoir créer un cluster capable d'augmenter ou de réduire dynamiquement le nombre de noeuds MEDLEY afin de distribuer au mieux la charge. Contrairement à Docker Swarm, qui est agnostique à l'application conteneurisée, notre ordonnanceur est capable d'expédier des compositions en fonction de leurs dépendances, mais aussi des ressources qu'elles consomment. En conséquence, la plateforme MEDLEY est facilement déployable sur des infrastructures cloud publiques, permettant ainsi l'optimisation des coûts opérationnels. De plus, afin de gérer les limites sur les taux de requêtes d'APIs des services tiers, la plateforme MEDLEY dispose de capacités de cache sur chaque nœud du cluster. L'ordonnanceur de MEDLEY se base sur des heuristiques afin d'optimiser l'affinité du cache, réduisant ainsi le nombre total de requêtes aux services tiers, et améliorant le passage à l'échelle de la plateforme.

Pour évaluer la pertinence de notre approche, nous présentons une évaluation approfondie dans Chapitre 5. Nous évaluons l'expressivité et les fonctionnalités des langages dédiés proposés, et conduisons une évaluation de performance de la plateforme MEDLEY, puis discutons des résultats obtenus. Enfin, le Chapitre 6 conclut cette thèse en résumant nos contributions, et en présentant plusieurs perspectives possibles pour étendre ces travaux.

Bibliography

Abiteboul, S. (2002). Issues in Monitoring Web Data. In Hameurlain, A., Cicchetti, R., and Traunmüller, R., editors, *Database and Expert Systems Applications*, number 2453 in Lecture Notes in Computer Science, pages 1–8. Springer Berlin Heidelberg. DOI: 10.1007/3-540-46146-9_1. Cited page 35.

Aceto, L. and Gordon, A. D. (2008). *Algebraic process calculi: The first twenty five years and beyond*. Elsevier. Cited pages 21 and 100.

Agarwal, S., Handschuh, S., and Staab, S. (2003). Surfing the Service Web. *The Semantic Web-ISWC 2003*, pages 211–226. Cited pages 21 and 100.

Alarcon, R., Wilde, E., and Bellido, J. (2010). Hypermedia-Driven RESTful Service Composition. In *ICSOC Workshops*, pages 111–120. Springer. Cited page 18.

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004). *Web Services: Concepts, Architectures and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg. Cited pages 2 and 99.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., and others (2003). *Business Process Execution Language for Web Services*. version. Cited pages 2, 22, 83, and 99.

Baeten, J. C. (2005). A Brief History of Process Algebra. *Theoretical Computer Science*, 335(2-3):131–146. Cited page 22.

Baeza-Yates, R., Castillo, C., Junqueira, F., Plachouras, V., and Silvestri, F. (2007). Challenges on Distributed Web Retrieval. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 6–20. IEEE. Cited page 35.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015). Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In *European Conference on Service-Oriented and Cloud Computing*, pages 201–215. Springer. Cited page 11.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3):42–52. Cited page 14.

Ben Hadj Yahia, E., Falleri, J.-R., and Réveillère, L. (2017). Polly: A Language-Based Approach for Custom Change Detection of Web Service Data. In *Proceedings of the 15th International Conference on Service-Oriented Computing*. To appear. Cited page 95.

Ben Hadj Yahia, E., Gonzalez-Herrera, I., Bayle, A., Bromberg, Y.-D., and Réveillère, L. (2016a). Towards Scalable Service Composition. In *Proceedings of the Industrial Track of the 17th International Middleware Conference*, page 3. ACM. Cited page 95.

Ben Hadj Yahia, E., Réveillère, L., Bromberg, Y.-D., Chevalier, R., and Cadot, A. (2016b). Medley: An Event-Driven Lightweight Platform for Service Composition. In Bozzon, A., Cudré-Mauroux, P., and Pautasso, C., editors, *Web Engineering - 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings*, volume 9671 of *Lecture Notes in Computer Science*, pages 3–20. Springer. Cited pages 46, 94, and 95.

