Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Sebastian Lindtvedt
Salvador Bascunan
Dennis Kristiansen

Cloud-native solution for building digital twins

Bachelor's thesis in Bachelor in Programming
Supervisor: Tom Røise
May 2022

NTNU
Norwegian University of
Science and Technology

Sebastian Lindtvedt
Salvador Bascunan
Dennis Kristiansen

Cloud-native solution for building digital twins

Bachelor's thesis in Bachelor in Programming
Supervisor: Tom Røise
May 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Summary of Bachelor Project

Title	Cloud-native solution for building digital twins
Project No.	32
Date	20.05.2021
Authors	Sebastian Lindtvedt Salvador Bascunan Dennis Kristiansen
Supervisor	Tom Røise
Client	Glex AS
Contact Person	Jørgen Engen Napstad
Keywords	CCS, Digital Twin, Full Stack, REST API, Cloud Native, Azure
Pages	100
Attachments	9
Availability	Open

Abstract

Glex develops digital twins for their customers. In 2022, they announced a task in which they want to explore the possibility of digital twins within ocean space. The task developed into a general solution for generating digital twins based on data. The general solution is used to create a digital twin of Smeaheia, an area for CO_2 storage. This thesis describes the process around a full-stack development project, with the cloud architecture in focus. The end product is a solution consisting of several micro-services for data processing, a REST API and a website.

Sammendrag av Bacheloroppgave

Tittel	Cloud-native solution for building digital twins
Prosjekt Nr.	32
Dato	20.05.2021
Deltakere	Sebastian Lindtvedt Salvador Bascunan Dennis Kristiansen
Veileder	Tom Røise
Oppdragsgiver	Glex AS
Kontaktperson	Jørgen Engen Napstad
Nøkkelord	CCS, Digital Twin, Full Stack, REST API, Cloud Native, Azure
Antall sider	100
Antall vedlegg	9
Tilgjengelighet	Open

Sammendrag

Glex utvikler digitale tvillinger for sine kunder. I 2022 utlyste de en oppgave der de ønsker å utforske mulighetene for digitale tvillinger innen havrom. Oppgaven utviklet seg til å bli en generell løsning for å genere digitale tvillinger basert på data. Den generelle løsningen anvendes for å skape en digital tvilling av Smeaheia, et område for CO_2 lagring. Denne oppgaven beskriver prosessen rundt et fullstack utviklingsprosjekt, med skyarkitekturen i fokus. Sluttproduktet er en løsning bestående av flere mikrotjenester for prosessering av data, et REST API og en nettside.

Preface

We would like to thank everyone involved in this bachelor's project. Thanks to our supervisor, Tom Røise, for continuous support and feedback during the development of this project. Our client, Glex, provided us with a unique task within a field we had no previous experience in. We are thankful for this opportunity, and we would like to thank our Product Owner, Jørgen Engen Napstad together with Brit Thyberg and Patrick Sullivan, for their active engagement and assisting us in developing the best possible solution.

Contents

List of Figures

List of Tables

List of Listings

Glossary

Atlassian Company offering tools for software engineering and software development. 92

Azure Microsoft's cloud service. 93

Azure Functions Azure's function as a service offering. 94

Azure Storage Explorer A program containing a GUI for accessing and manipulating data in Azure Blob Storage. 6

CCS Carbon Capture and Storage [59]. 2, 100

chronostratigraphy The study of the ages of strata. The comparison, or correlation, of separated strata can include study of their relative or absolute ages[33]. 12, 27, 71

Confluence Atlassian's document authoring solution. 92

DOM HTML Document Object Model. The parsed form of an HTML document. 44

Figma Prototyping and wireframing software. 25, 26

Flask Minimal Python framework for web applications. 32, 33, 47, 48, 93

glTF A open specification format for models and scenes. x, 32, 34, 37, 50, 55, 64, 66, 72, 94

Jest A testing framework for JavaScript. 43

Jira Atlassian's issue tracking solution. 92

lithological Pertaining to lithology, The study of rocks [61]. 71

lithostratigraphy The study and correlation of strata to elucidate Earth history on the basis of their lithology, or the nature of the well log response, mineral content, grain size, texture, and color of rocks[34]. 4, 12, 27, 71

microservice A microservices architecture is an approach to building a server application as a set of small services [27]. 1, 7, 10, 11, 17, 18, 29, 30, 32, 37, 47, 94, 96, 97

PyTest A testing framework for Python. 41, 42

React Popular frontend web development framework for JavaScript. 32, 34, 43, 44, 46, 47, 95, 97

React Testing Library A support library for writing DOM tests for React components. 43

React Three Fiber A JavaScript library combining ThreeJS and React. 32, 46, 95, 96

SDK Software Development Kit. 35–37, 39, 47, 93

strata Layers of sedimentary rock[35]. 71

ThreeJS A JavaScript library enabling 3D rendering using WebGL. 31, 32, 34, 44, 46, 47, 50, 66, 79, 95, 100

TVD True Vertical Depth, [63]. 69, 70

TVDKB True Vertical Depth Kelly Bush, [64]. 69

TVDSS True Vertical Depth Sub Sea, [65]. 69

Unity A high quality game engine. 95, 96

webgl WebGL is a cross-platform, royalty-free open web standard for a low-level 3D graphics API based on OpenGL ES [55]. 96

wellbore The drilled hole or borehole, including the openhole or uncased portion of the well [37]. 4, 11, 12, 37, 67–71, 73, 94, 95

1 Introduction

As of May 2022, there is a live version of the web-app. We recommend running it on a 1920x1080 resolution.

This bachelor's task was provided by Glex, our contact person was Jørgen Engen Napstad.

Glex is a company based in Bergen. Their focus areas are a unique mix of game development, geology and geospatial technology. Glex focuses on building solutions that solve complex problems for several customers.

Glex's main building blocks consist of Oil & Gas Exploration, Deep Sea Minerals, Renewable Energy, and Carbon Capture & Storage. With this unique toolkit, Glex is able to provide customers an end-to-end solution for managing and staying up to date with their exploration and/or renewable portfolios.

Currently, Glex has a couple of customers using their main software, Glex Energy[14]. This is within the Oil & Gas Exploration industry. For the time being, this is Glex's main area of focus, but they are experimenting with new segments, such as Carbon Capture and Storage.

For this project, Glex presented the idea of creating a digital twin, a virtual representation of a physical object. The group discussed several different cases with Glex, in regards to what the project focus was going to be. It was established early on that the group were to utilize a dataset that had to be processed and converted to a format which were sensible for further visualization. One of the datasets which Glex suggested for this purpose was the Smeaheia dataset. This dataset would serve as our specific project case. During these discussions, the group quickly realized that the functionality required to parse and process the Smeaheia dataset could be transformed into a more generalized format, facilitating the parsing of other datasets in the future. This would result in a solution that was not only able to process the data in the Smeaheia dataset, but any other datasets with similar types of data. The general idea behind the solution was to have a microservices-like architecture that could handle a broad range of different datatypes, parsing and processing them on-demand. The data would be both static and dynamic. Everything would then be served through a REST API for the web-app to consume and display with a graphical user interface.

1.1 Domain

To establish a common knowledge ground for this thesis, a few clarifications regarding terminology and concepts are beneficial.

Digital Twin

According to IBM[17]

A digital twin is a virtual model designed to accurately reflect a physical object. The object being studied for example, a wind turbine is outfitted with various sensors related to vital areas *of functionality. These sensors produce data about different aspects of the physical objects performance, such as energy output, temperature, weather conditions and more. This data is then relayed to a processing system and applied to the digital copy.*

Carbon Capture & Storage

There is a growing need to be able to safely dispose of excess CO_2 in the world today. This is especially important in the Oil and Gas industry. Carbon capture and storage, CCS, is a solution to this. CCS is a process where excess CO_2 is captured at a production site and transported to a storage site where the CO_2 is stored permanently. Interest in this area has increased following The Norwegian Government's announcement of project Longship, Equinor's project Northern Lights, and Blue Barents CSS project. All projects dedicated to realizing the CSS potential present on The Norwegian Continental Shelf.

According to NPD[31]:

Depending on their specific geological properties, several types of geological formations can be used to store CO_2. In the North Sea Basin, the greatest potential capacity for CO_2 storage will be in deep saline-water saturated formations or in depleted oil and gas fields.

CO_2 will be injected and stored as a supercritical fluid. It then migrates through the interconnected pore spaces in the rock, just like other fluids (water, oil, gas).

And according to Gassnova[13]

We need to increase the efficiency of our energy consumption and promote sources of non-fossil fuel energy. Yet despite advances in these areas and technological leaps forward in renewables such as solar, wind and hydropower, the world is in danger of failing to meet those targets. In turn, we are in danger of failing our responsibility to future generations. Energy intensive industry accounts for 25% of global CO2 emissions and cannot go down to zero without CCS.

