Visualization for the ABS Modelling Language

Christopher Bauge

Thesis submitted for the degree of Master in Design, Use, Interaction 60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2018
Visualization for the ABS Modelling Language

Christopher Bauge
Abstract

Discovering defects early in the system development phase can have a big impact on the overall cost of developing a system. With Cloud Computing being the emerging trend in deploying large-scale systems, there can be significant economic and environmental benefits in utilizing more efficient resource usage.

ABS is an executable modelling language designed for distributed systems. The users of ABS are utilizing the language to design and model systems early in the development phase. Running ABS simulations can imitate how a given system would behave in deployment. In doing so, service providers could obtain greater control over their resource usage. ABS has at present time no official support for visualization of simulation data to its users.

Computer generated data is often nonintuitive to the human eye. By visualizing data we provide users with an additional tool to interpret the information at hand. In this thesis we propose a proof of concept application where ABS users can get their simulation data visualized. Furthermore, this thesis will expand on traditional information visualization principles, and how these can be incorporated into the application.
Preface

This thesis concludes my master studies at the Department of Informatics at the University of Oslo. It has certainly been both a fun and challenging experience working on this project.

First of all I would like to thank my supervisors Ingrid Chieh Yu and Jacopo Mauro for their continuous support, guidance and encouraging words throughout this project. I would also like to thank the developers of all the software used in this thesis, especially the ones working on ABS.

Lastly, I really want to thank my close friends, family and girlfriend for believing in me and encouraging me until the end.
# Contents

## 1 Introduction

1.1 Motivation ........................................... 3  
1.2 Report structure ..................................... 4  
1.3 Summary ............................................. 4

## 2 Background

2.1 Summary ............................................. 5  
2.2 Information Visualization ............................. 5  
  2.2.1 Visualization examples ........................... 5  
  2.2.2 Why are we visualizing? ......................... 8  
  2.2.3 Visualization principles ......................... 8  
  2.2.4 Visualization techniques ....................... 10  
  2.2.5 Distortion ...................................... 11  
2.3 Abstract Behavioral Specification (ABS) ................. 15  
  2.3.1 Distributed systems ............................ 16  
2.4 Cloud Computing ................................... 17  
2.5 Technologies ....................................... 18  
  2.5.1 Meteor .......................................... 19  
  2.5.2 Plotly ........................................... 20  
2.6 Related work ....................................... 20

## 3 Implementation

3.1 Summary ............................................. 23  
3.2 Requirements and considerations ......................... 23  
  3.2.1 Overview ...................................... 23  
  3.2.2 Pipeline ....................................... 24  
  3.2.3 Technology Stack ............................... 26  
3.3 Use cases and dataset ................................ 27  
3.4 Running the model .................................. 29  
3.5 Storing the data .................................... 32  
  3.5.1 Database ....................................... 32  
  3.5.2 Collections .................................... 33  
3.6 Creating the front end ................................ 35  
  3.6.1 Overview ...................................... 35  
  3.6.2 Visualization implementation ................... 36  
  3.6.3 Comparison view ............................... 38
List of Figures

2.1 Napoleon’s march visualized. Displays how his army was reduced during the invasion of Russia. Retrieved from [4]. 6
2.2 Florence Nightingale’s rose diagram displaying causes of death to British soldiers in the Crimean War. Retrieved from [6]. 7
2.3 Typical dashboard design implementation. Retrieved from [8]. 8
2.4 Preattentive processing using different colors. 9
2.5 Preattentive processing using different shapes. 9
2.6 Example of distortion by truncating axes. 11
2.7 The same dataset as in 2.6, but with correct proportions. 12
2.8 Proportional x-axis. 14
2.9 Example of distortion by stretching x-axis. 14
2.10 Additional context added to graph. 15
2.11 Examples of spot prices in AWS. Retrieved from [29]. 18

3.1 Overview of our pipeline. 25
3.2 Overview of implemented application - top section displayed. 25
3.3 Overview of implemented application - showing bottom section. 26
3.4 Delete simulation dropdown with list of available simulations. 26
3.5 Object nesting in Model API. 28
3.6 All dropdowns in comparison view. 37
3.7 Comparison view section. 39
3.8 Comparing data on hover. 39
3.9 Filter plot type in comparison view with range slider. 41
3.10 Implemented view of component parameters together with plot. 42
3.11 Displayed object state view before any plot trace has been clicked. 42
3.12 Subplot view section. 43
3.13 Subplot dropdown component. 43

4.1 Comparison of three datasets with superposition design in the HyVar example. 52
4.2 Comparison of two datasets with superposition design in the HyVar example. 52
4.3 Comparison of datasets in subplot view with juxtaposition design. Here we see the New Year’s Eve example. 53
4.4 Comparison of datasets in subplot view with juxtaposition design. Here we see the New Year’s Eve example.

4.5 Measuring the Lie Factor in comparison view. Plot data is instances in time from the c compiler component.

4.6 Measuring the Lie Factor in subplot view. Targeted plot data (red graph) is pending in time from the c compiler component.

4.7 Reduced plot aspect ratio, data is pending in time from the c compiler component.

4.8 Full plot aspect ratio with same dataset as in Figure 4.7.

C.1 Implemented scatter plot.
C.2 Implemented overlay plot.
C.3 Implemented bar plot.
C.4 Implemented histogram plot.
C.5 Implemented filtering plot.
C.6 Implemented scatter plot with dotted lines mode active.
C.7 Implemented subplot.
C.8 Implemented detailed object view.
C.9 Fetching data from model-API and save to database.
C.10 Deleting simulations from API.
List of Tables

3.1 Specifications of VMs used to run ABS simulations. . . . . . 29
3.2 Parameters passed to our render function. . . . . . . . 40
## Listings

2.1 HTTPName annotation. ............................................. 16  
2.2 HTTPCallable annotation. ........................................ 16  
2.3 Query in MongoDB shell. Data returned as BSON/JSON objects. ................................................. 20  
3.1 Setting path to ABS compiler. ................................. 29  
3.2 Compiling the ABS model. ....................................... 29  
3.3 Running the Cowboy web server. ............................... 30  
3.4 Console output from ABS simulation. ......................... 30  
3.5 Detailed state of the hyvarrec component after finished simulation. ................................................. 31  
3.6 List of API endpoints. ........................................... 32  
3.7 Defining and exporting our collection. ....................... 34  
3.8 Subscribing to our collection in Meteor. .................... 34  
3.9 Rendering a new plot using Plotly. ............................ 36  
3.10 One of the implemented rendering methods. ............... 37  
3.11 Provided plot types as defined in graphs.html. .......... 40  
3.12 Filtering the data and pushing it to our trace object. ..... 41  
3.13 Updating reactive variable with coordinates and the object state when clicked on a plot trace. ....................... 42  
3.14 Adding our coordinates and anchoring the x-axes to their corresponding y-axes. ................................. 44  
3.15 Defining template views. ...................................... 45  
3.16 Rendering our template. ....................................... 45  
3.17 Declaring our reactive variables before rendering the webpage. ...................................................... 46  
3.18 Using template events to delete database entries. ....... 46  
3.19 Listening for events on our template elements. .......... 47  
A.1 Script to install the application. .............................. 67  
B.1 Description on how to run the application. ................ 69
Part I

Introduction
Chapter 1

Introduction

1.1 Motivation

Abstract Behavioral Specification (ABS) is an executable modelling language designed for distributed systems. The users of ABS are utilizing the language to design and model systems early in the development phase. Running ABS simulations can mimic how a given system would behave in deployment. This can result in service providers being able to maintain greater control over their resource usage. ABS has at present time no official support for visualization of simulation data. My project advisers introduced me to the ABS modelling language, and the lack of visualization was an apparent problem for its users.

At present time, ABS users have to implement their own visualizations and tailor them to fit each individual project. This is mainly done by importing third party libraries and modifying its source code to fit their existing data structures. This is a time consuming task, and it would be beneficial for ABS users to create their models and easily transform its simulation data into visual representations. Creating a general purpose application which supports ABS simulation data can facilitate users to focus their resources on creating precise models for their projects.

Research question

The primary goal of this thesis is to facilitate users of ABS with a general purpose application that offers visual representations of data. We seek to answer the following research question:

- How can we facilitate the interpretation of data to ABS users by utilizing visualization?

To answer our research question, we need to get insight into why the ABS language was created, who their users are and what our constraints are. We are proposing a general purpose visualization application, and
do not seek to solve all the niche problems ABS users are experiencing in their projects. The goal is to provide ABS users with a flexible and easy deployable application where they can store and visualize data.

1.2 Report structure

The thesis includes five different chapters:

• **Chapter one**: Introduces the motivation behind this thesis. The research question to be answered will be provided.

• **Chapter two**: Contains relevant background information on the principles we apply in this thesis. Information visualization as a concept is explained, and examples of traditional visualization use-cases and techniques are provided. The ABS language is also introduced, together with a brief overview of Cloud computing and distributed systems. Lastly, we provide an overview of the technologies used to create the visualization application.

• **Chapter three**: Explains how each step of the application pipeline is implemented in detail. The reasoning behind our technological choices is also introduced.

• **Chapter four**: Presents and analyse the results against our research question.

• **Chapter five**: Concludes and evaluates the project, while also investigating some future work that can be performed.

1.3 Summary

In this thesis, we start by giving an introduction into the modelling language ABS, why it was made, and explain how visualization can aid its users in gaining more insight into their project data. We will show some examples of traditional visualization techniques, and some of the pitfalls that may occur. Later on, we will implement an application that is suitable for developers using ABS.
Chapter 2

Background

2.1 Summary

In this chapter, we will first introduce information visualization as a term while providing examples of historical representations of data. Further on we will discuss various visualization techniques and principles that will be applied to our thesis. We will also provide some insight into the ABS modelling language, Cloud Computing and distributed systems. At the end of the chapter, we will introduce the most relevant technologies used in this thesis.

2.2 Information Visualization

Visualization can be defined as

"The act or process of interpreting in visual terms or of putting into visual form" [1].

or as

"The formation of a mental image of something" [2].