Ben-Kiki, O., Evans, C., and Ingerson, B. (2005). YAML Ain't Markup Language (YAML™) Version 1.1. *yaml. org, Tech. Rep.* Cited page 46.

Bergland, G. D. (1981). *A Guided Tour of Program Design Methodologies*. IEEE. Cited page 10.

Bergstra, J. A. and Klop, J. W. (1985). Algebra of communicating processes with abstraction. *Theoretical computer science*, 37:77–121. Cited page 22.

Bille, P. (2005). A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239. Cited page 36.

Bolognesi, T. and Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN systems*, 14(1):25–59. Cited page 22.

Bonetta, D., Peternier, A., Pautasso, C., and Binder, W. (2012). S: a scripting language for high-performance RESTful web services. *ACM SIGPLAN Notices*, 47(8):97–106. Cited pages 28 and 83.

Borgolte, K., Kruegel, C., and Vigna, G. (2014). Relevant change detection: a framework for the precise extraction of modified and novel web-based content as a filtering technique for analysis engines. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 595–598. ACM. Cited page 36.

Bosch, J. (2004). Software Architecture: The Next Step. *EWSA*, 3047:194–199. Cited page 10.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. (2000). *Simple object access protocol (SOAP) 1.1*. Cited page 15.

Brewington, B. E. and Cybenko, G. (2000). How dynamic is the Web? *Computer Networks*, 33(1):257–276. Cited page 35.

Brooks, F. P. (1975). *The Mythical Man-Month*. Addison-Wesley Reading, MA. Cited page 10.

Bryan, P. and Nottingham, M. (2013). JavaScript Object Notation (JSON) Patch. Technical report. Cited pages 37 and 80.

Buttler, D. (2004). A short survey of document structure similarity algorithms. In *International conference on internet computing*, pages 3–9. Cited pages 5 and 36.

Cao, H., Falleri, J.-R., Blanc, X., and Zhang, L. (2016). JSON Patch for Turning a Pull REST API into a Push. In *International Conference on Service-Oriented Computing*, pages 435–449. Springer. Cited pages 37 and 79.

Chappell, D. (2004). *Enterprise Service Bus*. " O'Reilly Media, Inc.". Cited page 4.

Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., and others (2001). *Web Services Description Language (WSDL) 1.1*. Cited pages 15 and 23.

Clark, J., DeRose, S., and others (1999). *XML Path Language (XPath) Version 1.0*. Cited page 26.

Cobena, G., Abiteboul, S., and Marian, A. (2002). Detecting Changes in XML Documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 41–52. IEEE. Cited page 37.

Curbera, F., Duftler, M., Khalaf, R., and Lovell, D. (2007). *Bite: Workflow composition for the web*. Springer. Cited page 28.

Cáceres, J., Vaquero, L. M., Rodero-Merino, L., Polo, A., and Hierro, J. J. (2010). Service Scalability Over the Cloud. In *Handbook of Cloud Computing*, pages 357–377. Springer. Cited page 6.

Danielsen, P. J. and Jeffrey, A. (2013). Validation and Interactivity of Web API Documentation. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 523–530. IEEE. Cited page 17.

Dinh, P. C. and Boonkrong, S. (2013). The Comparison of Impacts to Android Phone Battery Between Polling Data and Pushing Data. In *IISRO Multi-Conferences Proceeding. Thailand*, pages 84–89. Cited page 36.

Douglis, F., Feldmann, A., Krishnamurthy, B., and Mogul, J. C. (1997). Rate of Change and other Metrics: a Live Study of the World Wide Web. In *USENIX Symposium on Internet Technologies and Systems*, volume 119. Cited page 35.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2016). Microservices: Yesterday, Today, and Tomorrow. *arXiv preprint arXiv:1606.04036*. Cited page 13.

Dustdar, S. and Schreiner, W. (2005). A survey on web services composition. *International journal of web and grid services*, 1(1):1–30. Cited page 21.

Emmerich, W., Butchart, B., Chen, L., Wassermann, B., and Price, S. L. (2005). Grid service orchestration using the business process execution language (BPEL). *Journal of Grid Computing*, 3(3):283–304. Cited page 24.

