

# High-Level Modeling and Low-Level Adaptation of Serverless Function Choreographies

Benjamin Walch

Bachelor Thesis

Supervisor:

Dr. Shashko Ristov

Department of Computer Science  
Universität Innsbruck

Innsbruck, April 16, 2020



# Eidesstaatliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

---

Datum

---

Unterschrift



# Abstract

The Distributed and Parallel Systems group developed the *Abstract Function Choreography Language* (AFCL). It is a specification for describing serverless workflows. Furthermore, a Java API, to describe serverless application workflows programmatically, was developed. The product which results in using that API is a workflow being described in AFCL in a text based file. Until now, the workflow being described in the text file has to be created manually (by editing the file directly), or a programmer has to write Java code which utilizes the API to generate the file.

The aim of this bachelor project is to develop a visual workflow editor, which makes modeling of workflows possible at a high level of abstraction. The tool should not only be able to load, display and save workflows, but also optimize workflows for multiple FaaS provider(s) in case of quotas and limits, and also in case of performance.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Serverless Computing . . . . .	3
2.2	FaaS . . . . .	4
2.3	Serverless Functions . . . . .	4
2.4	Serverless Workflows . . . . .	4
2.5	AFCL . . . . .	5
2.5.1	Overview . . . . .	5
2.5.2	Control-flow . . . . .	6
2.5.3	Properties and Constraints . . . . .	6
2.5.4	Data-flow . . . . .	6
2.6	Development Tools and Technologies . . . . .	6
2.6.1	npm . . . . .	6
2.6.2	webpack . . . . .	7
2.6.3	ECMAScript . . . . .	7
2.6.4	SASS . . . . .	8
<b>3</b>	<b>Implementation</b>	<b>9</b>
3.1	System Architecture . . . . .	9
3.1.1	High level overview . . . . .	10
3.1.2	Frontend . . . . .	10
3.1.3	Backend . . . . .	10
3.2	System Design . . . . .	11
3.2.1	Frontend . . . . .	11
3.2.2	Encoding . . . . .	14
3.2.3	Layouting . . . . .	14
3.2.4	Alternatives . . . . .	14

3.2.5	Web Interface . . . . .	15
3.3	Backend . . . . .	18
3.3.1	Setup . . . . .	18
3.3.2	Api . . . . .	18
3.3.3	Decoding . . . . .	18
3.3.4	Persistence . . . . .	19
3.3.5	Adaptation . . . . .	19
<b>4</b>	<b>Evaluation</b>	<b>21</b>
<b>5</b>	<b>Conclusion and Future Work</b>	<b>23</b>
	<b>Bibliography</b>	<b>25</b>



# 1 Introduction

”Run code, not Servers” is a recent term in the cloud computing world. With the rise of serverless technologies during the last years, *Function-as-a-Service* (FaaS) became more and more popular. This cloud computing concept offers new advantages for software development: very high flexibility, nearly unlimited scalability and pay-per-use pricing. Global Players like Amazon, Google, IBM and Microsoft jumped on the train and provide their infrastructure to developers, with the ability to deploy and execute their code in the cloud.

Moving from *Infrastructure-as-a-service* (IaaS) over *Platform-as-a-service* (Paas) to FaaS, developers can completely focus on the program they need to run and not worry at all about the machine it is on or the resources it requires [8]. An entire program can be defined by a workflow, which in turn consists of functions, data dependencies and control structure. Workflows, though, can be executed using serverless technology in the cloud.

But each provider has its own definitions on how a workflow is expressed, as well as different pricing models, limits and quotas. One main issue is the maximum number of concurrent function invocations, which is not more than 1000 for current FaaS providers at time of writing<sup>1</sup>. Additionally, different providers also support different programming languages. That being said, a workflow from one system is not compatible with another, what means, the user is bound to a specific provider (vendor lock-in), if he wants to reuse or change a workflow later.

---

<sup>1</sup>checked providers: AWS, Google, IBM and Microsoft

The distributed and parallel systems group developed a specification to describe serverless workflows at a high abstraction level. This specification is called *Abstract Function Choreography Language* (AFCL) and was created to overcome weaknesses of current FaaS platforms; for example incompatibility of workflows between different providers.

Although many FaaS providers offer a tool to create a workflow (Amazon Step Functions, IBM Cloud Functions, Microsoft Azure Functions, Google Cloud Functions, ...), building workflows is still a complex task. This is also true for AFCL. Most of the time, this has to be done by a software developer, because others simply lack the necessary know-how; and even for developers, this task can be tedious and time-consuming.