An area with potential for CO_2 storage on the Norwegian Continental Shelf, is the Smeaheia area, located on the Hordaland Platform in the North Sea near Mongstad. This area has been evaluated by both Gassnova and Equinor, and together they have released and open-sourced a dataset containing subsurface data, reports, and geomodels, for the purpose of encouraging research and learning around CCS. [42]

Surface Layers

The definition of a surface is - geology and correlation of rock formations, structures, and other features as seen at the Earth's surface [58]. In this case, we worked with sub-sea level surfaces which contains several layers underneath itself. We have a seabed surface at the top, and multiple other layers of "rock" which creates the context for other visualizations such as faults and well data. The surfaces in this dataset are represented as a point cloud. An example of visualization of surfaces:

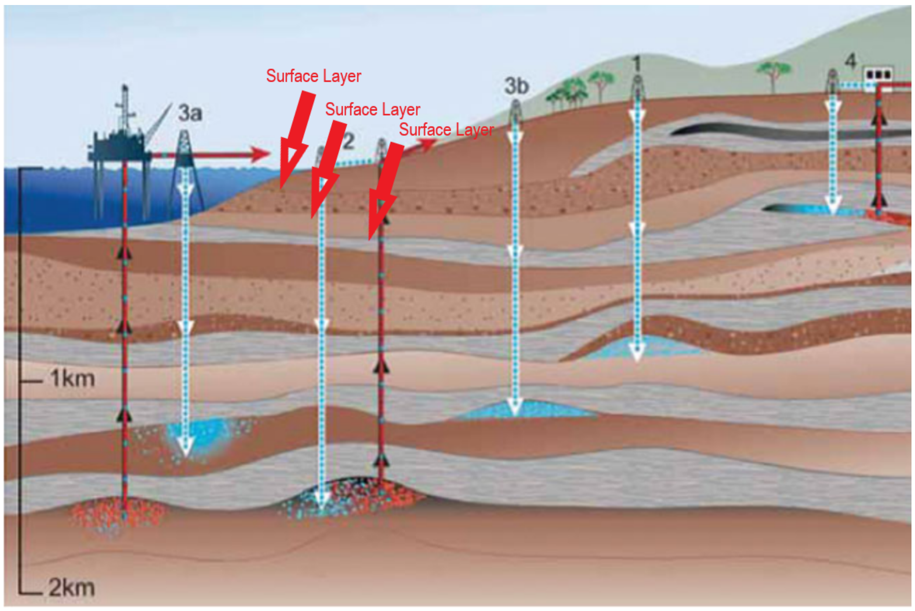


Figure 1: Examples of surface layers

Faults

A fault is one or more fractures between two blocks of rock [53]. The visualization of faults is utilized to increase the visibility of these fractures. A fault stick is a part of a fault, and is a single "line" within the fault. Geologists use the angle of the fault with respect to the surface and the direction of slip along the fault to classify faults[19]. Examples of visualization of faults:

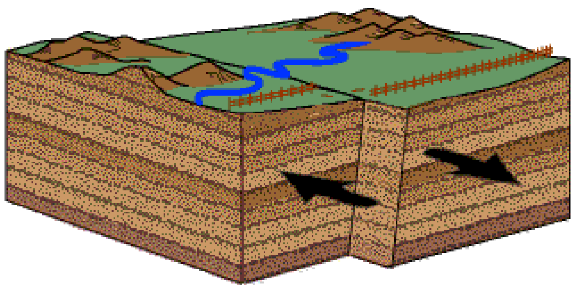


Figure 2: Example of how a fault occurs

The fault sticks are the black lines, the fault itself is the red "plane"

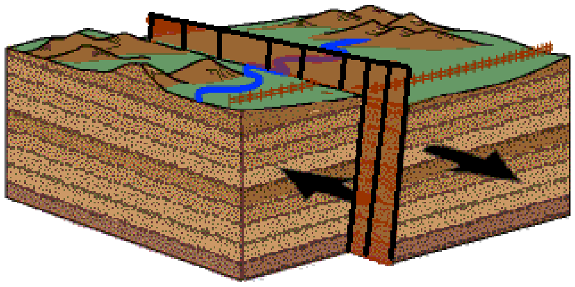


Figure 3: Example of how faults are visualized in our solution.

Wells

The well data consists of information about several parts that will be visualized in the scene. These are:

- Wellbores

  The drilled hole or borehole, including the openhole or uncased portion of the well. Borehole may refer to the inside diameter of the wellbore wall, the rock face that bounds the drilled hole[37]. In our specific case, the wellbore is utilized for the injection of CO_2.

- Well-logs

  The measurement versus depth or time, or both, of one or more physical quantities in or around a well[36]. These can be represented in the scene as geometry alongside the well trajectory, or as a 2D graph.

- Lithostratigraphy

  Lithostratigraphy is the study and correlation of strata, which are the layers of sedimentary rock [35], of Earth history based on the well log response, mineral content, grain size, texture, and color of rocks [34]. For instance, a lithostratigraphy log, can be a vertical slice of the earth, showing different types of rockmass, like sandstone and slate. These rockmasses have different properties that are of interest to a geologist.

- Chronostratigraphy

  Chronostratigraphy is the study of the ages of strata. This comparison, or correlation, of separated strata can include study of their relative or absolute ages [33].

1.2 Target Audience

The finished product will be of interest to several different actors. There will a resulting thesis, in addition to a product.

1.2.1 Thesis

The thesis will primarily interest of the actors involved in the grading process of this project, but we do believe that all individuals with an interesting in computer science and technology will find it interesting. It is worth noting that some technical experience within software development and programming is expected, in regards to understanding all aspects of the thesis.

1.2.2 Product

The solution we are developing will have different use cases for different actors. The entire solution is targeted at Glex, which will be able to utilize it to create digital twins for their clients. The case-specific digital twin generated through the use of the pipeline utilizing the Smeaheia dataset will be targeted at professionals within the field of Carbon Capture & Storage. This ranges from geologists to executives.

1.3 Group Background

We will shortly present the background of our group, both academically and motivationally speaking.

1.3.1 Academic Background

All the group members are students in the Bachelor in Programming programme, taught at NTNU Gjøvik. All the courses we have had throughout the programme have contributed to our overall level of knowledge and is thus contributing to our work on this project. There are some some courses that we deem especially relevant:

PROG2053 - Web Technologies	Overall knowledge of HTMl, CSS & JavaScript
PROG2002 - Graphic Programming	Low level graphic programming
PROG2005 - Cloud Technologies	Overall knowledge of cloud based programming
IMT3603 - Game Programming	Development of interactive applications
PROG2052 - Integration Project	Software development and project management
PROG2006 - Advanced Programming	General knowledge of programming
PROG2051 - Artificial Intelligence	Artificial Intelligence and machine learning

1.3.2 Motivations

When the group were reviewing the different project proposals, ambitious projects were in focus. We wanted to pick a project that would be both challenging and educational. We were intrigued by the idea of experimenting with new architectures and solutions within the field of digital twins, as it is a relatively unknown and uncovered field within computer science [12].

We established a great connection with the client early on. We had a great talk after the presentations at campus, which also served as a great incentive for choosing this project.

1.4 Delimitations

There are some delimitations in the project. The group is developing a solution capable of processing any dataset similar to Smeaheia, but due to limited access to data, this project will only feature a single geographic location, namely the Smeaheia storage site.

Carbon storage & Capture can be categorized into three different subsystems: Capture, Transport and Storage. This project will only focus on the sub-system "Storage".

Even though the solution requires data managers to handle the data associated with the different digital twins, the group will not develop the interface for this functionality. For this purpose, Azure's pre-existing tools such as Azure Storage Explorer will be utilized.

1.5 Constraints

There are three different types of constraints we have to pay attention to.

1.5.1 Time

The first constraint we have is time. The task at hand is both large and complex, and it is easy to get carried away. We therefore need to follow the development plan carefully, to ensure that we keep on track. The project must be finalized within the 20th of May 2022.

1.5.2 Hardware and Software

Hardware is rapidly evolving, any requirements in this regards can therefore be deprecated quickly. Instead, we will utilize some set browser versions as requirements for the web interface of the solution.

Users must utilize hardware supporting, and software version matching these browsers:

- Firefox 4+
- Google Chrome 9+
- Opera 12+
- Safari 5.1+
- Microsoft Edge build 10240+

1.5.3 Legal

The Smeaheia dataset comes with a license[44], whose terms we have to abide by. Some other assets have been acquired under creative commons licenses. Mainly, this allows us to use the dataset, on the condition that we credit Gassnova and Equinor.

1.6 Project Goals

1.6.1 Result Goals

- A general processing solution that facilitates generation of geological digital twins from data.

- A digital twin, generated through the developed solution using the Smeaheia dataset and made accessible through a web interface.
- The solution should be able to process different data types, such as Surfaces, Horizons, Faults and Well logs.
- The solution should be able to support real-time data.
- The solution should have plotting and visualization functionality to display processed data in graphs, both in UI panels and in the scene.
- The solution must be modular and expandable.

1.6.2 Effect Goals

Qualitative Goals

- Determine the advantages/disadvantages of an alternative workflow in terms of the visualization of digital twins, compared to Glex's current solution.
- Increase the accessibility of high fidelity visualizations.
- Ease the process of generating geographic digital twins.

Quantitative Goals

- Reduce the time it takes to show implementation to clients.
- Increase the accessibility of digital twins.
- Reduce the hardware requirements required to view digital twins.

Learning Goals

- Learn more about Scrum based development.
- Learn more about Test Driven Development (TDD).
- Learn more about Git best practices, continuous development and continuous deployment.
- Learn more about project management.
- Learn more about microservices.
- Learn more about complex cloud architectures.

1.7 Group organization

All the group members are primarily developers, in addition to this they have some secondary roles. This means that the group members are of the Scrum team, and are responsible for taking on and completing issues from the sprint backlog. The completion of all issues in each sprint is a shared responsibility for the whole Scrum team.

Sebastian Lindtvedt is the group leader. This entails ensuring the project progresses forward in a satisfactory manner. The group leader will also be responsible for solving internal conflicts.

Dennis Kristiansen is the Scrum master. The responsibilities of a Scrum master includes: maintaining the issue backlog, training and coaching the other members on Scrum and agile development, removing barriers for the Scrum team, and making sure the Scrum events take place and serve their purpose.

Salvador Bascunan is mainly responsible for communication with Glex. This will entail keeping track of our weekly meeting invitations, schedule, change date or time if needed, and bringing topics between us and Glex if needed.