Farrell, S. (2009). API Keys to the Kingdom. *IEEE Internet Computing*, 13(5). Cited page 68.

Ferrara, A. (2004). Web Services: A Process Algebra Approach. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251. ACM. Cited page 22.

Fetterly, D., Manasse, M., Najork, M., and Wiener, J. (2003). A Large-Scale Study of the Evolution of Web Pages. In *Proceedings of the 12th international conference on World Wide Web*, pages 669–678. ACM. Cited page 35.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext Transfer Protocol - HTTP/1.1. Technical report. Cited page 17.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine. Cited pages 2, 17, and 100.

Fokaefs, M., Oprescu, M., and Stroulia, E. (2015). WSDarwin: A Web Application for the Support of REST Service Evolution. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 336–338. IEEE. Cited page 19.

Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2005). Tool support for model-based engineering of web service compositions. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages 95–102. IEEE. Cited page 22.

Fowler, M. and Foemmel, M. (2006). Continuous Integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122. Cited page 14.

Fowler, M. and Lewis, J. (2014). Microservices. *ThoughtWorks. http://martinfowler. com/articles/microservices. html [last accessed on February 17, 2015]*. Cited pages 3, 11, and 13.

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. (1999). HTTP authentication: Basic and digest access authentication. Technical report. Cited page 68.

Fuchs, A. and Gürgens, S. (2013). Preserving Confidentiality in Component Compositions. In Binder, W., Bodden, E., and Löwe, W., editors, *Software Composition*, number 8088 in Lecture Notes in Computer Science, pages 33–48. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-39614-4_3. Cited page 4.

Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J., and Montesi, F. (2016). Self-Reconfiguring Microservices. In *Theory and Practice of Formal Methods*, pages 194–210. Springer. Cited page 14.

Galiegue, F., Zyp, K., and others (2013). JSON Schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, page 32. Cited pages 19 and 57.

Gao, S., Sperberg-McQueen, C. M., Thompson, H. S., Mendelsohn, N., Beech, D., and Maloney, M. (2009). W3c XML Schema Definition Language (XSD) 1.1 part 1: Structures. *W3C Candidate Recommendation*, 30(7.2). Cited page 16.

Goessner, S. (2007). JSONPath - XPath for JSON. *URL http://goessner. net/articles/JsonPath*. Cited pages 48 and 59.

Guidi, C., Lanese, I., Montesi, F., and Zavattaro, G. (2009). Dynamic error handling in service oriented applications. *Fundamenta Informaticae*, 95(1):73–102. Cited page 35.

Hadley, M. J. (2006). Web Application Description Language (WADL). Cited page 18.

Hammer-Lahav, E. (2010). The OAuth 1.0 Protocol. Cited page 68.

Hardt, D. (2012). The OAuth 2.0 Authorization Framework. Cited page 68.

Hasija, N. and Unger, C. H. (2015). *Integration of cloud-based services to create custom business processes*. Google Patents. Cited page 31.

Haupt, F., Fischer, M., Karastoyanova, D., Leymann, F., and Vukojevic-Haupt, K. (2014). Service Composition for REST. In *Enterprise Distributed Object Computing Conference (EDOC), 2014 IEEE 18th International*, pages 110–119. IEEE. Cited page 28.

Higuchi, S., Kan, T., Yamamoto, Y., and Hirata, K. (2012). An A* algorithm for computing edit distance between rooted labeled unordered trees. In *New Frontiers in Artificial Intelligence*, pages 186–196. Springer. Cited page 37.

Hirschberg, D. S. (1977). Algorithms for the Longest Common Subsequence Problem. *Journal of the ACM (JACM)*, 24(4):664–675. Cited page 37.

Hoare, C. A. R. (1978). Communicating Sequential Processes. In *The origin of concurrent programming*, pages 413–443. Springer. Cited page 22.

Hürsch, W. L. and Lopes, C. V. (1995). Separation of Concerns. Cited page 12.

Klusch, M. and Gerber, A. (2006). Fast composition planning of owl-s services and application. In *Web Services, 2006. ECOWS'06. 4th European Conference on*, pages 181–190. IEEE. Cited page 23.