Visualization of information has been done for many hundred years, hence this is not a concept restricted to computer science. Traditionally this has been a human activity, as shown for instance in Figures 2.1 and 2.2. In this thesis, we will be working specifically with information visualization, which bearing in mind our previous definition is a way to create mental models of information.

2.2.1 Visualization examples

To demonstrate how information visualization can be used to convey complex data, we will introduce some examples where it has been used
Napoleons March

A classic representation of information is Charles Joseph Minard’s map of Napoleon’s march during the French invasion of Russia as shown in Figure 2.1. This representation consists of multiple datasets, all nested together in one beautiful graphic illustration. The representation provides information of how the size of Napoleon’s army was reduced while crossing rivers, facing freezing temperatures and combating Russian soldiers en route to Moscow. Minard’s chart is by data scientist and professor Edward Tufte regarded as

"... the best statistical graphic ever drawn" [3].

Nightingales Rose Diagram

Another famous visualization example was created by the famous nurse Florence Nightingale which in 1859 released a statistical overview on reasons of death to British soldiers during the Crimean War [5]. Nightingale was convinced that poor sanitation and conditions in hospitals were causing soldiers to die from infectious diseases and under-nutrition. This was the background for the diagram in Figure 2.2. In the diagram, the blue outermost sections of the wedges represent all the soldiers which died from treatable diseases while hospitalized. The black middle sections represent soldiers dying from other reasons and the red innermost wedges show soldiers that died from wounds. Each wedge represents a month in the war that went on. Needless to explain, from this diagram alone we get an immediate understanding of the underlying reasons for the British death tolls in the Crimean War. Nightingale’s discoveries is believed to have been a
significant motivator to the improvement of sanitation in the period following the war.

Figure 2.2: Florence Nightingale’s rose diagram displaying causes of death to British soldiers in the Crimean War. Retrieved from [6].

Visualizations in modern times

In more recent times, visualization has become a trending tool for decision-makers and business analysts to assist in interpreting large and abstract sets of data. Dashboards are often used in enterprises as it requires little prerequisite knowledge or effort to utilize. A dashboard can be defined as

"... a user interface that, somewhat resembling an automobile’s dashboard, organizes and presents information in a way that is easy to read" [7].

Figure 2.3 displays a typical dashboard software implementation. We have this design in mind when constructing our application as we want most of the data to be available to our users at all times.
2.2.2 Why are we visualizing?

A paper introduced by Fekete et al. [9] in 2008 discusses the value of Information Visualization. In the paper, they argue that measuring the concrete value of information visualization is difficult because it seldom results in any tangible metrics. Interpreting data is in large part a human activity, where the goal is primarily to gain insight into a field of interest or a complex dataset. This kind of research stands in contrast to traditional mathematical algorithms where attributes such as performance, robustness, and efficiency can be analysed. Measuring human activities like insight and degree of understanding is more complex.

In the same paper Fekete et al. also argues that visual representations strengthen our memory capacity when it comes to performing operations on data. Solving complicated mathematical equations is more difficult in our head than with a visual aid such as pen and paper.

Working on complicated sets of data generated by computers is not a task which is natural for humans, and in such way, visualization builds a bridge in the gap between computer-generated data and the information we want to derive. Visualizations can also assist us in forming hypotheses and help us to see connections outside of the intended scope.

2.2.3 Visualization principles

Preattentive processing

Preattentive processing is a subconscious way of identifying patterns in our brain [10]. Examples of these patterns are color, hue, size, and shape.
Figure 2.4 demonstrates how adding color to one of the circles in the chart instantly draws our attention to that element.

Another way to draw our attention to certain elements is by using different shapes. The example in Figure 2.5 is somewhat more difficult to process, however the same principle applies. Different shapes draw our attention to a specific element, and we do not have to scan the entire field of view to find the anomaly.
Coloring

As mentioned earlier in the section, adding color to certain elements enables us to quickly recognize patterns. However, there are principles one should adhere to when choosing colors in visualizations. When creating visual representations we have to be aware of certain attributes that alter the way we process information. How we differentiate datasets using colors can have a significant impact on the efficiency of our perception. When displaying categorical data it is a good practice to use colors that are easily differentiated. Visualizing ordered data often works better by applying light-to-dark progressed hue values.

2.2.4 Visualization techniques

When designing visual representations, there are several factors to consider [11]:

- **Data type**: Working with numerical values (e.g. age, price, time) is very common. However there may be other forms of data to consider, such as categorical (e.g. dog breed, car type) and ordinal (e.g. size, chronology).

- **Data dimension**: When representing data, there may be variations in the number of attributes we have to consider. Designing a representation for a single variable is often easier than having to factor in multiple attributes.

- **User aspect**: Representations are designed towards human users. While the individual interpretation of representations may be trained and conditioned, people will sometimes perceive the same object in a different way.

As earlier mentioned visualizations can act as a cognitive aid in our way of understanding data [9]. Visual representations help our brain process larger sets of data and function as a temporary storage area. A study on visual comparison for information visualization categorizes the comparison of objects in three ways [12]:

- **Juxtaposition** Objects are separated either by time or space. An example of this would be to have two simulations with related datasets displayed side by side. By doing so users must actively shift their attention between the objects to see patterns and create connections.

- **Superposition**: Objects are placed in the same confined space and time. An example would be to display multiple simulations of the same type of dataset within the same plot. This relies on the use of different visual attributes in the graph to distinguish the objects.
• **Explicit encodings**: A way to calculate and display relationships between the data. This could for instance be the continuous difference between two values of different datasets in a graph \((Y_1 - Y_2)\).

It is possible to combine methods between all these categories, and many visualizations also operate in the range between juxtaposition and explicit representations [12]. However, in this thesis we will only be operating within the juxtaposition and superposition categories.

### 2.2.5 Distortion

When visualizing data, we may unconsciously introduce side-effects to our representations. **Distortion** is a technique to modify the representation of data using some set of boundaries [13]. This is often done to fit visual representations of complex data into small screens or devices. However, adjusting the scales or modifying some display parameter may cause the data to be wrongly interpreted.

We will present a few common examples where distortion can occur:

**Axis truncation and the Lie Factor**

Modifying the axes of a graph may result in a significant alteration of the representation of data, and how it is interpreted. These modifications are either introduced unwillingly or they are used to purposely highlight or exaggerate pieces of information. Consider the examples shown in Figures 2.6 and 2.7.

![Distortion example - truncating axes](image)

**Figure 2.6**: Example of distortion by truncating axes.

Here, we have a dataset with two given values \((80, 85)\). The numerical difference between these values is:

\[
85 - 80 = 5
\]

By using a mathematical formula dubbed by Edward Tufte as the Lie Factor [3], we can demonstrate how misleading the chart in Figure 2.6 actually is to the viewer. The Lie Factor is calculated as follows:
To find the size of effect shown in the graphic we look at our example charts. The graphical values are retrieved using eye measurements on the horizontal grid-lines in the plot. Each value represents an estimated number of grids. By utilizing the horizontal grids in Figure 2.6 it is evident that the rightmost bar extends over exactly three grid lines. The bar to the left spans over only two.

Given this, the graphical relative difference is calculated as:

\[
\frac{3 - 2}{2} \times 100 = 50\%
\]

In other words, the numerical difference of 6.25% is represented by a graphical increase of 50%. By applying the measurements to the Lie Factor formula, we find the Lie Factor to be 8% which is exceptionally high. According to Tufte, any values outside a range of 0.95 - 1.05 indicate a level of distortion introduced beyond the scope of plotting inaccuracies.

The excessively high Lie Factor shown above has been introduced as a result of zooming into the graph, thus changing the displayed range of the y-axis. From this, we can interpret that changes to the displayed y-axis will result in a different representation as the relational difference between the graphs changes along with it.

In Figure 2.7, the y-axis instead is set to zero, displaying its full range. By using the horizontal grid lines again, we get the approximate relative graphical difference between the bars:

\[
\frac{17 - 16}{16} \times 100 = 6,25\%
\]

Considering the numerical increase from before:
This results in a Lie Factor that is:

\[
\frac{6,25}{6,25} = 1.00\%\]

We can now see that the Lie Factor is within the recommended range, making the graph proportional. The distortion in Figure 2.6 was introduced by only zooming into the graph which in itself seems like a harmless action, but for a user with little or no information about the context of the data may significantly alter his or her interpretation of the dataset.

Tufte argues that the ambiguity of data can increase when the full context and scope of the data are not present. Therefore, it is a good practice to supplement the visualizations with values from the actual dataset. In doing so the user can validate the integrity of the graph.

**Axis stretch**

The effect of interpreting data where the x-axis has been modified can also be viewed. In the previous section, we explained how changes to the section displayed may alter how the data is interpreted. We also demonstrated how changes to the y-axis could be used to highlight or transform the perception of data. However, the x-axis is somewhat more tricky to manipulate as we have to seclude data from the set if we were to focus on specific sections of the graph. Nevertheless, it may introduce some of the same side effects as manipulating the y-axis. This is not often something we would want to do, as the context of the data is reduced. Changes to the aspect ratio of a display may lead to different interpretations of the same data. In Figures 2.8 and 2.9, two identical datasets are displayed, only with different aspect ratios.

Comparing Figures 2.8 and 2.9, we can see differences in the way the data is represented when stretching the x-axis. From the graph legend, it is shown that from one year to another, the price of bananas doubled in price. From 2014 the price increased from $4 to $8 US Dollars. In the first example from Figure 2.8, this is indicated as a rather impactful change, while in Figure 2.9 the same growth gives the impression of being more moderate. If we did not have the legend data available, the two graphs could be misinterpreted as having completely different values, or at least not having the same impact.

In the previous example, there are also more factors to consider. Since the dataset only represents the relative difference between two given years, the context of the graph is very hard to interpret. There is little or no way of knowing if the price growth of both apples and bananas are as expected
as the only points of reference are two years. Consider a new graph with additional context to the graph as shown in Figure 2.10.