Castro et al. mentions that "One of the major challenges slowing the adoption of serverless is the lack of tools and frameworks" [2].

Therefore, this thesis has the goal to implement a new tool providing a system, able to model workflows at a high level of abstraction and export them in AFCL. In addition, visualizing - in particular loading, displaying and editing - of existing AFCL workflows in should be supported. The system should also be able to adapt composed workflows at low-level.

By using the tool, it will be able for non-developers (business guys) to develop a workflow in less time it would take a developer to write the workflow in AFCL. Another advantage is that exported workflows are compliant with the API and the risk of typos is minimized. Regarding provider limitations, a scheduler (which is not part of this thesis) should decide to distribute one big concurrent loop - with more iterations than the maximum number of function invocations allowed by the provider - onto multiple FaaS providers. For this purpose, we will provide a service, which offers this low-level adaptation.

It is assumed that concrete implementations of the functions are already developed and deployed to a FaaS provider.

## 2 Background

This chapter contains important details on the terms and technologies used in this thesis and helps the reader to better understand the further topics.

First of all, the serverless concept, in particular Function-as-a-Service, Serverless Functions and Workflows are explained, before giving a short overview about AFCL. The tools and technologies used for the development are also presented to readers who are not yet familiar with them.

### 2.1 Serverless Computing

Serverless computing is widely known as an event-driven cloud execution model. In this model, the client provides the code and the cloud provider manages the life-cycle of the execution environment of that code. The idea is based on reducing the life span of the program to execute functionality in response to an event. Hence, the program's processes are born when an event is triggered and are killed after the event is processed [5].

Castro et al. define serverless computing as follows: "Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running" [2].

The term 'serverless' can be misleading though, as there are still servers providing these backend services, but all of the server space and infrastructure concerns are handled by the vendor. Serverless means that the developers can do their work without having to worry about servers at all [4].

## 2.2 FaaS

FaaS is a form of serverless computing, which is disrupting the way applications and systems have been built for decades. By abstracting infrastructure provisioning and deployment, user-provided functions can be invoked and executed remotely. The user only has to worry about development and triggering of the function. This serverless runtime has not only the advantage that it avoids costly pre-allocated or dedicated hardware, but also offers almost unlimited possibilities in scalability and billing.

## 2.3 Serverless Functions

Serverless functions - or tasks - are single-purpose, mostly stateless, programmatic functions that are hosted on cloud infrastructure. This infrastructure is provided through a FaaS platform. They are event-driven and can be invoked through the internet, mainly using HTTP. Like conventional functions, they accept arguments and return the result of the computation - with the difference that this is done over a network.

## 2.4 Serverless Workflows

On an abstract level, a workflow consists of a set of interdependent tasks that need to be executed to achieve a specific goal. [...] A task within a workflow has the following properties: dependencies on the software or service used by the task to perform its computation (software flow), dependencies on data (data flow), and dependencies on other tasks (control flow) [3].

Basic and advanced control flow patterns for workflows are shown in [7].

A serverless workflow is a complex workflow defined through the composition of serverless functions, connected by control- and data flow. Serverless workflows make it possible to combine and reuse serverless functions in order to build more complex applications. Workflows can declare the structure of applications - and using formats like YAML, XML or JSON, they can be described in a text-based file.

## 2.5 AFCL

Below, a quick overview to AFCL and the control-flow constructs will be given. More detailed information about AFCL will be available soon in a corresponding paper<sup>1</sup>, or in the documentation.[1]

The *Abstract Function Choreography Language* (AFCL) was created to overcome weaknesses in current FaaS platform implementations. As the name indicates, this specification describes serverless workflows at a high level of abstraction, forming a step on the way to cross-cloud workflow execution.

Consider a workflow built with AWS Step Functions, should be executed on IBM Cloud. To make that work, the workflow has to be ported to the other platform to be compatible. In other words - it has to be recreated by a skilled programmer - for example by using the exported workflow as a template.

The AFCL specification has more extensive control-flow and data-flow constructs available than current FaaS providers offer currently in terms of language features. Therefore, AFCL can not only eliminate the incompatibility of workflows between different providers (vendor lock-in), but also opens the door to execute a workflow by distributing its computation over multiple FaaS providers.

To be able to actually execute such multi-cloud workflows, some more tools are needed: an Enactment Engine and a Scheduler.

### 2.5.1 Overview

AFCL is based on YAML<sup>2</sup> and ships with a schema<sup>3</sup>. There exist two types of functions, base functions and compound functions, which can be connected by specifying control-flow and data-flow information. While base functions represent a single task, compound functions provide nesting - they can include some base functions or even other compound functions.