Krause, L. (2014). Microservices: Patterns and Applications. Cited page 13.

Küster, U. and König-Ries, B. (2006). Dynamic binding for BPEL processes-a lightweight approach to integrate semantics into web services. In *ICSOC Workshops*, volume 4652, pages 116–127. Springer. Cited page 21.

Lempsink, E., Leather, S., and Löh, A. (2009). Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming*, pages 61–72. ACM. Cited page 37.

Lindholm, T., Kangasharju, J., and Tarkoma, S. (2006). Fast and Simple XML Tree Differencing by Sequence Alignment. In *Proceedings of the 2006 ACM symposium on Document engineering*, pages 75–84. ACM. Cited page 37.

Liu, L., Pu, C., and Tang, W. (2000). WebCQ - Detecting and Delivering Information Changes on the Web. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 512–519. ACM. Cited pages 5, 28, and 100.

Louridas, P. (2008). Orchestrating Web Services with BPEL. *IEEE software*, 25(2). Cited page 23.

Lucky, M. N., Cremaschi, M., Lodigiani, B., Menolascina, A., and De Paoli, F. (2016). Enriching API Descriptions by Adding API Profiles Through Semantic Annotation. In *International Conference on Service-Oriented Computing*, pages 780–794. Springer. Cited page 19.

MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., Metz, R., and Hamilton, B. A. (2006). Reference Model for Service Oriented Architecture 1.0. *OASIS standard*, 12:18. Cited page 12.

Magee, J., Kramer, J., and Giannakopoulou, D. (1999). Behaviour analysis of software architectures. In *Software Architecture*, pages 35–49. Springer. Cited page 22.

Maximilien, E. M., Wilkinson, H., Desai, N., and Tai, S. (2007). *A domain-specific language for web apis and services mashups*. Springer. Cited page 4.

Mayer, S., Inhelder, N., Verborgh, R., Van de Walle, R., and Mattern, F. (2014). Configuration of smart environments made simple: Combining visual modeling with semantic metadata and reasoning. In *Internet of Things (IOT), 2014 International Conference on the*, pages 61–66. IEEE. Cited page 23.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2. Cited pages 7, 14, 71, and 102.

Milner, R. (1989). *Communication and Concurrency*. International series in computer science. Prentice Hall Englewood Cliffs. Cited page 22.

Milner, R. (1999). *Communicating and mobile systems: the pi calculus*. Cambridge university press. Cited page 22.

Morimoto, S. (2008). A survey of formal verification for business process modeling. *Computational Science–ICCS 2008*, pages 514–522. Cited pages 21 and 100.

Morrison, J. P. (2010). *Flow-Based Programming: A new approach to application development*. CreateSpace. Cited page 65.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580. Cited page 22.

Na, S.-H., Park, J.-Y., and Huh, E.-N. (2010). Personal Cloud Computing Security Framework. In *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*, pages 671–675. IEEE. Cited page 34.

Namiot, D. and Sneps-Sneppe, M. (2014). On Micro-Services Architecture. *International Journal of Open Information Technologies*, 2(9):24–27. Cited page 12.

Narayanan, S. and McIlraith, S. A. (2002). Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, pages 77–88. ACM. Cited page 21.

Nelson, B. J. (1981). Remote Procedure Call. Cited page 3.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc. Cited page 3.

OASIS (2004). UDDI Version 3.0.2. Technical report. Cited page 16.

Olston, C., Najork, M., and others (2010). Web Crawling. *Foundations and Trends® in Information Retrieval*, 4(3):175–246. Cited page 35.

Ovadia, S. (2014). Automate the Internet with "If This Then That"(IFTTT). *Behavioral & social sciences librarian*, 33(4):208–211. Cited page 29.

Pandey, S., Dhamdhere, K., and Olston, C. (2004). WIC: A General-Purpose Algorithm for Monitoring Web Information Sources. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 360–371. VLDB Endowment. Cited pages 5, 28, and 100.

Papazoglou, M. (2008). *Web Services: Principles and Technology*. Pearson Education. Cited page 20.