Here we can see how our previous dataset (years 2014 - 2015) fits in a long-term context. From this data, it is safe to assume that the price drop in 2014 was more of an anomaly, and the subsequent growth which in Figure 2.8 seemed dramatic was merely a correction in the market.

We have now demonstrated how creating honest representations of data demands accurate modelling as minor inaccuracies to seemingly trivial attributes can introduce undesired side effects.
2.3 Abstract Behavioral Specification (ABS)

ABS was first introduced in 2010, as an executable modelling language designed for distributed object-oriented systems [14]. ABS has provided a detailed language reference, and its syntax resembles that of Java which is familiar for many developers. The language is suited for distributed modelling as it utilizes asynchronous method calls, immutability, safe concurrency, and encapsulation. For easy testing purposes, a collaborative environment is available via the project’s home page [15]. The Collaboratory offers users the possibility to get familiar with the language without any local installation needed. Here, users have the possibility of creating and running their own models in ABS or expand and execute already provided examples.

ABS was designed to aid deployment decisions by moving them up the development chain [16]. The ABS language is for instance incorporated in the Envisage research project, which addresses issues regarding resource management in the early stages of development [17]. In the beginning development phases, it can be hard to accurately predict the volume of computational resources needed when deploying at later stages, and ABS aims to solve this problem.

By running ABS in the early development phase, modellers can mimic the intended behaviour of a system. By setting certain thresholds and parameters, users can obtain realistic information on how the system scales up and down to meet the project requirements. Often these requirements have certain metrics that should comply with a given Service Level Agreement (SLA). Results of the simulations can be accessed directly as generated log files. However, it can be a tedious task for users of ABS to interpret these log files as they are generated for computers to read, and not the human eye.

To make the data more accessible, ABS also offers the possibility of getting simulation data exposed as JSON objects via the Model-API. The user can specify which objects or methods they want to be exposed by using annotations in their model. Since JSON objects use a data structure well
suitable for JavaScript, it enables us to create web interfaces to work with ABS data.

To get objects accessible via HTTP requests, `HTTPName` annotation on variable declarations, assignment statements and also `new` expression statements are used as shown in Listing 2.1.

```java
[HTTPName : "Volkswagen"] = new Car();
```

Listing 2.1: HTTPName annotation.

Exposing methods is done by adding a `HTTPCallable` annotation when defining an object through an interface which is demonstrated in Listing 2.2.

```java
interface Vehicle {
    [HTTPCallable] String getOwner(String regNr);
}
```

Listing 2.2: HTTPCallable annotation.

By visualizing the simulations, there is potential for users to get an intuitive overview of how changes in the model affect the performance of the system. Currently, there is no official framework in the ABS environment that offer visual representations of generated data.

In this thesis, we do not perform any actual modelling in ABS, but instead, visualize different models in already existing projects.

### 2.3.1 Distributed systems

Considering ABS is a modelling language which targets distributed systems, we provide a brief introduction to the characteristics of a distributed system.

A distributed system can be defined as

> "A system in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages" [18].

This is a very shallow definition which today covers almost every interaction between systems.

More specific, distributed systems enable the use of resource sharing. The Internet is one example of resource sharing, where users all over the globe can access the same content regardless of their physical location. On a smaller scale, an Intranet is a private network that enables users to access resources such as the same printer, file server, forums and so forth. Coulouris et. al [18] describes a set of challenges for constructing distributed systems. Some of them include:

- **Concurrency**: Issues might occur where shared resources are being accessed at the same time. Coulouris et. al. provides the example of
an auction, where two bidders in a concurrent environment access a bid record at the same time. If the right concurrency measures are not implemented, the amount of the two bids might be altered based on the sequence in which the bids get processed.

- **Scalability**: A system is considered to be scalable if a significant increase in resources spent or active users does not affect its efficiency. When dealing with the scalability of resources, we often think of horizontal and vertical scaling as described in Section 2.4. When constructing a distributed system, one must take into account the cost of physical resources, performance loss and bottlenecks in performance.

These are only some of the concerns to have in mind when designing distributed systems. The ABS language contains implementations that address these issues. This provides an insight into what kind of problems ABS aims to solve and to get an understanding of who their users are. It is important to note that a normal ABS-user is indeed a developer, who should have basic knowledge of coding and developing a system.

### 2.4 Cloud Computing

Cloud-based services have seen a steady rise in recent years, with almost half of all Norwegian enterprises buying cloud computing resources [19]. Shifting away from on-premises systems to virtual machines has shown to be attractive for enterprises as developing software can be deployed without major hardware investments up front [20]. Cloud computing services are being provided by large companies like Google [21], Amazon [22], Microsoft [23] and IBM [24]. These companies offer services such as computational resources, storage, networking, databases and more. The cloud computing services are often quite flexible and can be re-provisioned within minutes. The services provided are more often than not subscription based, and comes with an hourly or monthly fee [25]. Providers such as AWS (Amazon Web Services) also offers the possibility to pay spot price for instances as shown in Figure 2.11, meaning customers will only pay the spot price that is set by AWS during the period they are running their instances. Spot prices are usually a lot cheaper than renting regular On-Demand instances as the providers are only renting out surplus resources. Common for these service providers is that they enable their users to utilize cloud elasticity. Two common features in cloud elasticity are scaling their resources horizontally (add/remove instance) or vertically (reassigning resources to already instantiated Virtual Machines (VMs)) [26].

With Cloud Computing emerging as the dominant trend in deploying large-scale systems, there are possible economic and environmental benefits yet to be discovered in pursuing effective resource usage [27, 28].
Buying virtual computational resources clearly differ from the traditional approach where you pay for hardware up front. Usually, buying hardware means larger initial investments, which makes it hard to accurately predict resource usage down the line [30]. It can therefore, be difficult to justify the investment costs at the early stages of development. Having the option to rent resources when required, and also being able to scale the instances should in an ideal world enable you to not pay for unnecessary resources. However, this is not always the case. Resource usage is seldom constant, and the system must be able to dynamically react to changes. Traditionally, businesses approach this type of scaling after the developed system is deployed. Modelling languages such as ABS aim to tackle these scaling decisions earlier in the project cycle.

### 2.5 Technologies

We have already introduced the ABS programming language as our point of reference for this project. To implement a visualisation application which supports ABS simulation-data we use JavaScript-oriented technologies. Technologies utilized in the project will be introduced in this section.
2.5.1 Meteor

Meteor is an open source full-stack JavaScript platform for creating dynamic web and mobile applications [31]. It had its first release in 2010, however, companies were already using the platform before its mainstream release [32]. Creating real-time web applications makes sure that changes to a server will be conveyed to an application instantly. The user will not have to refresh the web application for the changes to be reflected in their view [33]. Meteor integrates well with MongoDB which handles the storage of data. This will be introduced further in Section 3.5. Meteor also comes with a good selection of built-in packages which makes building and modifying an application more convenient. Since Meteor bundles all the pieces of a traditional web-application, it enables developers to create complete web applications in a short amount of time. The developer does not need vast programming experience within each of the application components to create a working environment. Web applications created using Meteor are implemented by Mazda, Dispatch, Codefights and others [34].

MongoDB

MongoDB is a non-relational (NoSQL) database which stores data in collections. Collections can be viewed as a grouping of records, which in MongoDB is called documents. Documents are stored as Binary JSON (BSON). BSON is an extension of the JSON model, supporting additional data types. MongoDB does not require any specific document structure to what is inserted, and therefore it is the developer’s responsibility to create logical references where needed. This is in contrast to relational databases which uses schemas to define how the data must be structured. Since MongoDB is bundled into the default Meteor installation, it became our database flavour of choice.

MongoDB also comes with a shell where users can access a running database and query its collections. A typical query is displayed in Listing 2.3.

MongoDB also provides support for a variety of Create, Read, Delete and Update (CRUD) operations. Read operations such as aggregation and count are available just like in a traditional relational database system. Scaling the database can be done by utilizing sharding (distributing partitions of a database) [35].
2.5.2 Plotly

Plotly is an open-source visualization framework with support for programming languages such as JavaScript, MATLAB, R and Python. Their dashboard solution Dash is a Python framework which enables the building of web applications that can be hosted on their cloud. They provide their own cloud solutions for hosting Plotly applications and dashboards. Plotly.js which is utilized in this thesis is an open-source library containing more than 20 chart types [36]. We will elaborate further on what Plotly provides this thesis in Section 3.2.3

2.6 Related work

There is much research done on the implementations of domain-specific visualization tools. These implementations are often tailored directly towards the task at hand, however since we are proposing a general-purpose application, we have to look at the underlying mechanisms of what a good visualization is. Traditional examples as shown by Tufte [3] demonstrates the beauty and intuitiveness displayed by a successful visualization, but not so much why the representation is good. Yi et al. [37] address some of these topics, and identifies different patterns in the way insight is obtained into a dataset. They argue that operations such as selecting, filtering and aggregation can assist users in exploring a dataset.
Heer & Shneiderman [38] discusses a taxonomy of tasks that should be addressed in order to create visualization tools.

Kibana [39] is an open-source visualization tool designed for ElasticSearch, and is a popular tool for analysts. Together with Logstash, they form the ELK stack which has gained popularity over the recent years. It is primarily used for managing logs and analysing data. Kibana has been implemented at the INFN-Torino computing center as a scientific experiment to monitor their Cloud resource usage [40]. Atlas EventIndex is another research experiment where they use Kibana to monitor and evaluate their systems [41]. Gómez et al. [42] propose an implementation of Kibana and ElasticSearch where students can get visual representations of relevant, available job offers. Common traits for applications built with Kibana is that they often require the construction of queries to visualize data. We are proposing an application where the entire dataset is displayed directly from our origin, and where the user performs filtering on the representation.
Chapter 3

Implementation

3.1 Summary

In this chapter, we describe how our visualization application is designed and implemented. The datasets and models which are used will be introduced. We also provide the reasoning behind our technological and architectural choices.