---

<sup>1</sup>At time of writing, *AFCL: An Abstract Function Choreography Language for Serverless Workflow Specification*, by Sasko Ristov et. al was not yet public available

<sup>2</sup><https://yaml.org>

<sup>3</sup><http://dps.uibk.ac.at/projects/afcl/files/schema/schema.yaml>

Base and compound functions have data input (`dataIns`) and data output (`dataOuts`) ports to define input or output data, respectively. Data input can refer to data output of another function by specifying the `nameIn` combination with the data output port of the other function.

### 2.5.2 Control-flow

In AFCL, base or compound functions are specified one after another, which defines they are executed sequential. However, the following control constructs are introduced with AFCL: `sequence`, `if-then-else`, `switch`, `for`, `while`, `parallel` and `parallelFor`.

### 2.5.3 Properties and Constraints

Additional, optional attributes for `{text}dataIns` and `dataOuts` ports of a function, or for the function itself can be defined in `properties` and `constraints`. While those are simple key-value pairs accepting string values, AFCL has a few defined properties and constraints to specify concrete attributes like invocation type, element index or data distribution information.

### 2.5.4 Data-flow

## 2.6 Development Tools and Technologies

### 2.6.1 npm

The node package manager<sup>4</sup> is the world's largest software registry, where open-source software packages of developers and companies are shared all over the world. Over the last years, npm became a de-facto standard for package management in JavaScript development.

---

<sup>4</sup><https://www.npmjs.org>

## 2.6.2 webpack

webpack is a module bundler, its main purpose is to bundle JavaScript code for the usage in a browser.<sup>5</sup> In particular, multiple modules (often hundreds of) with dependencies [to each other] are processed and bundled into a few files. To be able to process other types of files than JavaScript or JSON, webpack offers the opportunity to configure a **loader**. In this application, the following loaders are configured:

- babel-loader, to transform ECMAScript and React JSX to browser-compatible JavaScript
- sass-loader, to transform SASS to CSS
- css-loader, to transform CSS to CommonJS
- file-loader, to handle static resources like images and fonts

## 2.6.3 ECMAScript

The scripting language specification ECMAScript (ES) was created to standardize JavaScript. With the release of ES6 (also known as ECMAScript 2015), features like class declarations, module imports and arrow function expressions became possible. After ES6, every year a new edition of the ECMAScript standard was finalized and released, offering new features. Worth mentioning here is the rest/spread operator released with ES9, which occurs a lot in the source code of this thesis.

Since current browsers only have partial support of ECMAScript, a 'transcompiler' (or transpiler) is needed to transform the ECMAScript source code to JavaScript, which common browsers are capable of interpreting. This compiling process is done with Babel<sup>6</sup>, which is configured to not only compile ECMAScript, but also JSX to JavaScript.

---

<sup>5</sup><https://webpack.js.org>

<sup>6</sup><https://babeljs.io>

## 2.6.4 SASS

CSS, in its pure form, reaches its limits when one thinks about using variables, functions or nested rules. SASS<sup>7</sup> is a stylesheet language, which is - similiar to ES - compiled to CSS and offers the mentioned and even more features. A lot of CSS Frameworks also provide their source code in SASS, often served with a large variable set which makes it easy to customize it. The transpiler used in this thesis to transform SASS to CSS is node-sass<sup>8</sup>.

---

<sup>7</sup><https://sass-lang.com>

<sup>8</sup><https://github.com/sass/node-sass>



## 3 Implementation

The main goals for this thesis were the implementation of a tool, which should not only visualize AFCL Workflows, but also give non-developers the opportunity to create and edit workflows through a GUI. This GUI ships with a powerful editor with additional features like real-time validation, versioning and automatic layouting.

Furthermore, workflows can be adapted at low-level by dividing concurrency in loops.

The following sections explain the high-level overview architecture and give specific information about how the implementation was done.

### 3.1 System Architecture

The whole system consists of two sub systems which are operating mostly decoupled from each other: a Java backend and a web-based frontend. Those communicate and share JSON or XML encoded objects between each other through an API the backend provides and the frontend consumes.

The backend is responsible for converting and adapting workflows at low-level as well as persisting of function data.

The frontend displays and controls the application's user interface. It is responsible for visualizing the data and provides the editor with the canvas where a user can compose a workflow.