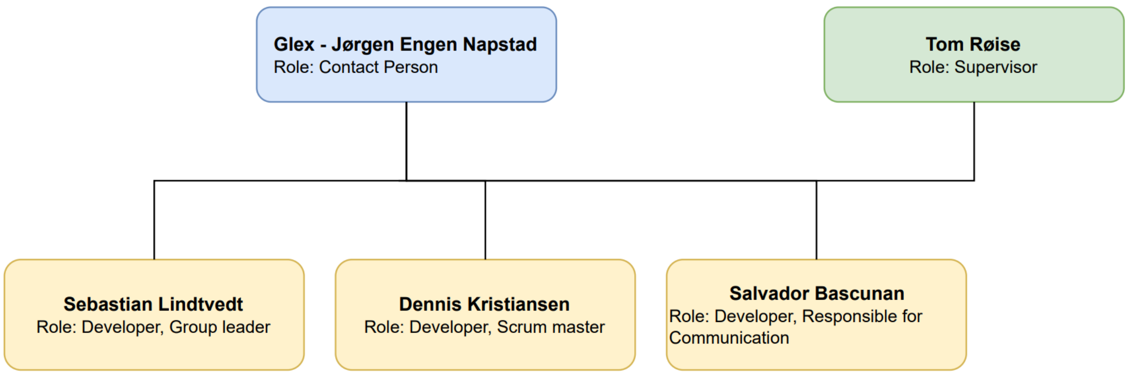

Glex - Jørgen Engen Napstad
Role: Contact Person
Tom Røise
Role: Supervisor
Sebastian Lindtvedt
Role: Developer, Group leader
Dennis Kristiansen
Role: Developer, Scrum master
Salvador Bascunan
Role: Developer, Responsible for Communication

Figure 4: Diagram of different roles in the project.

1.8 Thesis Structure

This thesis is structured into ten different chapters, each containing several sections. In addition to this, the thesis includes a list of figures, tables, listings, and a glossary which can be found above the introduction. We used a template from NTNU [46], made by Jon Arnt Kårstad. This provided us with the title page and examples of how chapters and section could be implemented. Our chapter structure is based on internal discussions within the group and feedback from our supervisor. The structure aims to facilitate what the group members deemed an optimal reading experience of the thesis as a whole. Below is a short description of each chapter in chronological order.

1. **Introduction**: Contains a introduction to the client, the task and the group.

2. **Requirements**: Covers the requirements for the solution, as well as specifications and requirements

from the client.

3. **Development Process**: Contains the planning and process of the development.
4. **Graphical User Interface**: Covers the development of the graphical user interface.
5. **Technical Design**: Contains the groups technical design decisions regarding each part of the solution.
6. **Testing**: Covers our approach to test driven development, and other aspects of testing in the project.
7. **Implementation**: Contains a more detailed description of how the different aspects of the solution were implemented.
8. **Deployment**: Covers how the solution was deployed to Azure, and our integration of CI and CD in the development process.
9. **Discussion**: Contains reflections of how the technical design and implementation of the different aspects of the solution worked.
10. **Conclusion**: Covers our own thoughts on the project process, product and future development. It also covers a few final words on our learning experience in this bachelor's project.

2 Requirements

The requirements for this project are a result of a collaboration process between the group and the client. Glex made it clear from the beginning that they wanted a resulting digital twin, but they were unsure about what they wanted the digital twin to be a virtual representation of. Through our first meetings we exchanged some ideas, like:

- Offshore Windmill Park
- Carbon Capture and Storage
- Carbon Transportation

While discussing the different ideas with Glex, we identified that a more general digital twin generation solution could be created for this project. This entailed creating separate modules to process and generate visualizations based on the dataset. Glex supported the idea, but were concerned about the scope, believing the project would be too large. Our main argument was that the resulting solution would allow Glex to implement new digital twins from other datasets much faster, and save valuable development time for the team and their clients in the future.

It was important for Glex that this digital twin would be derived from real world data. This was the main reason for why the Smeaheia Carbon Capture and Storage dataset was selected. Even though the Smeaheia site is not in production as of today (May 2022), it would still provide great insights into how a real-world digital twin potentially could look like. In the dataset, we had data for surfaces, horizons, faults and well data. Real-time data does not exist, as the site is not live yet. Real-time data will be mocked in this project.

Below we will cover the requirements we had for the cloud-native solution designed to handle the dataset, as well as the specifications and requirements we received from Glex for the Smeaheia Carbon Capture and Storage site specifically.

Cloud-Native Solution

As previously mentioned, the specific project case was a result of continuous conversations with Glex during the project planning period. This made the group recognize that there was room to build a modular system for generating the different aspects of the digital twin. In reality, this would look like this: A new file is uploaded, the file then gets automatically parsed, processed and served to the web app, allowing a client to get quick and flexible access to a visualized version of the data.

Cloud-Native Solution Requirements

- Several microservices for parsing and processing different data formats such as surfaces, horizons, faults and well data.

By creating modular solutions for the parsing and processing of several different data types we provide Glex with flexibility in regard to processing of new datasets in the future.

- All the microservices must be modular, loosely coupled and testable.

  By creating microservices with these properties, we establish a environment which in itself is easily extensible for further development.

Smeaheia Specification

The main objective of the specification was to build an interactive digital twin of the Smeaheia Carbon Storage site. This is based on a subset of the existing historic data, as well as real-time data simulating an active CO_2 injection process. This meant that the digital twin should feature a 3D visualization of the area, the Smeaheia Carbon Storage site. This included the well trajectories for the wellbores:

- 32/4-1 T2
- 32/4-1
- 32/2-1

It would also include a 3D seismic cube that represents the two areas, horizons and surfaces of the storage site, and the fault sticks and well data. The well data would consist of different types of data such as pressure, porosity, salinity, etc. In addition to this, the digital twin would include reports on crucial and relevant information based on subsurface evaluations and interpretations.

This project showcases the possibilities that are enabled by creating digital twins for carbon storage. The digital twin may later be integrated into the Glex Digital Platform and put in a regional context. This would enable new ways of working and instant access to relevant data and knowledge for the end-users. The end-users would be companies working with Carbon Capture and Storage on the Norwegian Continental Shelf today. These are companies like:

- Shell
- Equinor
- Horisont Energi
- Northern Lights JV
- Vår Energi

Smeaheia Requirement 1 - Historical data to be implemented

For the digital twin, one of the requirements was to implement historical data. This is static data that would be displayed either through graphs or generated into geometry. We will go through each of the requirement and briefly explain what the requirement was, and point to the section where the development process of the implementation is further explained.

- **3D Cube outlines for TNE01 and GN1101**

  These cubes would outline the areas in which the wells, faults, and surfaces would be. In this dataset we only had data that would cover the GN1101 cube. Both cube outlines are implemented in the scene.

- **Interpreted surfaces from 3D seismic**

  The surface files we received through the dataset were point clouds. The files had to be parsed and processed into generated geometry. We will explain how this was conducted in the Surfaces section of the implementation.

- **Faults**

  The fault files in the dataset had to be parsed and processed into geometry. This is further explained in the Faults section of the implementation.

- **Well locations and Trajectory surveys for wellbores 32/2-1, 32/4-1 and 32/4-1 T2**

  The well data for the wellbores contained the location and trajectory for the wells in the digital twin. This data was contained within an Excel spreadsheet. Details regarding how these files were parsed and processed is located within the Well logs section of the implementation.

- **Composite log data for all wellbores**

  The composite logs in the dataset were in LAS format. Details regarding how these files were parsed and processed is located within the Well logs section of the implementation.

- **Lithostratigraphy and Chronostratigraphy data**

  The lithostratigraphy and chronostratigraphy data had to be retrieved from the Excel spreadsheets. Details regarding how these files were parsed and processed is located within the Well logs section of the implementation.

- **Formation Pressure and Core Porosity/Permeability data**

  We received the formation pressure and core porosity/permeability data through an Excel spreadsheet. Details regarding how these files were parsed, processed and plotted is located within the Well logs section of the implementation.

Smeaheia Requirement 2 - Plotting and Visualization functionality

Our second requirement was to plot and visualize the dataset in different ways. The data that were to be visualized in this manner:

- Composite Log Data

- Lithostratigraphy
- Chronostratigraphy
- Porosity/Permeability
- Formation Pressure

The second part of this requirement was displaying the historical data as a graph. The data that were to be visualized in this manner:

- Formation Pressure vs Depth (MD and TVDSS)
- Porosity vs Permeability
- Porosity vs Depth (MD and TVDSS)
- Permeability vs Depth (MD and TVDSS)
- Lithostratigraphy data should be indicated in these plots