The full source code of the application would be too comprehensive to fit into this report, so it is made available in a public GitHub repository: https://github.com/baugztar/visualization-abs

The steps needed to run the application is provided within the repository description.

3.2 Requirements and considerations

3.2.1 Overview

Before creating the application, we had to consider the following requirements:

- Simulations should be stored in order to enable comparison of data.
- The application needs to communicate with the ABS model-API which returns data represented as JSON objects.
- The web interface needs to be dynamic and react to changes in our database.
- The plot implementations must adhere to established visualization techniques.

For the specific use cases we are working on, and as a data visualization tool in general, these requirements should be met in order to answer our
research question. In this chapter we will explain how meeting each of these requirements results in a solution that is targeted specifically towards ABS. The application will primarily be tested towards the HyVar dataset, but also against a second example to ensure flexibility.

**Context**

Seeing that ABS is an executable modelling language, there could be significant value in running multiple simulations with different parameters. Consider an example where Business A has an SLA ensuring its customers a maximum one-way delay of 150 milliseconds (ms) while using their services. Suddenly, changes in the SLA requires the one-way delay to not exceed a 90 ms threshold. When modelling this system, it now requires a change in parameters to comply with the SLA. Comparing the running of these simulations should be beneficial in seeing how the entire system reacts to the changes introduced. Maybe the changes in latency appear to have little influence on the already robust system, or perhaps it drastically alters the amount of VM’s needed in order to meet the threshold. The thesis therefore, aims to introduce a feature where users can compare two or more simulations, to help identify how changes to model parameters are reflected in the system.

Having precise models that are aimed towards cloud deployment can enable service providers to be more efficient as it reduces the surplus of computational resources. The research conducted in this thesis is based on the assumption that there is a demand for visualization support to the ABS language.

**3.2.2 Pipeline**

The finished implementation of the application is presented in Figure 3.1. When a client is running an ABS simulation, the data annotated in the model by the user will be exposed through an API running on Cowboy [43], which is a lightweight web server for Erlang. From here our application is free to fetch the desired data through standard HTTP requests. The data is in turn stored to a local MongoDB database associated with a Meteor instance. The collections of data stored are then accessible to the application and is free to transformed and represented in the front end. Users of the application are subscribing to all the data stored in the database, meaning that when a new simulation is either inserted or deleted, the view is reactively updated to display the changes made.

This only describes a high-level representation of how the application is wired together. Figures 3.2, 3.3, 3.4a, and 3.4b shows examples of the implementation rendered to a client. Further on in this chapter, we will explain how each of the components is implemented and how they interact.
An overview of the implemented UI is demonstrated in Figures 3.2, 3.3, 3.4a and 3.4b.

![ABS visualizations]

Figure 3.2: Overview of implemented application - top section displayed.
3.2.3 Technology Stack

Application platform

For simplicity purposes, we wanted to implement a web-server communicating with a database on the same domain. Again, we had to consider that our choice of technologies is open-source. When selecting a stack, we initially tested the popular MEAN-stack (MongoDB, Express, Angular, Node) [44]. The MEAN-stack should cover the use-cases needed for this thesis, however, there was a lot of tedious wiring needed to be done for the stack to be incorporated into our project. Another alternative was to use the LAMP stack [45], which consists of Linux, Apache, MySQL and PHP. This however, is a heavier stack than what we wanted, and also requires more initial configuration to fit our project scope.

The JavaScript platform Meteor was deemed a preferred option as the functionality required was provided to us out of the box. Setting up a working web server which communicates with a local MongoDB database was completed only minutes after the initial installation. Meteor also provides us with the possibility of subscribing to a database. Traditional web applications rely on HTTP requests and responses between the client and server, but Meteor takes advantage of their introduced Distributed Data Protocol [46] to re-render UI templates when data sources changes. Subscriptions are a part the publish-subscribe pattern [47], where any changes to the data sources of interest are transmitted to the client and

![Figure 3.3: Overview of implemented application - showing bottom section.](image)

(a) Delete dropdown element  
(b) Collapsed dropdown element

![Figure 3.4: Delete simulation dropdown with list of available simulations.](image)
instantly reflected in the User Interface (UI). In this project, this gives us the freedom to update the UI when an ABS simulation is inserted into the database. Taking into account that simulations often take considerable time to finish, this should be a very beneficial feature as we don’t have to wait around for the simulation to actually complete. When we are confident that the simulation is done, we can fetch the data without having to refresh the entire web-page. We can therefore, be confident that the data currently displayed in the view is a reflection of what is stored in our database.

Visualization-API

When creating our application we did not seek to re-invent the wheel and create visualizations entirely from scratch. There are a large number of available software that should provide us with the tools needed to answer the research question. When choosing the right technology stack, we had to consider that the software is open-source. Bearing this in mind, we ended up using Plotly as our visualization tool [48]. More specific we are using Plotly’s JavaScript library (Plotly.js), which builds upon d3.js [49], and stack.gl [50]. Plotly.js supports basic charts such as scatter plots, line charts, and bar charts. More sophisticated models such as 3D Charts, heat maps and animations are also available. However, the basic features first mentioned should be sufficient for the data we are working within this thesis.

As mentioned in Section 2.3, the data we wish to represent from the ABS simulations is accessible as JSON objects. This is why we decided to use the JavaScript library from Plotly, as it makes JSON objects easy to work with. It is also extensible to our Meteor platform and integrates well with both our front- and back end.

3.3 Use cases and dataset

To demonstrate how visualization in ABS can be utilized, we chose to work on the HyVar-project [51] and its dataset for this thesis. HyVar is a framework developed for devices with Electronic Control Units (ECU). We focus on a specific use case where the HyVar system is deployed on a large set of cars [52]. The HyVar cloud-system is in charge of gathering information from the cars, and if changes to the software are required, the system should compile an update and send it back to the car. Compilation and deployment of the updates are performed by a set of individual microservices deployed on the AWS cloud. When modelling systems in ABS, we take into consideration that the scaling approaches are coherent with a given SLA. The HyVar data is based on real traffic history and provides us with a realistic experiment.

The data which has been annotated by the modeller and exposed to the Model-API is each of the components in the HyVar chain. The output
of the simulation will show the state of each component throughout the simulation process. By visualizing this dataset we get a detailed overview of how the different components react to changes in traffic and the parameters specified through the model implementation.

To further expand on our use cases, we have also tested the application against the New Year’s Eve example from the collaboratory environment [53]. This example is intended to demonstrate how two servers react to the alternation of a surge of text messages and phone calls made by its clients. This is an event which is likely to occur around midnight on New Year’s Eve.

The project was modified by adding annotations (as shown in Listing 2.1) to the components we wanted to monitor. This gives us two complete and independent projects to monitor against our application.

Since the data structures in the Model-API are created by the individual modeller, there will certainly be variations between different models. We therefore, want to introduce an application where we only have to adhere to simple principles in order for the visualizations to work.

The primary requirement is that data to be visualized is accessible as arrays. Proper nesting of the component data must also be applied. Figure 3.5 shows the JSON object nesting required for our visualizations to work.

![Figure 3.5: Object nesting in Model API.](image)

In the HyVar use case, the simulations are quite large and can take several minutes to complete. This tells us that there is a need to store the simulations in a database for later access. Having data stored omits the
need to repeat simulations every time we want its data displayed. Storing
the simulations also facilitates the possibility of comparing previously
generated data. Consider a project where some requirements have changed
and we wish to re-run the model with different parameters. Having
data from previous simulations stored enables faster comparison than re-
running both simulations every time.

### 3.4 Running the model

In this section, we will explain how we run an ABS-model together with
the Cowboy web server, and how we transfer the simulation data into our
application. For reference purposes it is noted that the ABS-simulations
were performed on a VM with specifications shown in Table 3.1.

<table>
<thead>
<tr>
<th>RAM</th>
<th>8 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCPUs</td>
<td>4 VCPU</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 16.04</td>
</tr>
</tbody>
</table>

Table 3.1: Specifications of VMs used to run ABS simulations.

When running the HyVar ABS-model we are utilizing the Erlang backend.
The model is first compiled into Erlang combined with a small runtime
library which contains key ABS features such as Concurrent Object Groups
(cogs) [54], futures, objects and method invocations. In Section 2.3 we
explained how the Model API exposes objects and methods. When an
Erlang compiled ABS model is executed with the port parameter `-p`, a
Cowboy web server is started, listening on the specified port. The state
of our exposed methods and objects are now accessible as JSON objects.
Since this data was explicitly defined in the model, this is the information
that should be relevant to the user. Therefore this is the data we will be
working within our examples.

To run the ABS-compiler on our VM, we first need to set the environ-
ment variable to match the compiler bash-script. In our Ubuntu test-
environment, this is done by running the command shown in Listing 3.1.

```bash
export PATH=$PATH:~/my_dir/abs/abstools/frontend/src/bash/
```

Listing 3.1: Setting path to ABS compiler.

After setting the correct path, we are now able to compile the model and
start our web-server using the `absc` command. Running the sequence as
shown in Listing 3.2 compiles all the files with `.abs` file extensions in our
current directory.

```bash
ubuntu@master-abs:~/master/abs_optimizer/abs_model$ absc -erlang *.abs
```

Listing 3.2: Compiling the ABS model.
Then we can run the web server, with the sequence shown in Listing 3.3:

```
$ gen/erl/run -p 8084 -l2000
```

Listing 3.3: Running the Cowboy web server.