Papazoglou, M. P. (2003). Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE. Cited page 20.

Pautasso, C. (2008). BPEL for REST. In *Business Process Management*, pages 278–293. Springer. Cited page 28.

Pautasso, C. (2009a). On Composing RESTful Services. *Software Service Engineering*, (09021). Cited page 4.

Pautasso, C. (2009b). RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9):851–866. Cited page 28.

Pautasso, C. and Alonso, G. (2005). Flexible binding for reusable composition of web services. In *Software Composition*, pages 151–166. Springer. Cited page 4.

Pawlik, M. and Augsten, N. (2011). RTED: A Robust Algorithm for the Tree Edit Distance. *PVLDB*, 5(4):334–345. Cited page 37.

Peltz, C. (2003). Web Services Orchestration and Choreography. *Computer*, 36(10):46–52. Cited page 20.

Perry, D. E. and Wolf, A. L. (1992). Foundations for the Study of Software Architecture. *ACM SIGSOFT Software engineering notes*, 17(4):40–52. Cited page 10.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., and Vrgoč, D. (2016). Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee. Cited pages 19 and 57.

Rao, J. and Su, X. (2003). Toward the Composition of Semantic Web Services. In *International Conference on Grid and Cooperative Computing*, pages 760–767. Springer. Cited pages 21 and 100.

Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J. C., Canali, L., and Percannella, G. (2016). REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In *International Conference on Web Engineering*, pages 21–39. Springer. Cited page 17.

Rosenberg, F., Curbera, F., Duftler, M. J., and Khalaf, R. (2008). Composing RESTful services and collaborative workflows: A lightweight approach. *Internet Computing, IEEE*, 12(5):24–31. Cited pages 28 and 83.

Sheng, Q. Z., Qiao, X., Vasilakos, A. V., Szabo, C., Bourne, S., and Xu, X. (2014). Web Services Composition: A Decade's Overview. *Information Sciences*, 280:218–238. Cited pages 19, 83, and 100.

Simon, J., Schmidt, P., and Pammer, V. (2014). An energy efficient implementation of differential synchronization on mobile devices. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 382–383. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). Cited page 37.

Smeds, J., Nybom, K., and Porres, I. (2015). DevOps: a definition and perceived adoption impediments. In *International Conference on Agile Software Development*, pages 166–177. Springer. Cited page 14.

Ur, B., Pak Yong Ho, M., Brawner, S., Lee, J., Mennicken, S., Picard, N., Schulze, D., and Littman, M. L. (2016). Trigger-action programming in the wild: An analysis of 200,000 IFTTT recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 3227–3231. ACM. Cited pages 5, 29, and 100.

Urpalainen, J. (2008). An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors. Cited page 59.

Vaquero, L. M., Rodero-Merino, L., and Buyya, R. (2011). Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52. Cited page 6.

Wang, G. and Fung, C. K. (2004). Architecture Paradigms and their Influences and Impacts on Component-Based Software Systems. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 10–pp. IEEE. Cited page 12.

Wang, Y., DeWitt, D. J., and Cai, J.-Y. (2003). X-Diff: An Effective Change Detection Algorithm for XML Documents. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 519–530. IEEE. Cited page 37.

Weerawarana, S., Curbera, F., Leymann, F., Storey, T., and Ferguson, D. F. (2005). *Web Services Platform Architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR. Cited page 14.

Zhang, K. and Shasha, D. (1989). Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.*, 18(6):1245–1262. Cited page 36.

Zhang, K., Statman, R., and Shasha, D. (1992). On the editing distance between unordered labeled trees. *Information processing letters*, 42(3):133–139. Cited page 37.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18. Cited pages 3 and 99.

Zhao, H. and Doshi, P. (2009). Towards Automated RESTful Web Service Composition. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 189–196. IEEE. Cited page 23.

Zur Muehlen, M., Nickerson, J. V., and Swenson, K. D. (2005). Developing web services choreography standards—the case of REST vs. SOAP. *Decision Support Systems*, 40(1):9–29. Cited page 28.

List of Figures

List of Tables