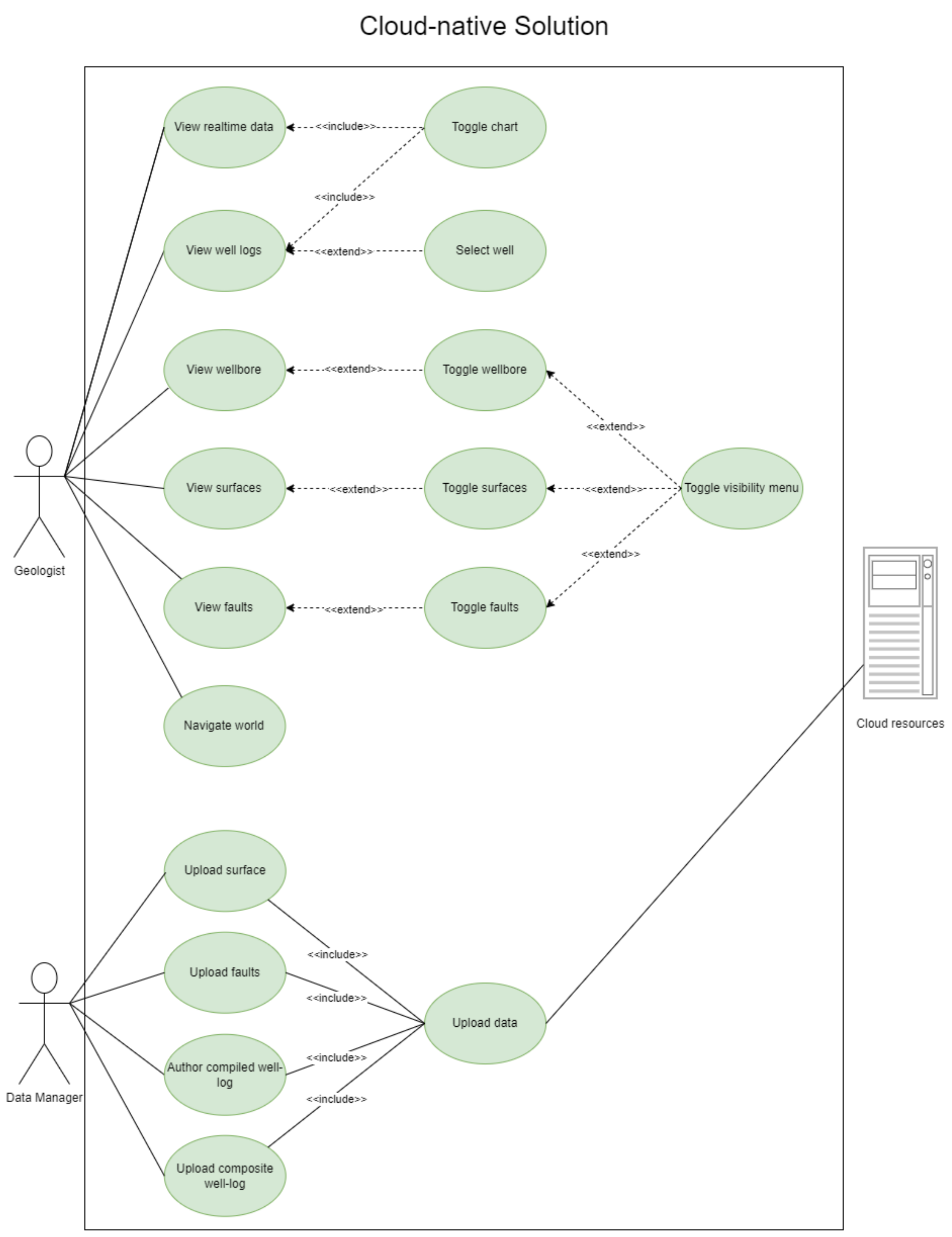
Smeaheia Requirement 3 - Real-time data

Since the Smeaheia site is not in active production today, no real-time data currently exists. This data had to be mocked. The real-time data that would have existed had the site been active would have looked like this:

- Well Pressure
- Flow Rate data

2.1 Use Case

We created a use-case diagram based on the requirements we received from Glex. The diagram includes two different actors: Geologist and Data manager. The geologist operates within the web app of the use-case, while the data manager operates outside of the web app, on the solution itself.



2.1.1 Actors

- **Geologist**: A geologist that will use the digital twin for analyzing and research. The geologist uses digital twin software on a regular basis, several times per week. A geologist know how software like ours work and how to utilize it.
- **Data Manager**: A data manager that will add new data to the storage account to trigger functionality.

The data manager has previous knowledge of the storage system to upload the files, and know what format the files needs to be in.

2.1.2 User stories

The user stories for geologists are derived issues from the project backlog. These are meant to describe functionality and actions that a geologist may want to perform/achieve. We have also created a specific user story for the data manager. The only pre-condition for the user stories is that the actors needs to have an internet connection.

User Story:	As a geologist, I want to visualize well-logs
Actor:	Geologist
Goal:	View the well logs of the digital twin
Description:	The user clicks on a well on the scene. Then clicks on the graphs icon at the top center of the screen. A panel will open on the left with the well logs data.

User Story:	As a geologist, I want to visualize faults
Actor:	Geologist
Goal:	View the fault sticks of the digital twin
Description:	The user clicks on the visibility menu icon at the top center of the screen. A panel will open on the right with the fault stick visibility check boxes. The user can click these checkboxes to make certain faults visible or invisible.

User Story:	As a geologist, I want to visualize surfaces
Actor:	Geologist
Goal:	View the surfaces of the digital twin
Description:	The user clicks on the visibility menu icon at the top center of the screen. A panel will open on the right with the surface names. Next to these names there is a checkbox to turn either on or off the visibility.

User Story:	As a geologist, I want to view wellbore
Actor:	Geologist
Goal:	View the wellbores of the digital twin
Description:	The user clicks on the visibility menu icon at the top center of the screen. A panel will open on the right with the wellbore names. Below these names there is a checkbox to turn on either of the options to view default, chrono, formations and groups.

User Story:	As a geologist, I want to view realtime data
Actor:	Geologist
Goal:	View the realtime data of the digital twin
Description:	The user clicks on the graph icon at the top center of the screen. A panel will open on the left with the realtime data displaying.

User Story:	As a data manager, I want to upload data to visualize in the web-app
Actor:	Data Manager
Goal:	Add a data file to storage that will trigger the service and visualize it in the web-app
Description:	The user adds a data file to the Azure Storage Account. Authentication is taken care of by Azure. This will trigger the services that will process and generate a visualization of the data in the web-app.

2.2 Performance

The web-app requires an internet connection. We have also established some minimal requirements for the computer/laptop running the service:

- 4-core Intel Core i5-8265U
- 8 GB RAM DDR3 2133MHz
- Intel HD Graphics 620
- 1920x1080 resolution screen

Having a dedicated GPU is in most cases beneficial, but in our case it has not proven to make a significant difference. We have discovered that newer hardware, computer/laptop, has shown better results in terms of performance. In some cases, a newer computer/laptop with integrated graphics proved to run the service better than a two year old computer/laptop with dedicated graphics.

2.3 Security

In terms of security, the solution must follow all industry standards and best practices. Security concerns that arose during development is covered in the relevant sections.

3 Development Process

The group will explain some of the key characteristics of the project in general and how this affected the initial plan. The initial development plan will be covered, and then compared to how the actual development turned out. In addition to this, a summary of the project management tools utilized will be covered. We will showcase the initial Gantt diagram and the final one. The final Gantt diagram reflects the changes that occurred during development.

The focus of this chapter is not to go into depth about technical details, but to give a summary of **what** was implemented **when**. Technical details around the implementation is located in Chapter 7.

3.1 Project Characteristics

This project was the largest and most complex development task the group had taken on so far. This was something that had to be considered when planning out the project. It was of utmost importance that the process was well organized from start to finish to ensure that things progressed in a consistent manner. There were many moving parts and a broad range of different tasks happening at the same time. There were several stakeholders involved, which had different expectations at different times, such as our supervisor and the client. The key thing we drew from these characteristics was that the project had to be structured.

Some of the stakeholders representing the client did not have any technical background in terms of software development. It was therefore essential to keep a close dialogue with them to avoid any miscommunication. This further strengthened the need for a structured development process.

The project naturally facilitated a variety of different programming languages and frameworks. The reason for this was due to tasks such as parsing of different data formats, which was conducted at non-scheduled time periods. An example of such formats is geological data. We did not know when new data would be presented, but we had to process it whenever it was released. This meant that the processes for conversion, etc. had to be run multiple times. When creating several different microservices, an agile way to work was beneficial.

Due to the reasons mentioned above, the project had to be conducted in an agile, but structured way. This is because there were several sub systems that needed to be implemented, which benefited from an agile methodology, and several different stakeholders that wanted to monitor the progress of the project which benefited from a solid structure.

3.2 Software Development Model

The project facilitated an agile methodology. We needed to create several different microservices with varying languages and frameworks. An incremental, rapid development cycle was beneficial in this regard.

Each microservice was developed as an individual software project. This meant that each microservice was planned, designed, implemented and tested in its own cycle.

We considered two different Software Development Models to use: Kanban and Scrum. Kanban consists of visualizing all the tasks of the project into a board. There is a maximum amount of tasks at any given time, and each team member is free to pick any task they see fit. Kanban is very flexible, and could have been a good solution for this project, given that we were a small team that communicated frequently. The problem with Kanban is that it lacks some of the structure Scrum provides. Kanban could very easily have gotten out of hand, leading to a messy development environment.

A Scrum project is split into sprints. During each sprint, several Scrum events take place. Sprint review, sprint planning, daily stand-ups, etc. These meetings provide the project with structure. By scheduling several preplanned meetings in advance, it became much easier to actually see them through.

By comparing these two models, we concluded that Scrum provided us with the structure required for a project of this scale. We did however like the board from Kanban. We therefore planned to implement the board alongside the structure of Scrum.

We followed Scrum as our development model. We organized the project into 2-week long sprints, with two stand-up meetings each week (totaling to four each sprint). During the first week of each sprint we had two special meetings. We met with the client every other Tuesday at 1400. This meeting served as a combination of sprint review/sprint planning meeting. During the first half of the meeting, we presented the progress from the previous sprint. This allowed us to receive rapid feedback on our development work, and gave us good insights regarding where to concentrate our energy in the coming sprint. The second part of the meeting was the planning part. Here, Glex presented their perspective on the development work. In addition to presenting their perspective on the development work, they described/showcased features of which they wanted us to incorporate into the project. After the meetings with Glex, we processed the information received and converted them into relevant issues for the upcoming sprint. Secondly, we had a meeting with our supervisor, who gave us great feedback on different aspects of our project, providing insights from a outwards perspective. This feedback helped stake out the general direction of the project, allowing us to concentrate our energy on the more academically important aspects of the project.