Running this sequence on the HyVar example results in the output shown in Listing 3.4.

```
1 gen/erl/run -p 8084 -l2000
2 Starting server on port 8084, abort with Ctrl-C
3 aux,switch_time_slot,250
4 aux,initial_instances_list,list([1, 1, 1, 1, 1, 1, 1])
5 aux,instance_cost_list,list([30, 1500, 30, 500, 30, 30, 33500, 2000])
6 aux,scale_in_threshold_list,list([2000, 94, 2000, 80, 2000, 2000, 448, 400])
8 aux,instance_speed_list,list([500, 500, 500, 500, 500, 500, 500, 500])
9 aux,instance_init_time_list,list([0, 906, 0, 762, 762, 762, 762, 762])
10 aux,drop_requests_list,list([0, 0, 0, 0, 0, 0, 0, 0])
11 scale_in,0,encoder
12 scale_in,0,hyvarrec
13 scale_in,0,resolution
14 scale_in,0,variant_gen
15 scale_in,0,code_gen
16 scale_in,0,decoder
17 scale_in,0,java_compiler
18 job,0,175,175,1,1,1,1,8,1,1,1,1,9,1,1,1,1,156,1,6,1,1,1,1,119
19 job,94,284,190,1,3,1,1,1,9,1,1,1,1,1,5,1,1,1,1,175,1,8,1,1,1,1,172
20 job,189,459,270,1,3,1,1,1,5,1,1,1,1,1,9,1,1,1,1,251,1,9,1,2,1,202
21 .... additional simulation data
```

Listing 3.4: Console output from ABS simulation.

Here, the argument `-p` specifies the dedicated port to listen on, whereas `-l` sets an optional timeout of the simulation in milliseconds. The web-server itself will keep running after the simulation has finished, either until manually terminated by the user or if the server crashes.

With the web-server up and running on the specified port, we can now start investigating our exposed objects. In the HyVar-example, the dataset contains more than 1440 entries. Depending on the provided parameters, the simulation will take approximately 20 minutes to finish on our VM. Querying the state of an object before the simulation is finished is possible, but will return an incomplete dataset. We therefore, advise the users modelling in ABS to add some flag or boolean value to determine whether a simulation is finished or not. Doing so we will be able to prevent the storing of simulations before they are completed.
We can access our instantiated web-server in the browser by typing the address:

http://our_targeted_IP:port/o

Here, IP is the origin IP-address where our web server runs, or "localhost" when running the server locally. Querying this address will return each exposed component in the model as JSON objects. These components basically act as API-endpoints for the data we want to access. An example of the data returned by querying one of the endpoints is shown in Listing 3.5.

```
{
  "name": "hyvarrec",
  "cost": 320,
  "parallel_cost": 0,
  "instance_init_time": 480,
  "instance_speed": 500,
  "scale_in_threshold": 55,
  "scale_in_amount": 1,
  "scale_out_threshold": 8,
  "scale_out_amount": 1,
  "initial_instances": 1,
  "scaling_down_ratio": 142,
  "max_conn": 30,
  "cooling_off": 600,
  "instances": {},
  "history": [],
  "instances_in_time": [...data],
  "latency_in_time": [...data],
  "requests_in_time": [...data],
  "pending_in_time": [...data],
  "real_instances_in_time": [],
  "real_latency_in_time": [],
  "not_killed": false,
  "pending_jobs": 0,
  "running_jobs": 0,
  "job_counter": 52252,
  "pending_job_list": [],
  "measure_list": [...data],
  "measure_count_list": [...data],
  "round_robin_instances": []
}
```

Listing 3.5: Detailed state of the hyvarrec component after finished simulation.

The arrays which are given by each endpoint will act as our source of data for the visualizations. For the HyVar example these endpoints are displayed in Listing 3.6. As mentioned earlier, this is one of the only constraints the user/developer must be aware of for our application to work towards the model data. As long as the correct nesting of the objects is applied, and the data is accessible as arrays, the application should be able to convert it into representations. The algorithm implemented to fetch our data is presented in Section 3.5.2.
3.5 Storing the data

In this section, we explain the implementation of fetching and saving simulations. Having our simulations stored in a database makes the application more flexible in terms of comparing simulations. Throughout a development process, the user may change the structure or requirements of the model. Storing simulations will provide more insight into how the model has developed over time and how current simulations compare to previous ones.

We have to be aware that there are some limitations to the application. For the application to understand that two simulations comes from the same model, the component-name must remain unchanged. The same rule applies to the variable names of our datasets. When listing the components in the chart comparison view, which we will demonstrate later on, only components with the exact name will be grouped. The user will obviously always have the possibility to view both simulations, regardless of the component- or variable name change. However, they will if changed, be separated by the application logic, meaning that they will also be separately displayed.

3.5.1 Database

As mentioned in Section 2.5.1, MongoDB comes with a shell that accepts database queries. In Meteor this shell is accessed by running `meteor mongo` in any folder of a Meteor project. Queries to the database can also be done in the JavaScript code which will return its result to the application.

Some of the queries relevant to this project are:

- `show dbs` - Lists up the available databases on the web-server.
- `db` - Returns current database name.
• *show collections* - Lists all available collections.

• `db.<collection>.find()` - Selects all documents in a collection. Optional query parameters can be added.

• `db.<collection>.findOne()` - Returns one document from a collection. Optional query parameters can be added.

• `db.<collection>.insert(obj)` - Inserts one or more documents into a collection.

Developing the application requires us to consider that the type of information to visualize may differ from one project to another. Thus it is important to create a flexible application which adapts to these changes. Since we can get our data represented as JSON objects, using a NoSQL database such as MongoDB enables us to insert the data directly, without any transformation. MongoDB does not enforce any document structure and shifts the responsibility in terms of structuring the data to the developer. We explained how the object structure should be applied in the previous section.

### 3.5.2 Collections

As Meteor stores its data in a running MongoDB instance, we need to predefined our collections. A good practice would be to implement a new collection for each type of data. In the application, we are using only one collection for all the data, but we can see the advantages of separating different types of data into multiple collections for bigger projects. For the examples in this project, the UI-components need access to all the data, so we chose to only define a single collection. If one were to use this application on multiple projects at the same time, it would be advised to either run separate instances of the application or have the application components connect to different databases.

Defining and exporting collections is done in a separate file residing in our api-folder. This way we can easily import it where needed.

In our export statement shown in Listing 3.7, we specify an optional `idGeneration` parameter so that Meteor generates each document with a unique *id* represented as a *Mongo.ObjectID* instead of a default arbitrary string. We also define a name for the collection, in our case *FullSimulations*, which exposes the data to both the server and the users of our application. Meteor publishes all collections and documents to our application by default. This can be handled using the *autopublish* feature. If we were to turn off autopublish, we would need to explicitly define which collections our users should have access to. We have decided to turn it off for this project, although we still allow all users to subscribe to the collection. This feature however, could be beneficial in cases where we would want to restrict some data from certain user groups.
import { Mongo } from 'meteor/mongo';

export const FullSimulations = new Mongo.Collection('FullSimulations', {idGeneration: 'MONGO'});

Listing 3.7: Defining and exporting our collection.

Since we have turned off autopublish, we need to tell our server which data it should expose to our clients. As previously mentioned we only have a single collection `FullSimulations`, so we define a helper to return our data. Helper functions are template specific and enables us to access and modify data. In Meteor, the HTML `body` tag is accessible as a special template, and acts as a wrapper around all the other child templates. As shown in Listing 3.8, this is where we subscribe to the data, and in doing so we enable all other templates to query the data.

```javascript
Template.body.created = function () {
  this.subscribe("simulations")
}

Template.body.helpers({
  simulations(){
    return FullSimulations.find({});
  }
});
```

Listing 3.8: Subscribing to our collection in Meteor.

In the helper method shown Listing in 3.8, we are returning all the data to our templates. We can also perform other operations using the helper methods, such as sorting or filtering if necessary. In the implementation, listing data in drop-down menus is done by sorting the datasets alphabetical before rendering them. Since the JavaScript `fetch` API methods are inconsistent in the order they retrieve our endpoints, sorting the datasets before passing it to our templates assures consistency in our application, and makes the process of switching between components or datasets more seamless for the users. If this sorting had not been implemented, the values of the drop-down menus would be more or less randomized with every insertion to the database. The sorting is done by using the `Array.prototype.sort()` method provided by JavaScript.

The algorithm we use to fetch data from the model-API into our database is shown in Algorithm 1.
**Algorithm 1:** Fetching data from Model API and storing it in database.

**Data:** Fetch API-endpoints from ABS model-API

**Result:** Inserts data from model-API to database

\[
\text{begin} \\
\quad \text{foreach } \text{endpoints } n \text{ do} \\
\qquad \text{fetch dataset;} \\
\qquad \text{foreach } \text{data in dataset do} \\
\quad \quad \text{if } \text{length of data} > 0 \text{ and data of type Array then} \\
\quad \quad \quad \text{add data to list } T; \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{end} \\
\text{Insert list } T \text{ to database;} \\
\text{end}
\]

In the function where we implement this algorithm, we are initially parsing the input string, and validating against e.g. trailing front slashes or empty strings. The function is being called when the user clicks a button `fetch running model`. Here, the user must specify the URL to where the ABS model-API is running. Since the user specifies the input-URL via a prompt, we have to check the connection against the server. If the connection fails in any way, it defaults the connection back to the applications’ local IP-address.

Fetching data from the model-API is currently a challenge unless the web-server is located at the same origin as the running API. The web-server does not allow Cross-Origin Resource Sharing (CORS) [55] to make fetch requests. Since the user specifies the input-URL via a prompt, we have to check the connection against the server. If the connection fails in any way, it defaults the connection back to the applications’ local IP-address.

To circumvent this error we are using a browser add-on which adds to our response header: ‘Allow-Control-Allow-Origin: *’, enabling us to retrieve the data. This is a limitation the model-API server imposes on us, and possible solutions to this are discussed further in the future work section.

### 3.6 Creating the front end

#### 3.6.1 Overview

As mentioned in Section 2.5.2, the Plotly.js API was selected as the visualization tool for creating our representations. Plotly provides us with basic charts that support the scope of this thesis, while also offering more advanced functionality which can be added at a later stage if needed. Some of these charts include:

- Scatter plots
• Line charts
• Bar charts
• Pie charts
• Point cloud
• Histogram
• Heatmaps
• 3D Plots and animations

We have not incorporated all of these representation types in our thesis, however, the application should have no restrictions on expanding to other types. The Plotly API have good documentation on how to implement further charts.