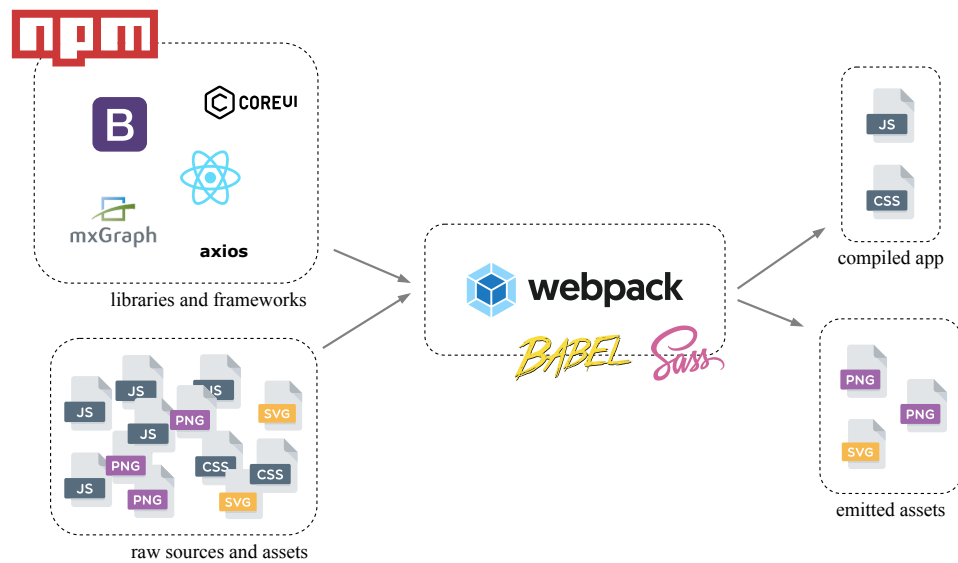


Figure 3.1: frontend build stack

### 3.1.1 High level overview

### 3.1.2 Frontend

### 3.1.3 Backend

The conversion of workflows is needed because the frontend encodes the visual representation of a workflow in XML, which is given as input to the backend. The conversion process is done by utilizing the AFCL API<sup>1</sup>.

<sup>1</sup><http://dps.uibk.ac.at/projects/afcl/files/jar/afclAPI.jar>

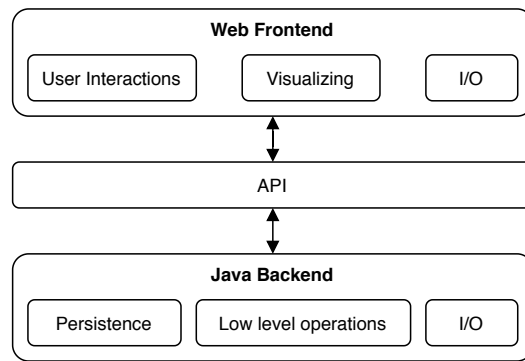


Figure 3.2: application architecture

## 3.2 System Design

Since a web based frontend which runs in the user's web browser, is a hard requirement, the core technologies are limited to HTML, CSS and JavaScript.

### 3.2.1 Frontend

The setup of the frontend application is a selection of tools and technologies for modern and complex web development. The base forms npm, which is used to manage and resolve dependencies. Additionally, npm's CLI is used as tool to execute development and build tasks. Webpack is in use to bundle all sources and assets into single files, and execute additional build steps by utilizing so-called 'loaders'.

Under the hood, JavaScript and several JavaScript Frameworks are in action to control the application consisting of several UI components. To give the application its look and feel, CSS and some CSS Frameworks are in use.

Babel is used as such a webpack loader, for compiling the ECMAScript and React JSX source code to browser-compatible JavaScript. The same applies for the CSS extension SASS, which is also integrated as a webpack loader, and is used to make the process of styling the user interface more efficient.

## Data model

All AFCL Java model classes[1] have been ported to ECMAScript, to have all classes and properties available in the frontend, similar like they are in the backend. Although plain JavaScript objects would also do the job, this adds a kind of type-safety to the ECMAScript sources and improves readability of the code. Also, the data exchange with the backend and the encoding of the graph benefit from this approach, since the XML node names map to constructor names.

## Graph drawing

An AFCL workflow can be visualized as a directed acyclic graph (DAG). For reasons of efficiency, reusability, documentation and future safety, there is no question that an existing library has to be used to do the visualizing part. At the time of implementation, the JavaScript library `mxGraph`<sup>2</sup> was the best choice<sup>3</sup> to achieve the mentioned goals. It has an outstanding documentation, enriched with a lot of demos and example code which show many use cases and extensive features, as well as a powerful production-grade example<sup>4</sup>.