We structured the issues into user stories. Issues typically looked like: ‘As a geologist, I want to be able to visualize fault sticks and their faults’, ‘As a geologist, I want to visualize well logs alongside the well trajectory’, etc. The reason for choosing this approach was due to our previous experiences with sprint planning. Sprint planning was hard, and especially hard due to the fact that we wanted to create good issues that lasted an entire sprint. By going for a user story oriented approach, it became easier for us to create issues at the beginning of a sprint. It is natural that new, small issues arise during development. Since we already had our more general user stories, these smaller child issues were added to the parent issues consecutively.

The issues were primarily assigned during the sprint planning meetings, but issues were also assigned during sprint stand-ups. Each team member stood free to dispose the tasks they have been assigned freely. Meaning that it was each member's responsibility to ensure that each task was completed in the allocated time frame. All issues had to pass through a review phase, before finally being marked as done. The review process was conducted manually in the beginning, and partially through code tests as the project progressed.

When a team member picked up an issue, they first transitioned the issue from "Todo" to "In progress". If other issues blocked the progress, the member transitioned the issue into a "Blocked" state. When the issue was completed, the issue was transitioned into "For review", where the other group members reviewed the issue before finally transitioning the issue into the "Done" state.

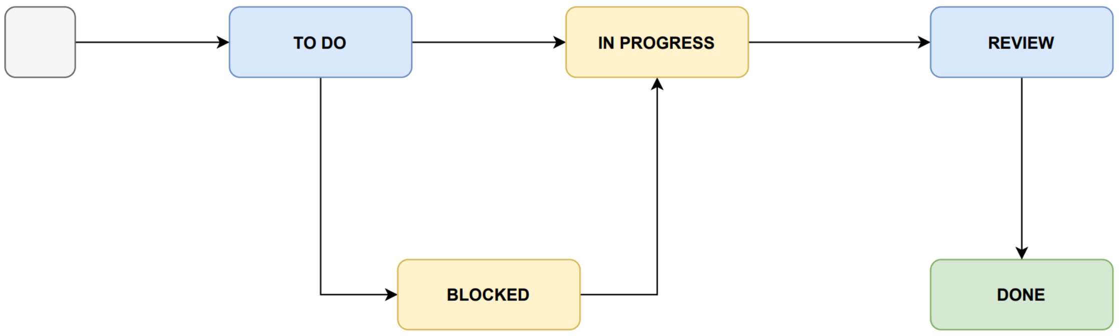


Figure 5: The different stages of issues

We conducted sprint retrospective meetings during the first meeting of each new sprint. We utilized these meetings to discover things that could be improved for further project work. In the beginning, a few key problems were brought up and discussed. Some examples of things we discussed:

Problem	Solution
We feel like we spend too much time in unnecessary meetings.	Reduce meeting frequency, but increase the prework before each meeting.
Issues are too broad and general	Spend more time planning issues

We brought up a broader range of issues at the retrospective meetings in the beginning of the project work. We believe that the amount of issues arisen was reduced throughout the project due to the fact that discovered issues were handled and fixed. The retrospective meetings were a nice way to bring up project related issues in a structured manner.

As a summary, our meeting schedule looked like this:

Week	Mon	Tue	Wed	Thu	Fri
1st	• Daily Scrum • Retrospective	• Client Meeting • Sprint Review • Sprint Planning	Supervisor meeting	Daily Scrum	No meetings
2nd	Daily Scrum	No meetings	No meetings	Daily Scrum	No meetings

Table 1: Overview of sprint structure

3.3 Project management tools

We utilized the Atlassian suite for most tasks related to the development process, with the exception of version control, which was done through GitHub. The main tool for keeping track of the project was Jira [6]. We organized all of the issues of the project into a backlog in Jira.

We utilized a tool called Tempo for time tracking. Tempo is tightly integrated into Jira and the Atlassian suite as a whole. The use of Tempo facilitated a common way for us to track time spent on the project. There also exists extensions for editors such as VSCode, making it easy to track time spent coding.

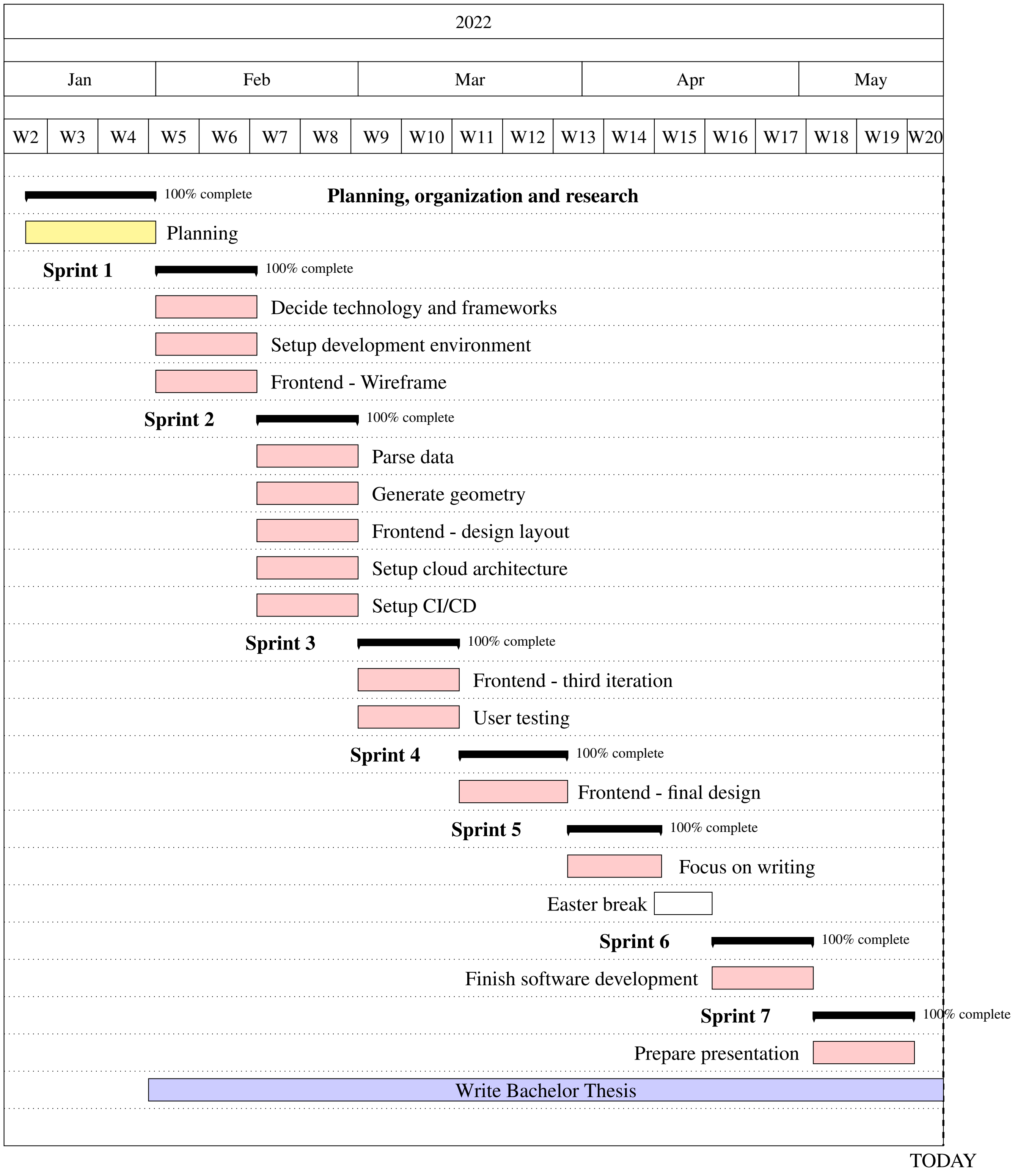
For project pages we used a tool called Confluence. Confluence allows all members of the group to cooperate on documents in the cloud. This was very handy in meetings and other activities that requires some sort of note taking/overview tracking.