3.6.2 Visualization implementation

Plotly provides us with functions to render our charts by utilizing their API. Rendering a new plot using Plotly can be done as simple as shown in Listing 3.9:

```javascript
Plotly.newPlot(divToRender, data, layout, optionalConfigArgs);
```

Listing 3.9: Rendering a new plot using Plotly.

The charts created with Plotly are constructed using JSON objects which define each attribute. These attributes are ranging from the selected dataset, type of chart, the desired layout, to various annotations. Many of the attributes are optional and contain default values if not explicitly defined.

We demonstrate how we implement one of the graph functions in our application in Listing 3.10.
function renderFromSource(element, type, mode, data, dataset) {
    plotData = {};
    Object.keys(data).forEach(function(key) {
        if (data[key].namo == method) {
            method = data[key];
        }
    });

    Object.keys(method).forEach(function(key) {
        if (key == dataset) {
            plotData = method[key];
        }
    });

    var traceB = {
        y: plotData,
        type: type,
        mode: mode,
        name: dataset
    };

    // Here we perform our type specific operations

    var layout = {
        title: plotname + " - " + dataset
    };

    trace = [traceB]

    return Plotly.newPlot(element, trace, layout)
}

Listing 3.10: One of the implemented rendering methods.

The method shown in Listing 3.10 creates a trace containing a given dataset `traceB.y`, where the plot type, drawing mode and name of the dataset is defined as parameters in our method call. The iterators makes sure the data passed on to the `Plotly.newPlot` call is consistent with the name of the dataset selected from the user via our UI as shown in Figure 3.6. Worth noticing is that `Plotly.newPlot` actually re-renders the entire plot inside a `<div>` element, which is a costly operation in terms of performance.

![Figure 3.6: All dropdowns in comparison view.](image)

When we only want to update the plot, we instead use the `Plotly.restyle` method, as it only re-renders the traces, significantly improving the rendering performance. To implement this we use a reactive variable, `instance.instanciated`, which ensures that we only perform the initial render once. If a plot has not been rendered in the UI, the entire `<div>` containing the plot is rendered, otherwise, we just update the plot traces.
We will explain more closely how these parameters are selected from the user, and which possibilities they give. First of all, the plot types we have chosen to implement in our application are:

- **Scatter**: One of the most basic charts, and is commonly used where it is needed to interpret bivariate data [11]. For rendering choices we are using WebGL-rendering which compared to traditional SVG-rendering is faster and can handle more data.

- **Filled**: Operates as a scatter plot where the area between the current trace and the previous trace is filled with semi-transparent color. Changes to attributes can be made to fill different areas of the graph.

- **Bar**: Categorically displays the different y-axis values for a given x-axis value using bars.

- **Histogram**: Shows the distribution between our data. The distribution is grouped into *bins*. By default, the range of each bin is decided using a Plotly algorithm.

- **Filter**: Provides the user with a scatter plot and a range-slider to filter values in the range of its *y-axis values*.

For our scatter and overlay plots, we also have the possibility of selecting different *modes*. The available modes alter the way our scatter plot is displayed:

- **Lines**: Draws lines between each coordinate in the scatter plot.

- **Markers**: Shows only markers at each coordinate.

- **Lines+Markers**: A combination of the two above.

### 3.6.3 Comparison view

The application UI is mainly sectioned into three different parts. The topmost section consists of a comparison layout where components and datasets from all the stored simulations can be selected and viewed. This part of the application is shown in Figure 3.7. The data available is accessed by the user through a drop-down menu. After a component is selected by the user, its corresponding datasets are automatically filled into an adjacent drop-down menu. These datasets consist of the arrays the user specified when creating the model.

Once a component with corresponding data has been selected by the user, the plot is being rendered on the page. The chart selection area is retrieved via the Plotly API, and have features such as zoom, pan, auto scale and save to disk. The user can also toggle *compare data on hover* in the plot as shown in Figure 3.8, which will compare the y-axis values of all the simulations displayed on a given x-axis point on the graph. This is a fast way to compare single values between multiple datasets.
Also in the comparison section, the user has the option of toggling between different simulations within the same dataset. This is one of the key features implemented. Having the option to display multiple simulations, and toggle between them enables efficient comparison. It would be reasonable to assume when comparing simulations, the user would want to see which parameters have changed. Clicking a trace in the graph will therefore, display its corresponding data in a box below the chart. Toggling between multiple graphs will give an overview into which parameters have been modified. This is explained further in Section 3.6.4.

Next to the drop-downs containing components and datasets, there are two additional menus containing the available graph types and display modes. The graph types and modes were listed earlier in this section.
Swapping plot types

Switching between different types of plots is a task performed by the user via the UI. All the available plot types are predefined in our HTML code, and any additions to be made in the future can be done directly in the project `graphs.html` file.

The values are provided via a `<select>` element as shown in Listing 3.11.

```
<select id="plotselect" class="form-control" size="7" disabled>
  <option disabled value> -- select graph type -- </option>
  <option selected value="scattergl">Scatter</option>
  <option value="filled">Filled</option>
  <option value="bar">Bar</option>
  <option value="histogram">Histogram</option>
  <option value="filter">Filter</option>
</select>
```

Listing 3.11: Provided plot types as defined in graphs.html.

The values specified within each option is available to our JavaScript code. Each time a selection is done that constitutes a render of our plot, we call the render function. The render function for the comparison view takes the parameters shown in Table 3.2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>element</td>
<td>Element to be rendered</td>
</tr>
<tr>
<td>type</td>
<td>Plot type</td>
</tr>
<tr>
<td>mode</td>
<td>Plot mode</td>
</tr>
<tr>
<td>datasetName</td>
<td>Name of dataset</td>
</tr>
<tr>
<td>filterRange</td>
<td>Lowest and highest value of our dataset</td>
</tr>
<tr>
<td>filterOperator</td>
<td>Filter operator (either &gt; or &lt;)</td>
</tr>
<tr>
<td>instance</td>
<td>Template instance used to set reactive variables</td>
</tr>
</tbody>
</table>

Table 3.2: Parameters passed to our render function.

The render function initially fetches the simulation data, iterates through the datasets and tries to match it with the name of the user selected option. When the function finds the correct dataset it then checks which plot type is specified. Mode- and graph specific operations are then performed on the trace before we treat the data which are rendered in the object state view we explain later in this section. Next, we add attributes to the plot area which, making it responsive when scaling the browser window. Lastly, we attach our listeners and render the plot.

The `filterRange` and `filterOperator` parameter shown in Table 3.2 is specific for the filter graph type. This plot allows users to filter y-axis values below/above a given threshold. The user can control this parameter directly in the UI by using a range slider together with a radio form. An example of how filtering can be done in the application is seen in Figure 3.9.
Since the user should be able to perform a comparison of the simulations we also implemented a section of our application where if a user clicks a point in the graph, they get the annotated information from the model displayed. The implementation is done by checking all the objects in the component and disregarding arrays (which are the datasets rendered in the plots). We iterate through all the component data, add them to an array as strings and separate them using a key => value structure. The data is in turn updated into a reactive variable which refreshes our UI each time a new point on the graph is clicked.

The implementation of the detailed object state view is performed in the render function we are using. The full simulation data with its parameters are stored in a variable inside our trace object, filtered, and passed onto two new separate string arrays inside the same trace object.

The parameter names are specified as key and its values fittingly is stored as val as shown in Listing 3.12.

```javascript
1 Object.keys(traceA.fullData).forEach(function(key) {
2   if (!Array.isArray(traceA.fullData[key])
3     && typeof traceA.fullData[key] !== "boolean" && typeof
4     traceA.fullData[key] !== "object") {
5     stringData += (key + ": " + traceA.fullData[key] + 
6     "\n")
7     o = {};
8     o.key = key;
9     o.val = traceA.fullData[key];
10    stringArray.push(o);
11  }
12});
13 traceA.stringData = stringData
14 traceA.stringArray = stringArray
15 data.push(traceA)
```

Listing 3.12: Filtering the data and pushing it to our trace object.

The data is in turn sent to a Plotly function which listens on any click events performed on the trace. If a trace is clicked the reactive variable as mentioned is updated as demonstrated in Listing 3.13.
Listing 3.13: Updating reactive variable with coordinates and the object state when clicked on a plot trace.

This is reflected in the view as shown in Figure 3.10. Initially the `<div>` where we display the object state is only rendered with a descriptive message that explains how we get the full data as displayed in Figure 3.11.

![Comparison diagram - showing requests_in_time](image)

---

FIGURE 3.10: Implemented view of component parameters together with plot.

---

FIGURE 3.11: Displayed object state view before any plot trace has been clicked.
3.6.5 Subplot view

In the bottom section of our application, we are proposing a more detailed representation of each simulation component. Here the users can select a specific simulation they want to investigate. When selecting a simulation, all its available components become available as subplots. This is demonstrated in Figure 3.12. Choosing a component will render an overview of all its datasets. This is particularly handy if we want to see how each component is behaving throughout a simulation. The simulation is being selected by the user through a drop-down menu as shown in 3.13.

![Subplot diagram - showing component_encoder](image)

Figure 3.12: Subplot view section.

Select a dataset from the dropdowns below to display subplots

![Select a simulation](image)

Figure 3.13: Subplot dropdown component.

To dynamically fit all the subplots into a display we have created an algorithm which ensures that every subplot gets an equal amount of space. The positioning of multiple plots in Plotly is done by defining domain attributes which range between [0 - 1]. A position has to be defined for all the subplots. The value 0 represents the top section, and 1 is the bottom section. For instance, will a plot with domain attributes x: [0, 0.5], y: [0.5, 1] be rendered in the lower left quadrant of the available display space. Since
the number of subplots to render is specific to each dataset we need to solve this dynamically. Algorithm 2 creates a set of values that take up the entire horizontal space and splits the vertical space approximately equal. We also add a small aesthetic buffer which separates the plots slightly.