The authors of `mxGraph` state:

“`mxGraph` is pretty much feature complete, production tested in many large enterprises and stable for many years. We actively fix bugs and add features [...]”

In `mxGraph`, vertices and edges are represented by an `mxCell` model, which stores information about the cell’s position, dimension, geometry and style.

---

<sup>2</sup><https://jgraph.github.io/mxgraph/>

<sup>3</sup>see 3.2.3 to learn why

<sup>4</sup><https://draw.io>

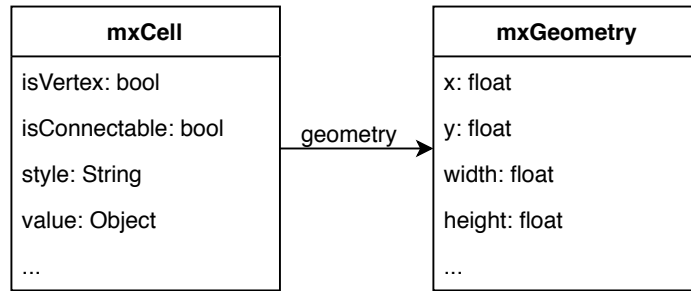


Figure 3.3: `mxCell` model

Additionally, a *user object* can be associated with a `mxCell` through the `value` property. User objects give the graph its context, they store the business logic associated with the visual part. [6] For simple cases, the user object may be a string to display the cell's label. In more complex applications, the user object may be an object and some property of that object will generally be the label that the visual cell displays. In this application, the cell's user objects are instances of the ported AFCL classes, respectively.

The following table shows the implemented `mxCell`'s shapes which are used for composing a workflow. Beside a short description, the type of the associated user object is denoted.

## Validation

A DAG which represents a valid AFCL workflow always fulfills the following constraints:

- There exists exactly one starting point
- There exists exactly one termination point
- There exist no cycles
- Function, If-Then-Else, Switch, Parallel, ParallelFor and End have exactly one incoming edge
- Start, Function, Parallel, ParallelFor and Merge have exactly one outgoing edge
- IfThenElse has exactly two outgoing edges (then, else)

- Switch and Merge have multiple outgoing edges

These constraints are enforced at two different stages: At modeling-time, when composing a workflow, more specifically when adding and connecting vertices, and before exporting or adaptation of a workflow. Due the fact that the user cannot even create an invalid workflow, efficiency increases and errors in the corresponding backend services will be minimized.

### 3.2.2 Encoding

The visual representation of the workflow and the included data targets human-readability. This data needs to be put into another, proper format for storing. mxGraph provides built-in support for converting the visual representation to XML. An advantage of using XML over JSON is that constructor names (class names) are mapped to the XML node names, no additional property is required. XML is also very convenient for the backend side, since Java has excellent built-in XML support. The XML-encoded workflow is converted on backend-side internally to an AFCL object.

### 3.2.3 Layouting

Combination of nested hierarchical layout

### 3.2.4 Alternatives

One of the more difficult tasks was to choose a proper library for graph drawing. Since the graph drawing and visualization is one of the most important features of the frontend, a careful choice has to be made. Of course there exist multiple excellent JavaScript graph drawing libraries.

Below is a list of tested and researched JavaScript graph libraries with additional information why they were insufficient for the project.

- **jsPlumb** is a library of great quality and looks like it would fulfill all requirements for graph drawing and user interaction out of the box. It has a very good documentation, and a lot of examples. Furthermore its animations and drag and drop handling are outstandingly good. Unfortunately this software is not open source and a license costs a few thousand dollars.
- The same goes for **Rappid** (formerly known as JointJS) and **GoJS**, which is even more expensive.
- **diagram-js** is one of the newer diagram libraries. It seems to be a good candidate to draw BPMN<sup>5</sup> graphs. The lack of an appropriate documentation - even after 4 years of existence (there exists an open issue on github<sup>6</sup> for this) - is still an issue.
- **Cytoscape.js** is older than mxGraph but the development and the community is not less active. This library has a very good documentation and a lot of demos are provided. But it offers fewer examples and "out-of-the-box" features for modeling flowcharts than mxGraph does.
- **vis.js** is a star on npmtrends.<sup>7</sup> It has very smooth animations when manipulating the graph. There exists only one example of graph manipulation on its documentation, it is mainly used for visualization only.
- **Sigma js** is a tiny library with a small documentation. It clearly focuses on displaying and visualizing graphs, not on modeling them. This part would have to be developed manually. Furthermore, the last activity on this project on github was 2 years ago.