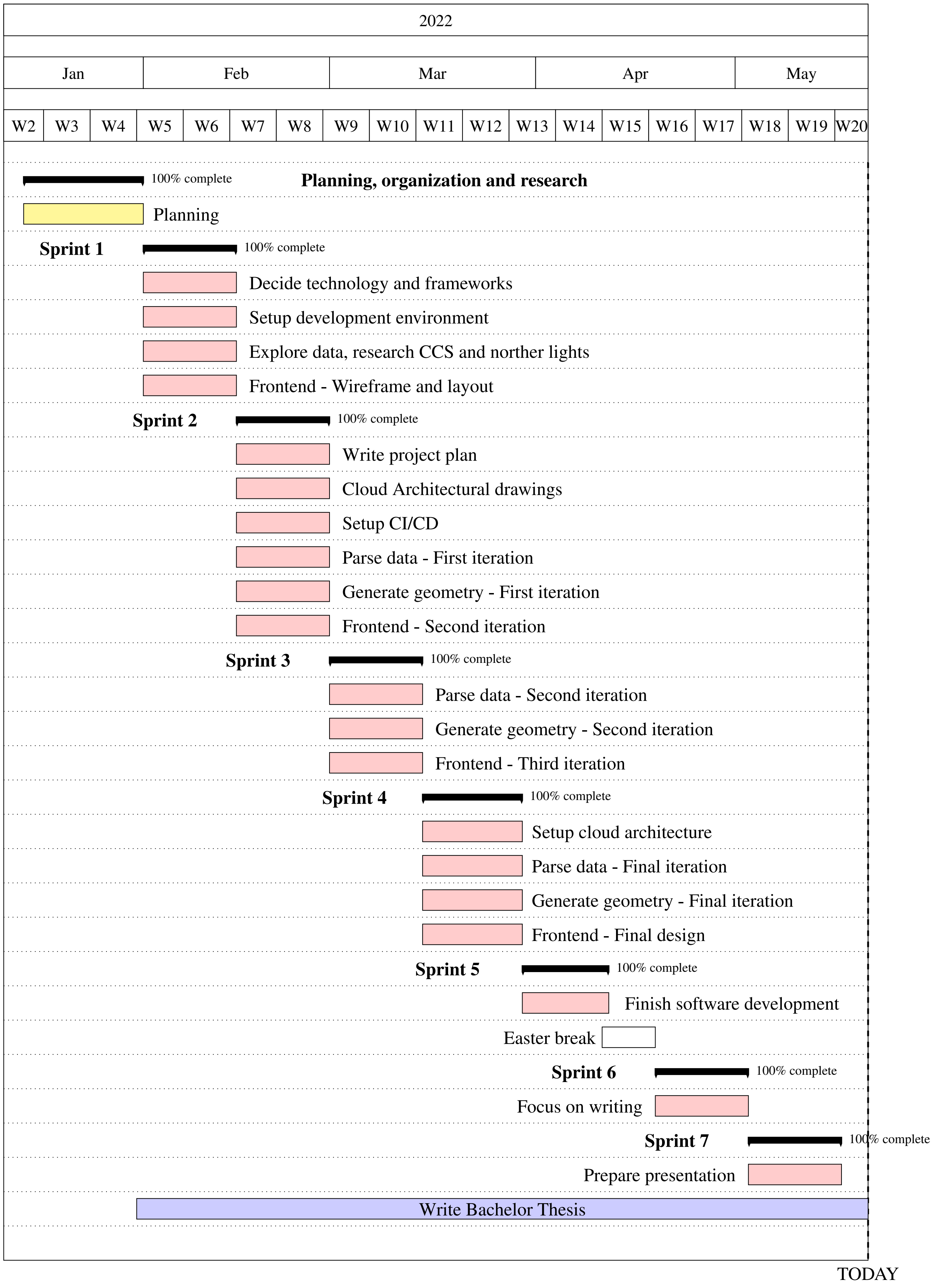
3.4 Version Control and Code Organization

We choose to use GitHub as the collaboration tool for the project's source code. The project's code is organized as subprojects in a monorepo. Individual parts of our project like the web-app, the REST API, the services, etc. are all stored as separate folders in one larger git repository. The advantages of this approach is that we get one single source of truth, it is easier to share and reuse common code amongst subprojects, and we can more easily make atomic changes across subprojects, like changing the REST API, which requires changing API consumers as well[32]. This approach also allowed us to set up larger test runners, facilitating running all the test of the project as a whole at the same time.

3.5 Gantt Diagrams & Sprint Breakdowns

We initially planned on a sequentially based development process. What we mean by that is that we estimated that we were going to finalize different parts of the 'pipeline' at different stages. As the project turned out, it became more of a continuous improvement cycle where several modules were continuously improved to match the project's progress. One example here is the parsing, where aspects came up during the deployment stage that encouraged us to revisit the parsers to improve them. Because of this, some issues were continuously re-added to the project backlog for further iterations. We planned to finalize the development work before the Easter break. Due to the fact that we did continuous improvements of previous iterations of code, the development work spanned over a longer period of time than initially planned. This resulted in five sprints with development work in focus.




2022
Jan
Feb
Mar
Apr
May
W2
W3
W4
W5
W6
W7
W8
W9
W10
W11
W12
W13
W14
W15
W16
W17
W18
W19
W20
100% complete
Planning, organization and research
Planning
Sprint 1
100% complete
Decide technology and frameworks
Setup development environment
Explore data, research CCS and norther lights
Frontend - Wireframe and layout
Sprint 2
100% complete
Write project plan
Cloud Architectural drawings
Setup CI/CD
Parse data - First iteration
Generate geometry - First iteration
Frontend - Second iteration
Sprint 3
100% complete
Parse data - Second iteration
Generate geometry - Second iteration
Frontend - Third iteration
Sprint 4
100% complete
Setup cloud architecture
Parse data - Final iteration
Generate geometry - Final iteration
Frontend - Final design
Sprint 5
100% complete
Finish software development
Easter break
Sprint 6
100% complete
Focus on writing
Sprint 7
100% complete
Prepare presentation
Write Bachelor Thesis
TODAY

3.5.1 Planning phase

We had frequent meetings with the client in the planning phase. The main focus in the phase was to establish the specific case of which to base the entire project on. The project plan was written concurrently alongside the planning.

3.5.2 Sprint 1: Technology choices and setup

During sprint 1, the majority of our focus was targeted at research of relevant technologies, the setup of development environments, and the wire-framing of the web app. Several small "demo" projects were created during this phase, to test the capabilities of different languages and frameworks. One example of this is a CSV-file parser that was written in Python.

3.5.3 Sprint 2: Cloud Architecture drawing

During this sprint, we created the first iteration of the web app's design. We researched Azure and created architectural drawings for the general cloud infrastructure. We started working on the different user stories: Fault sticks, Surfaces and Well logs. The primary focus was parsing of data at this stage, but some geometry generation of surfaces were performed.

3.5.4 Sprint 3: CI/CD

During sprint 3, we established the initial setup of the CI/CD pipeline. After this had been set up, code that was pushed to the repository was automatically deployed to Azure. Work also continued on the user stories, in addition to data processing and geometry generation.

3.5.5 Sprint 4: Web app

An MVP was finalized, all major components were implemented in a minimal, but functional manner. The primary focus of the MVP was to showcase the project progression to the client. Everything was starting to come together, which uncovered a new set of issues.

3.5.6 Sprint 5: Finish software development

During sprint 5, the finalization of the software development was in focus. This meant solving various issues and bugs with the implementation, polishing the UI, and updating the documentation and READMEs with the latest information.

3.5.7 Sprint 6 & 7: Report writing

During spring 6 and 7 the writing of the thesis itself was in focus. We went through several iterations with our supervisor gathering feedback in addition to meetings with the client about regarding sensitive contents in the report. We did this to ensure all the stakeholders were satisfied with the end result.

4 Graphical User Interface

We will talk about our development process regarding the visualization of the different parts of the digital twin. How we developed the different prototypes, and how we worked with the feedback we received from Glex.

4.1 Figma

Our first iterations of the GUI were done in Figma. Here, we were able to draft some suggestions on how the GUI could look. We started by creating a top menu on the scene.

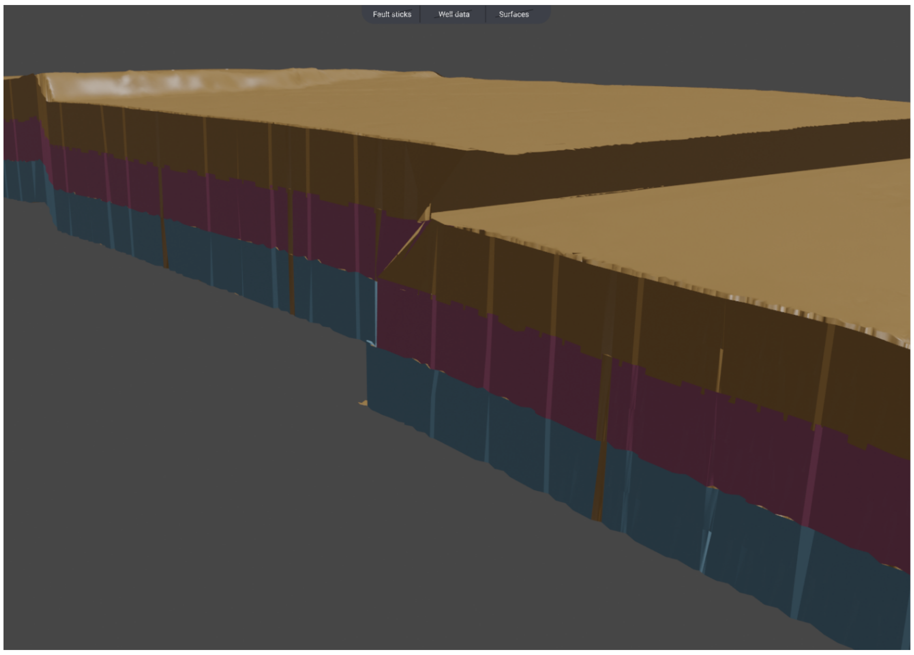


Figure 6: The first prototype, used in the planning phase

Our idea here was to have a top menu with the most essential buttons available. We chose to display the fault stick, well data and surfaces buttons because these are parts we knew were going to be visualized in the web-app.

When it came to presenting the fault sticks, we knew that the requirement was that the faults had to be visualized in the scene, and also include a menu to toggle specific faults on and off. We decided to present this menu as a side panel. This side panel would open when the fault sticks button was pressed on the top menu.