**Algorithm 2:** Create a dynamic subplot based on number of datasets.

**Data:** Datasets to be plotted in subplot

**Result:** Subplot positions are equally distributed within domain

```
function(n)
    A ← []
    r ← (1/num) × 100)/100
    b ← ((1/(num + 1)) × 100)/100
    a ← (r – b)
    c1 ← 0
    c2 ← 0 + b
    for all n do
        B ← []
        B.insert(c1, c2)
        A.insert(B)
        c1, c2 ← c1 + r, c2 + r
    end for
```

After creating an array with all the coordinates of our subplots, we anchor the x-axes in our plots to its corresponding y-axes to ensure a correct display. This is also done dynamically as shown in Listing 3.14.

```
for(j = 0; j < arr.length; j++){
    if (j == 0) {
        y = {domain: arr[j].d}
        obj['yaxis' + (j+1).toString()] = y
    }
    %
    else {
        x = {anchor: 'y'+(j+1).toString()}
        y = {domain: arr[j].d}
        obj['xaxis' + (j+1).toString()] = x
        obj['yaxis' + (j+1).toString()] = y
    }
}
```

Listing 3.14: Adding our coordinates and anchoring the x-axes to their corresponding y-axes.

### 3.7 Implementation of the plots

We will explain how we are implementing the different views and plots as explained in the last section.

To separate the concerns of our application, we have divided each of the UI sections into different templates. Each template holds a set of data that
is relevant to their specific use case. The comparison diagram is defined using a template named sims. To render this template into our application, we define it using HTML syntax as shown in 3.15.

1 <template name="ourTemplate">
2  <!-- Our template data -->
3 </template>

Listing 3.15: Defining template views.

The data within the template tags will be compiled into Meteor templates which can then be used in our application. To include a template into the view we have to explicitly render it where needed. This can be done in our HTML code as shown in Listing 3.16, by using Spacebars templates [56].

1 {{> ourTemplate }}

Listing 3.16: Rendering our template.

When the Meteor templates are compiled, they are also accessible to the JavaScript code. Here we can handle application logic and pass data to elements inside the template. We will explain how we are utilizing templates in the application.

First, the templates we have defined are:

- **body** - The global template which acts as a wrapper around all other templates. Inside we include all of our defined templates and put the templates within proper div elements.

- **<template name="sims">** - This is our comparison diagram template. Here we handle the logic and elements relevant to the uppermost section of the page as shown in Figure 3.7.

- **<template name="subplotTemplate">** - The template which handles logic and rendering of the bottom part of our application, which shows all the datasets from a selected component.

- **<template name="delete">** - Handles the logic and rendering where the user can display or delete a stored simulation from our database.

In our JavaScript code we can access and modify the template data using a template.ourTemplate object which enables us to pass data to any of the template elements.

Meteor templates have specific callback events which are based on their life cycle. If we want to modify some data before the DOM elements are rendered, we specify this in the Template.ourTemplate.created function. In Listing 3.17, we are declaring all the reactive variables in this callback function making sure the data is accessible before the web page is actually rendered.
Template.sims.created = function () {
    this.methods = new ReactiveVar([])
    this.dataset = new ReactiveVar([])
    this.plottype = new ReactiveVar("scattergl")
    this.typo = new ReactiveVar("lines")
    this.selectedmethod = new ReactiveVar(["default"])
    // more declarations....
}

Listing 3.17: Declaring our reactive variables before rendering the webpage.

By declaring the reactive variables and specifying default values before rendering any elements, we do not force any changes to the web page before the user has actually selected an initial data set. At that point, all the reactive components have already been declared, and additional re-rendering of elements only happen when the user specifies a change, or if an additional simulation gets added to/deleted from our database.

Now that we have a Meteor template function that handles the work which happens before the DOM elements render, we also need to deal with what takes place after the initial render. Here, we take advantage of the Meteor.ourTemplate.rendered callback function. Templates in Meteor comes with a tracker function which refreshes the specific templates when data sources changes. In the application, this is done either when the values of our reactive variables or database change. All the events we want to handle is defined within the autorun function of each template.

3.7.1 Handling events

As previously mentioned, our application contains various events. These can be either inserting or deleting simulations to/from the database or when the user toggles datasets or chart types. Since we are expecting interaction from the user we must process these events.

Meteor provides us with an event function which we are using on the database queries. This is demonstrated in Listing 3.18.

Template.delete.events({
    'click #btndelete'(event){
        element = document.getElementById('deletesimulations')
        val = element.options[element.selectedIndex].value
        var del = val.split("\"")[1]
        FullSimulations.remove({_id: new Mongo.ObjectID(del)})
        Tracker.flush() // Force reactive updates pending
    }
})

Listing 3.18: Using template events to delete database entries.

For the rest of the application events, such as when a user toggles any of the datasets or chart types, we are attaching listeners to elements inside our
templates. This is demonstrated in Listing 3.19.

```javascript
Template.sims.rendered = function() {
  this.autorun(() => {
    var typo = document.getElementById('typeselect')

    // Listener to change plot type.
    typo.onchange = function() {
      instance.set(typo.options[tyho.selectedIndex].value)
      if(instance.methodselected.get() && instance.d_datasetselected.get()){
        if(instance.plottype.get() != "filter"){
          renderplot("simList", instance.plottype.get(),
                      instance.tyho.get(), instance.selectedmethod.
                      get(), instance.selecteddataset.get(),
                      instance.filterRange.get(), instance.
                      filterOperator.get(), instance)
        }
      }
    }
  }
}
```

Listing 3.19: Listening for events on our template elements.

The code shows how we are first fetching the Template element as an object. We can attach listeners to this object which check for any changes the user might be specifying. In the example above, we are listening on a `<select>` element, and any changes to its options will update our instance variables. We have applied listeners to each of our Template-elements, except the ones that are rendered directly by Plotly. Worth noting is that the DOM elements inside our Templates are only accessible after they are rendered. Hence we have to add our listeners and data transformation methods inside the `Template.ourTemplate.rendered` function. This prevents errors where any listeners try to attach non-existing DOM elements.

Since there is no default limitation on how many listeners that can be added to our elements, we have to make sure we are only attaching each listener once. If listeners are added each time we update a plot the application will encounter memory leaks as the listeners will stack.
Part II

Conclusion
Chapter 4

Results

The research question we set out to answer was:

- How can we facilitate users of the ABS language with visualization support?

In order to answer this question we have looked at various established visualization techniques and principles. We have implemented an application which utilizes the ABS model-API to fetch and store simulation data. Representations of this data are presented in three different sections of a web application which dynamically reacts to modification of data and events performed by the user. In this chapter, we will take a look at how the visualization principles are carried out in the application.

4.1 Measuring our visualizations

4.1.1 Positioning

In Chapter 2 we introduced different techniques and principles to displaying our data. Here we take a look at how our implemented plots enable our users to compare the data presented.

Comparison view

The comparison view of the application displays multiple graphs within the same space. This enables users to compare different points of the graph without shifting their attention out of the visual space. Rightmost inside the same space, as shown in Figures 4.1 and 4.2, the user can also toggle between the different simulations, enabling the exploration of connections between the datasets. This is one of the main requirements that had to be successfully implemented in the application. The categorical display of the comparison view falls within the superposition principle.
We can also see that the different charts are separated by distinct colors, which should enable users to spot and identify sections of interest in the plots.

Figure 4.1: Comparison of three datasets with superposition design in the HyVar example.

Figure 4.2: Comparison of two datasets with superposition design in the HyVar example.

**Subplot view**

In the subplot view as shown in Figures 4.3 and 4.4, we present users with the ability to track the progress of an entire component throughout a simulation. The plots are displayed within the juxtaposition category. They are separated in different grids, and the user must shift its attention between multiple sets of data to draw connections or create hypotheses.
4.1.2 Distortion

We will take a look at what level of distortion is introduced in our application. As mentioned in Chapter 2, data integrity can be compromised if displayed incorrectly.

**Lie Factor**

To check the application for any misleading graphs initially rendered we test against the Lie Factor introduced in Section 2.2.5. We remember how the Lie Factor is calculated.

\[
\text{Lie Factor} = \frac{\text{Size of graphical difference}}{\text{Size of actual data difference}}
\]
First, the Lie Factor is measured towards the comparison view. In Figure 4.5, the numerical difference between our values is calculated as

\[
\frac{8 - 4}{4} \times 100 = 100\%
\]

This is an easily calculated example as the increase is doubled. We then measure against the graphical difference in the plot. The grid lines of the plot are used as measurements. The first value (4) spans over exactly two grid lines, and the second value (8) is measured at four grid lines. This increase is calculated as

\[
\frac{4 - 2}{2} \times 100 = 100\%
\]

Remembering our equation above, we calculate the Lie Factor:

\[
\frac{100}{100} = 1,00\%
\]

The calculation of the Lie Factor in this example proves that the initial render of the graph is proportional. The integrity of the graph is therefore validated.

We apply the same calculations as above in our subplot view. In Figure 4.6, we use only a section of the graph for reference. The plot has not been zoomed or modified, we just use a selection of the view for testing the values.

This graph represents the pending in time dataset from the c compiler component. First, we calculate the numerical difference between the two values. The first value we use to measure is (x = 334, y = 157). We compare this to the second value on the graph (x = 393, y = 213). This equates to

\[
\frac{213 - 157}{157} \times 100 = 35,66\%
\]

From the grid lines we calculate roughly the graphical difference. The first value (157) spans over 3,2 grid lines. The second value (213) spans over 4,3 grid lines. This equates to an increase of

\[
\frac{4,3 - 3,2}{3,2} \times 100 = 34,37\%
\]

Calculating the lie factor gives us the following equation:

\[
\frac{34,37}{35,66} = 0,9638
\]
This proves that the implemented subplot is within the recommended values of the Lie Factor, meaning that the relative differences between the values in the graph are proportional.

The way we make sure we are within the recommended Lie Factor range is to always display the full range of the y-axis by default. If a user of the application wants to modify the context or manipulate data, they can either change the source code directly or use the zoom functionality in the active plot.