### 3.2.5 Web Interface

The layout of the web interface is based on coreUI<sup>8</sup>, an admin panel template, built on top of the web toolkit Bootstrap<sup>9</sup>. A clean and nested structure of components, which is typical for a React app, guarantees modularity and forms the whole frontend application. One of the benefits when using React is that the data model displayed to

---

<sup>5</sup><http://www.bpmn.org>

<sup>6</sup><https://github.com/bpmn-io/diagram-js/issues/78>

<sup>7</sup><https://www.npmtrends.com/vis>

<sup>8</sup><https://coreui.io>

<sup>9</sup><https://getbootstrap.com>

the user maps most of the time nicely to UI components. Basically, the components of the application respect the single responsibility principle<sup>10</sup>. The main component has four sub-components - where each of them consists again of multiple sub-components - representing the core features of the app.

## Dashboard

The dashboard is the entry point where the user lands after accessing the app. The purpose of this component is to give the user a quick overview of the application, provide short informational texts and links to the specific modules.

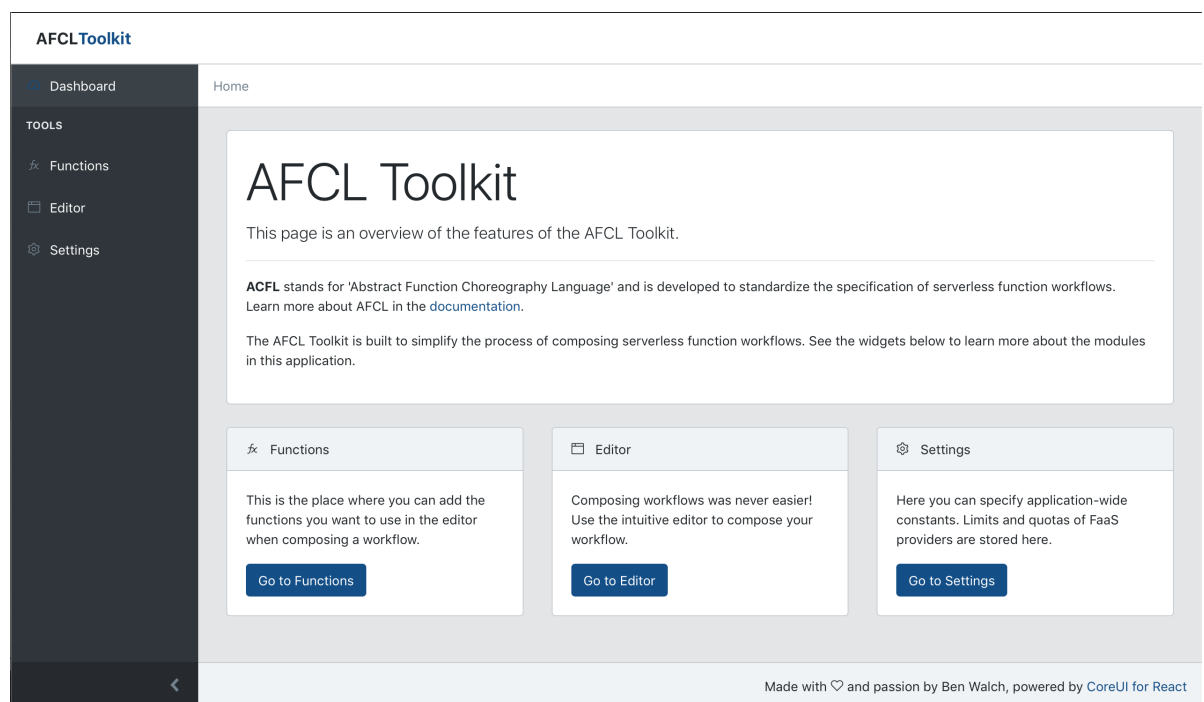


Figure 3.4: the dashboard

## Functions

A function repository is needed to know about available functions when composing a workflow. An item in the function repository holds the following data: **name**, **type** and

<sup>10</sup>[https://en.wikipedia.org/wiki/Single\\_responsibility\\_principle](https://en.wikipedia.org/wiki/Single_responsibility_principle)



provider. In the implementation of this thesis, this data is persisted to an internal file.<sup>11</sup>