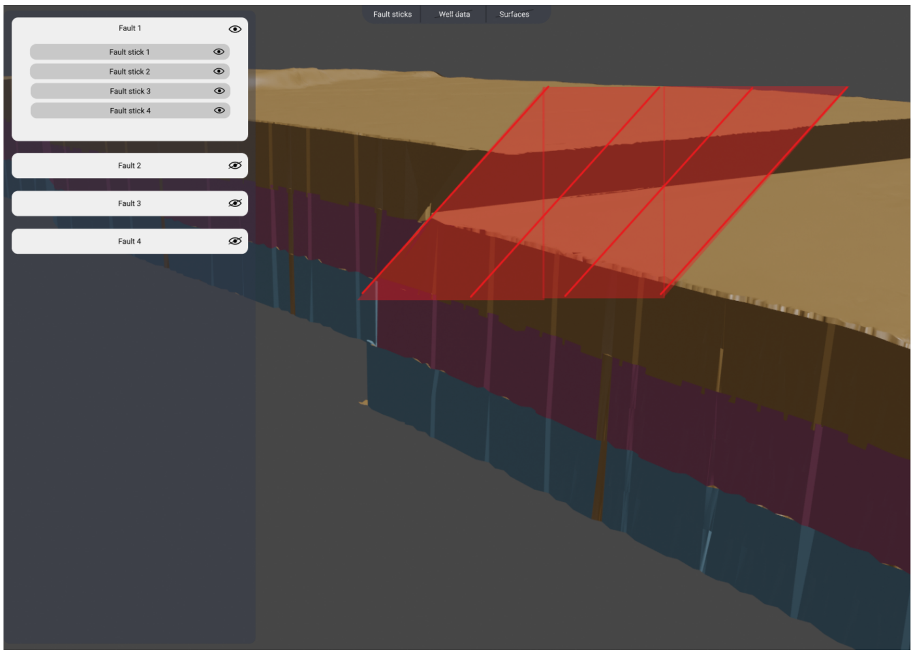


Figure 7: Prototype for faults menu

We had a similar idea for the well data. This data was going to be plotted alongside the well trajectory, and with a side panel to chose which data was to be displayed.

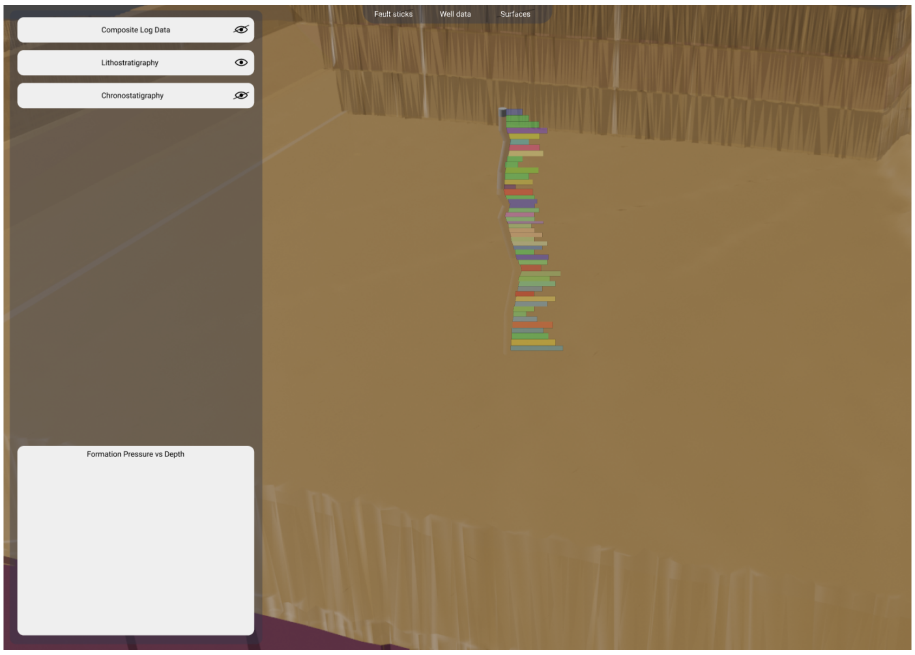


Figure 8: Prototype for well-logs menu

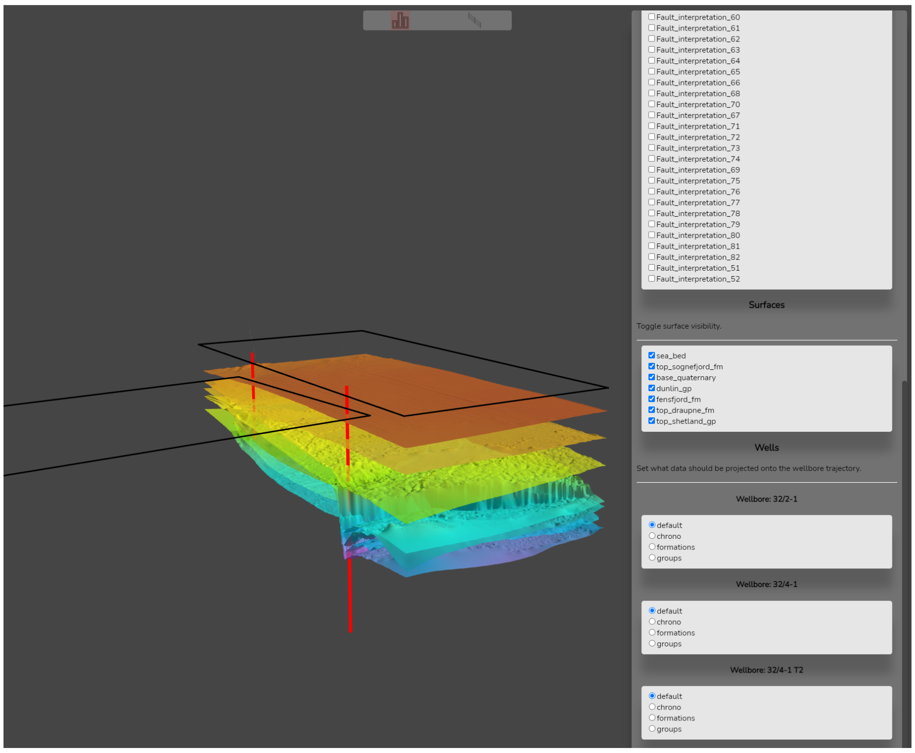
Our plan for visualizing the surfaces on the web-app were similar to the fault sticks and well data, so we decided against creating a page on Figma specifically for this. The plan for the surfaces was that they would have a menu on the side panel to toggle each one on and off. This was also the case for horizons.

We showed Glex our Figma iterations and got positive feedback. This lead us to begin the GUI work on the web-app. From here, we could show Glex the iterations on the web-app and adjust the GUI based on the feedback.

4.2 Improvements

After each sprint in the development process, we did some manual user testing with Glex. This allowed us to gather useful feedback and make improvement during the development. With the experience Glex has in the geology field, we were able to determine how the user interface should be organized to best fit their needs.

Organizing the GUI was the first thing we performed after receiving feedback. Glex suggested that all of the visibility toggles should be grouped into one side panel, and the rest of the data should grouped in another. So, for instance, the visibility toggles could be grouped in the right side panel, and the data on the left panel. These panels are collapsible via the panel buttons on the top center of the scene.



☐ Fault_interpretation_60
☐ Fault_interpretation_61
☐ Fault_interpretation_62
☐ Fault_interpretation_63
☐ Fault_interpretation_64
☐ Fault_interpretation_65
☐ Fault_interpretation_66
☐ Fault_interpretation_68
☐ Fault_interpretation_70
☐ Fault_interpretation_67
☐ Fault_interpretation_71
☐ Fault_interpretation_72
☐ Fault_interpretation_73
☐ Fault_interpretation_74
☐ Fault_interpretation_69
☐ Fault_interpretation_75
☐ Fault_interpretation_76
☐ Fault_interpretation_77
☐ Fault_interpretation_78
☐ Fault_interpretation_79
☐ Fault_interpretation_80
☐ Fault_interpretation_81
☐ Fault_interpretation_82
☐ Fault_interpretation_51
☐ Fault_interpretation_52

Surfaces

Toggle surface visibility.

☑ sea_bed
☑ top_sognefjord_fm
☑ base_quaternary
☑ dunlin_gp
☑ fensfjord_fm
☑ top_draupne_fm
☑ top_shetland_gp

Wells

Set what data should be projected onto the wellbore trajectory.

Wellbore: 32/2-1

◉ default
○ chrono
○ formations
○ groups

Wellbore: 32/4-1

◉ default
○ chrono
○ formations
○ groups

Wellbore: 32/4-1 T2

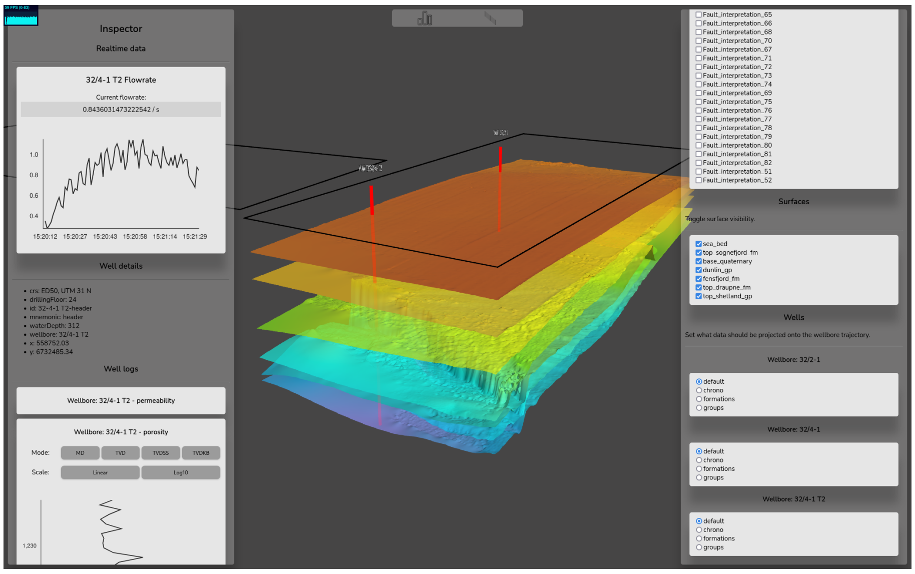
◉ default
○ chrono
○ formations
○ groups

Figure 9: Showing the right side panel with toggle visibility menu

Glex also mentioned that it would be helpful to be able to toggle between the chronostratigraphy and lithostratigraphy color palettes on the wells. This was added to the toggle menu as seen on the right side panel image above.