![Comparison diagram - showing instances_in_time](image1)

Figure 4.5: Measuring the Lie Factor in comparison view. Plot data is *instances in time* from the c compiler component.

![Subplot view](image2)

Figure 4.6: Measuring the Lie Factor in subplot view. Targeted plot data (red graph) is *pending in time* from the c compiler component.

### Axis stretch

The initially rendered plots are designed to utilize the entire width of the display size. Changes to the aspect ratio of the display will modify how the plot can be interpreted. There are at least two ways to tackle this issue. The first would be to not scale the plot with the aspect ratio of the window. For instance, when downscaling the browser window, some of the plot data would be hidden and required horizontal scrolling to be displayed. This is not in alignment with creating responsive charts, and for this reason, we have implemented another solution to this problem.

The application behaves in a way where it scales with the size of where it is displayed. Consider the graph in Figure 4.7. The peaks in this plots seem
more dramatic and frequent than the one in Figure 4.8 which is displayed in a full-size browser window. This indicates that some level of distortion is introduced. However, we are always displaying the full range of values in the dataset. By doing so the context should be available to the user.

Figure 4.7: Reduced plot aspect ratio, data is pending in time from the compiler component.

Figure 4.8: Full plot aspect ratio with same dataset as in Figure 4.7.
Chapter 5

Summary

5.1 Evaluation

The primary goal of this thesis was to create a visualization application which supports simulation data from ABS. By testing the application against traditional visualization principles, and following the constraints provided by the ABS language, we have developed a proof of concept application that is tested towards an existing ABS project. By integrating our implementation with Plotly, a well known and established open-source visualization API, we are able to provide users of ABS with an easily deployable application that can support their simulation data. The script for deploying the application together with an instance of the HyVar project and the ABS tool chain is available in Appendix A. A link to the repository with the project source code was provided in Section 3.1.

To construct the proof of concept application we had to fulfil a set of requirements that we defined in advance. We had to construct an application that could be able to: 1) store simulation data to enable comparison of data, 2) communicate with the model-API, 3) be dynamic and react to changes in our data, 4) adhere to established visualization techniques.

Section 3.5 shows us how we are able to store simulation data to enable comparison. In Section 3.5.2 we implement the fetching of data which is targeted towards the model-API. Section 3.6 explains how the application is staying dynamic, while Chapter 4 tests the application against introduced visualization techniques such as distortion and positioning.

To counteract the misinterpretation of data, we have implemented multiple ways of viewing the same dataset. In doing so, we provide users with the ability to view the problem at hand from multiple points of view. Making sure that all values of a dataset are available to the user will also help with reducing the ambiguity of the data, as any representation can be viewed in combination with the actual values of data.
There is obviously not a universal solution to all programming problems, and tailoring visualizations to fit each niche ABS project were not within the scope of this thesis. However, we have demonstrated how we can create a general purpose application which integrates with established ABS projects. We will discuss relevant additions to our solution in Section 5.1.1.

The application succeeds in creating visual representation which adheres to established principles.

### 5.1.1 Future work

**Extending visualization support**

Plotly provides a lot of interesting features that can be viable to programmers and developers using ABS. Extending the application to fit more sophisticated graph types and charts could be a way to improve the application. However adding too many types of visualizations to the user may clutter the experience, and take attention away from the problem at hand. This would be interesting to investigate further. Testing the application on large scale projects from beginning to end would be interesting when developing the application further.

**Interface**

We have not performed any changes to the ABS language when working on this project. However there might be changes that could facilitate further support for visualization of data.

By enforcing stricter rules to the structure of the application data, and enable some sort of universal truth to how the users can incorporate their model, there is a possibility of creating an ABS-interface that is in alignment with the application. Since every ABS model is compiled in order to run, and also support interfaces similar to the ones in Java, creating one for this kind of visualization might be a good solution. Implementing methods in the interface which returns specific types of data may ensure a more consistent way of handling the data which our application receives. As long as the application can handle the data types returned from the interface, we should in theory always know the structure and format of the data in advance. This should probably be looked more into.

**Web server**

At present time, to successfully retrieve data that is not local to the origin of our web-server, we need to use a browser add-on which enables us to request AJAX from any source. Open API’s tend to allow access from all sources by adding an ‘Access-Control-Allow-Origin’ header with the value ‘*’. There are potential risks in doing this as you are exposing your API
to the public. However, white-listing requests from the origin of where our web-server is located should be looked further into, as this will make our application more accessible. Adding some security to the application such as OAuth [57] or HTTP authentication [58] could be considered as a useful implementation, especially if the data is of a sensitive nature.

Statistics

Testing the application towards a wider range of projects may produce some trends into recurring data important to the user. Creating statistical operations for these types of data could be interesting. One could imagine that viewing statistics such as highest latency, most spawned instances or lowest combined resource usage across all simulations would be interesting for modellers to get calculated.

Usability and design

The application follows basic visualization principles and is designed in a manner which is trending in modern web-applications. Dashboard resembling applications provides the user with a comprehensive overview over the data at hand, however, this emphasizes the need for good design. Testing the application with ABS users can be beneficial for further development.
Bibliography


Half of Norwegian enterprises are buying cloud services. URL: https://www.ssb.no/en/teknologi-og-innovasjon/artikler-og-publikasjoner/half-of-norwegian-enterprises-are-buying-cloud-services.


URL: https://cloud.google.com/.


This is not just any cloud. This is the IBM Cloud. May 2015. URL: https://www.ibm.com/cloud/.


Amazon EC2 Spot Instances Pricing. URL: https://aws.amazon.com/ec2/spot/pricing/.


Sophie Rust, Jasper Schelling, and Donna Schipper. Building real-time web applications with meteor.


Meteor. URL: https://www.meteor.com/showcase.


plotly. URL: https://plot.ly/javascript/.

Ji Soo Yi et al. “Understanding and characterizing insights: how do people gain insights using information visualization?” In: Proceedings


[53] New Year’s Eve. URL: http://abs-models.org/new-years-eve/.

[54] 6 The Concurrency Layers. URL: http://abs-models.org/chapter/6-the-concurrency-layers/.


Appendices
# Appendix A

## install.sh

1. ```#!/bin/bash```
2. ```# Install erlang```
3. ```wget https://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb && sudo dpkg -i erlang-solutions_1.0_all.deb```
4. ```sudo apt-get update```
5. ```sudo apt-get install esl-erlang -y```
6. ```sudo apt-get install elixir -y```
7. ```# Install java```
8. ```sudo apt install default-jre -y```
9. ```sudo apt install default-jdk -y```
10. ```# Ant```
11. ```sudo apt install ant -y```
12. ```# Create folder structure```
13. ```mkdir master```
14. ```mkdir master/abs```
15. ```cd ~/visualization-abs/master/abs```
16. ```# Clone abs```
17. ```git clone https://github.com/abstools/abstools.git```
18. ```cd abstools/frontend```
19. ```# Build```
20. ```ant dist```
21. ```# Install hyvar```
22. ```cd ~/visualization-abs/master/abs```
23. ```git clone https://github.com/HyVar/abs_optimizer.git```
24. ```# Install Meteor```
25. ```curl https://install.meteor.com/ | sh```
26. ```# Install node```
27. ```curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -```
28. ```sudo apt-get install -y nodejs```
29. ```# Install Meteor Up for easy deployment```
30. ```sudo npm install --global mup```
```bash
41 cd ~/visualization-abs/meteor/master
42 sudo npm install
43 sudo npm update
```

Listing A.1: Script to install the application.
Appendix B

readme.txt

# Steps to run the environment - Available at https://github.com/baugztar/visualization-abs

# 1. Inside project folder set permissions to run install script: `chmod +x install.sh`
# 2. Run install script: `sudo ./install.sh`
# 3. After finished installation set permissions on .meteor folder: `sudo chown -R $USER ~/.meteor`
# 4. To run meteor: `cd meteor/master/ && meteor`

# Make sure the server is able to handle connections on port 3000.

# To make fetch requests against a running model-API, use a browser extension which enables CORS and adds to header response 'Allow-Control-Allow-Origin: *'.

Listing B.1: Description on how to run the application.
Appendix C

Implemented views

Figure C.1: Implemented scatter plot.

Figure C.2: Implemented overlay plot.
Figure C.3: Implemented bar plot.

Figure C.4: Implemented histogram plot.

Figure C.5: Implemented filtering plot.

Figure C.6: Implemented scatter plot with dotted lines mode active.
Figure C.7: Implemented subplot.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>c_compiler</td>
</tr>
<tr>
<td>rest</td>
<td>7200</td>
</tr>
<tr>
<td>parallel.set</td>
<td>0</td>
</tr>
<tr>
<td>instance.val.time</td>
<td>80</td>
</tr>
<tr>
<td>instance.vari</td>
<td>500</td>
</tr>
<tr>
<td>scale.in.threshold</td>
<td>0.5</td>
</tr>
<tr>
<td>scale.in.unas</td>
<td>1</td>
</tr>
<tr>
<td>scale.out.threshold</td>
<td>1</td>
</tr>
<tr>
<td>scale.out.unas</td>
<td>1</td>
</tr>
<tr>
<td>init_refinance</td>
<td>1</td>
</tr>
<tr>
<td>scaling ratios</td>
<td>7</td>
</tr>
<tr>
<td>rest.open</td>
<td>10</td>
</tr>
<tr>
<td>cooling.off</td>
<td>0</td>
</tr>
<tr>
<td>instances</td>
<td>(0.000000000000000000)</td>
</tr>
<tr>
<td>pending_jobs</td>
<td>0</td>
</tr>
<tr>
<td>running_jobs</td>
<td>0</td>
</tr>
<tr>
<td>pk_counter</td>
<td>1383</td>
</tr>
</tbody>
</table>

Figure C.8: Implemented detailed object view.

ABS visualizations

(a) Fetch data button.  (b) Url prompt  (c) Confirm prompt.

Figure C.9: Fetching data from model-API and save to database.

Delete simulation

(a) Delete simulation button  (b) Confirm delete prompt

Figure C.10: Deleting simulations from API.