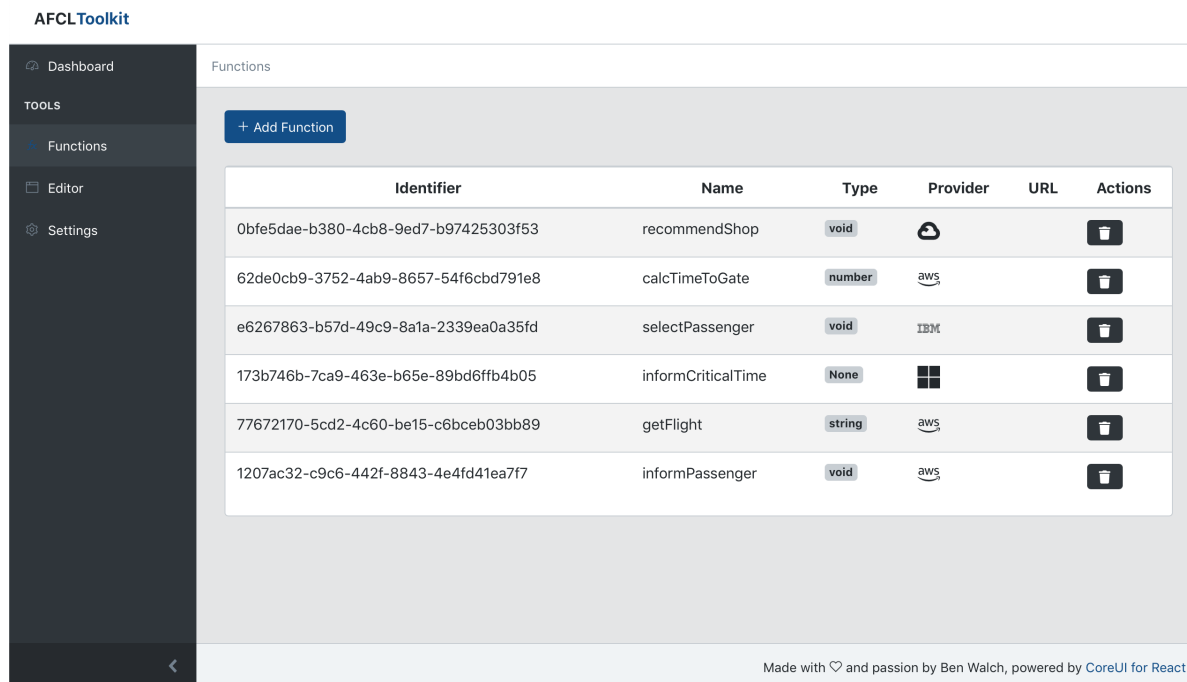


Figure 3.5: the function repository

## Editor

The interface of the component responsible for composing workflows is divided into two parts. On the left side is the editor, consisting of a toolbar and a drawing area, on the right side is the property view. In the editor, the user can choose a function from the toolbar to place it in the drawing area below. For the sake of clarity, each function has a defined shape and style. In the drawing area, these shapes (functions) can then be connected to each other by drag and drop. This leads to a directed graph, which represent the execution flow. The possible connection points (ports) of a shape will be displayed when hovering over it. These ports are input and output for most of the functions. When hovering a port, the port name appears in the tooltip.

When selecting a shape in the canvas, the property view changes. It displays each specific property of the selected function, with the ability to change, add or remove properties.

<sup>11</sup>see 3.3.4 for detailed information on how data is persisted

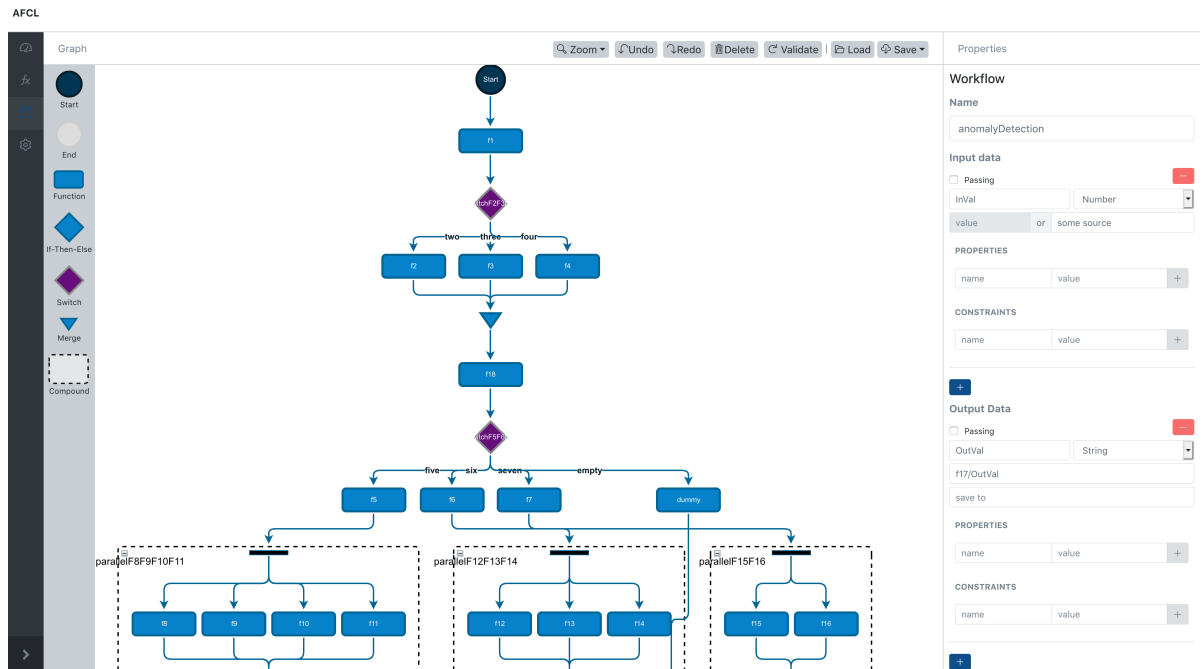


Figure 3.6: the editor

## Settings

## 3.3 Backend

### 3.3.1 Setup

Maven, Servlets

### 3.3.2 Api

RESTful (Functions) and RPC-Style (Workflows)

### 3.3.3 Decoding

XML is good because built-in XPath support in Java.

### 3.3.4 Persistence

Composed workflows are saved to the user's file system by offering a download. The user can then choose the target location. The storing of function data, on the other hand, is abstracted from the user and persisting of the data is handled internally. An interface<sup>12</sup> for the repository was created on the backend side to be capable of supporting any data source. The repository which implements the interface in this thesis, stores the serialized data into a file. This could be easily replaced with any other implementation, for example an implementation which stores the data in a DBMS.

```
1 public interface Repository<T> {  
2  
3     public Collection<T> findAll();  
4  
5     public T findOne(String id);  
6  
7     public void add(T obj);  
8  
9     public void remove(T obj);  
10  
11    public void remove(String id);  
12  
13    public void update(T obj);  
14 }
```

Listing 3.1: Repository Interface

### 3.3.5 Adaptation

---

<sup>12</sup><https://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html>









Cell	Description	User Object
 Start	Defines the workflow's entry point	String (for the label)
 End	Defines the workflow's termination	String (for the label)
 Function	An atomic function	AtomicFunction *
 If-Then-Else	A mutual exclusive condition	IfThenElse *
 Switch	A multi condition	Switch *
 Merge	An element for merging previous branches	null (None)
 Parallel	A container, where each child cell is meant to be executed in parallel.	Parallel *
 ParallelFor	A container, where each child cell is meant to be executed in a parallel for loop.	ParallelFor *

Table 3.1: Implemented Shapes, \*: AFCL Object

## 4 Evaluation



## **5 Conclusion and Future Work**





# Bibliography

- [1] *AFCL Documentation*. URL: <http://dps.uibk.ac.at/projects/afcl/files/documentation/index.html> (visited on 04/10/2020).
- [2] Paul Castro et al. “The Rise of Serverless Computing”. In: *Commun. ACM* 62.12 (Nov. 2019), pp. 44–54. ISSN: 0001-0782. DOI: 10.1145/3368454.
- [3] Erwin van Eyk. “The design, productization, and evaluation of a serverless workflow-management system”. 2019.
- [4] Cloudflare Inc. *What Is Serverless Computing?* URL: <https://www.cloudflare.com/learning/serverless/what-is-serverless/> (visited on 03/08/2020).
- [5] Ali Kanso and Alaa Youssef. “Serverless: Beyond the Cloud”. In: *Proceedings of the 2nd International Workshop on Serverless Computing*. WoSC ’17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 6–10. ISBN: 9781450354349. DOI: 10.1145/3154847.3154854.
- [6] JGraph Ltd. *mxGraph User Manual*. URL: <https://jgraph.github.io/mxgraph/docs/manual.html> (visited on 04/07/2020).
- [7] Nick Russell et al. *Workflow Control-Flow Patterns: A Revised View*. 2006. URL: <http://workflowpatterns.com/documentation/documents/BPM-06-22.pdf>.
- [8] Neil Savage. “Going Serverless”. In: *Commun. ACM* 61.2 (Jan. 2018), pp. 15–16. ISSN: 0001-0782. DOI: 10.1145/3171583.