4.3 Final Result

After considering Glex's requirements and feedback during the development process, we came to a final version of the GUI.



Inspector

Realtime data

32/4-1 T2 Flowrate

Current flowrate:

0.8436031473222542 / s

Well details

- crs: ED50, UTM 31 N
- drillingFloor: 24
- id: 32-4-1 T2-header
- mnemonic: header
- waterDepth: 312
- wellbore: 32/4-1 T2
- x: 558752.03
- y: 6732485.34

Well logs

Wellbore: 32/4-1 T2 - permeability

Wellbore: 32/4-1 T2 - porosity

Mode: MD TVD TVDSS TVDKB

Scale: Linear Log10

- [ ] Fault_interpretation_65
- [ ] Fault_interpretation_66
- [ ] Fault_interpretation_68
- [ ] Fault_interpretation_70
- [ ] Fault_interpretation_67
- [ ] Fault_interpretation_71
- [ ] Fault_interpretation_72
- [ ] Fault_interpretation_73
- [ ] Fault_interpretation_74
- [ ] Fault_interpretation_69
- [ ] Fault_interpretation_75
- [ ] Fault_interpretation_76
- [ ] Fault_interpretation_77
- [ ] Fault_interpretation_78
- [ ] Fault_interpretation_79
- [ ] Fault_interpretation_80
- [ ] Fault_interpretation_81
- [ ] Fault_interpretation_82
- [ ] Fault_interpretation_51
- [ ] Fault_interpretation_52

Surfaces

Toggle surface visibility.

- [✓] sea_bed
- [✓] top_sognefjord_fm
- [✓] base_quaternary
- [✓] dunlin_gp
- [✓] fensfjord_fm
- [✓] top_draupne_fm
- [✓] top_shetland_gp

Wells

Set what data should be projected onto the wellbore trajectory.

Wellbore: 32/2-1

[✓ default] chrono formations groups

Wellbore: 32/4-1

[✓ default] chrono formations groups

Wellbore: 32/4-1 T2

[✓ default] chrono formations groups

Figure 10: Final GUI

Some of the requirements had to be revised and implemented in a different manner than initially planned. This was mostly regarding the plotting functionality alongside the well trajectory. We will revisit these matters in the Discussion section.

5 Technical Design

In this section we will explain our choices in regards to the architecture of the project as a whole, in addition to specific choices related to the different submodules.

5.1 System Architecture

The system architecture of our solution, and how it enables us to build a pipeline for building digital twins, is perhaps the most interesting aspect of this entire project. Not only does it serve to build a digital twin for the Smeaheia area, but it can be extended and expanded to include additional areas and form new digital twins, all within the same architecture and system.

Some of the key observations that we made were the following. The vast majority of the geological data relevant to the digital twin, is static. This means that the optimal architecture for this system is one that enables us to perform any heavy processing needed on this static data only once, without losing the ability to add more data later. This static data takes many forms, which all require their own form of processing, which can be done entirely separate from one another. This is why we decided to build a pipeline for processing geological data, and importing it into the system. Additionally, the system must also be able to handle real-time data, which is mocked for the time being. The system is designed and facilitated to easily transition from mocked data to actual real-time data, provided by sensors out in the real world.

The idea for how to build this pipeline efficiently was taken from the concept known as Materialized Views[26] from the field of databases. In a database, a view is a way to define aggregate queries, pulling and combining data from multiple tables, into a pseudo table every time the view is queried. Materialized Views turns this around, by materializing the view, the database will run all the computations every time the data changes, so that by the time it is queried, it is ready. Additionally, it also avoids unnecessary computations. For us, this signifies finding a way to run the processing of static data only when new data comes in, or when data changes, and storing the result in some kind of persistent storage.

While looking into the dataset we were provided, we also observed that we would need to deal with a large variety of different file formats and data types. Potentially requiring different approaches when it comes to parsing and processing. It would be best if we could build entirely separate and independent services for handling these different data types, such that we could use the technologies best suited for the individual tasks. This led us to consider the idea of using a microservice Architecture[27]. Using such an architecture would allow us to build entirely decoupled services which could perform parsing and processing of various data types, each in the most optimal way. The only requirements given to these services is that they expose inputs and outputs for the data and results.

We used ideas from the API gateway pattern[24] as inspiration when making a REST API, making up the business logic layer in a traditional n-layer architecture[28] from the perspective of the frontend. This REST API would serve the processed data from the individual processing services to the frontend, which would then visualize the data and present it to the user.

When combined, the overall architecture looked like this:

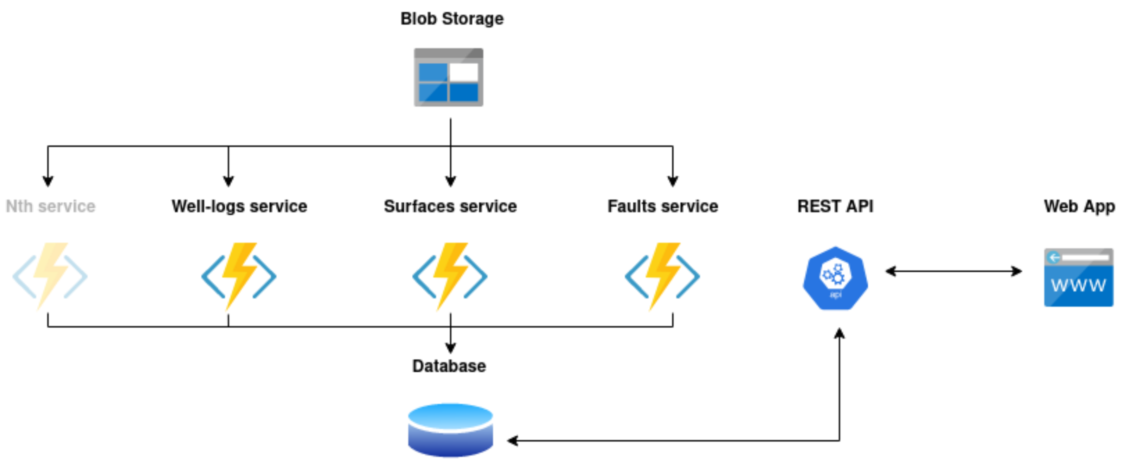


Figure 11: Simplified data flow diagram of the solution

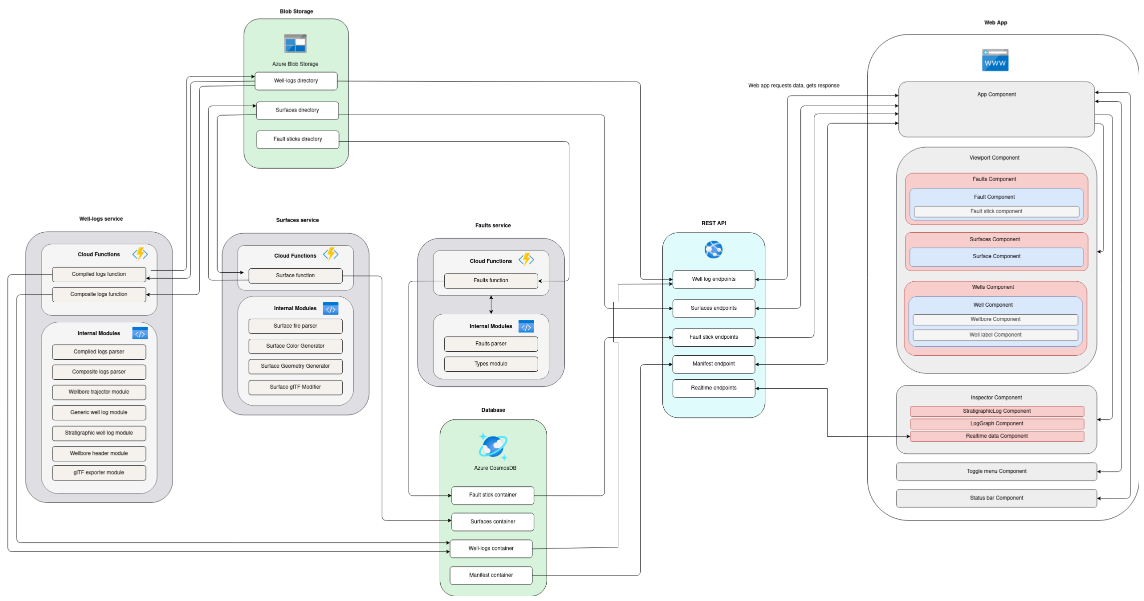


Figure 12: Detailed data flow diagram of the solution

5.2 Architecture Alternative

Our final architecture consisted of a presentation layer accessible to the user, with a REST API backend which fetches data and subsequentially serves it to the user. The processed data is stored in a database for ease of access, and data is not processed more than once. The microservices which processes the data are triggered by new file uploads. We did however consider a different approach in the planning phase. An architecture based on the Gateway Routing pattern was what we initially planned. The service would appear identical to the users regardless of the choice of architecture. This architecture would look more like this: The user accesses the presentation layer, which sends requests to the REST API. The REST

API would manage the different services and call them when necessary. The data requested by the users would be processed on demand. When data gets processed, it would be cached in-memory in the REST API. Subsequent calls to the REST API after the initial data processing would be fast and efficient. The architecture would have looked like this:

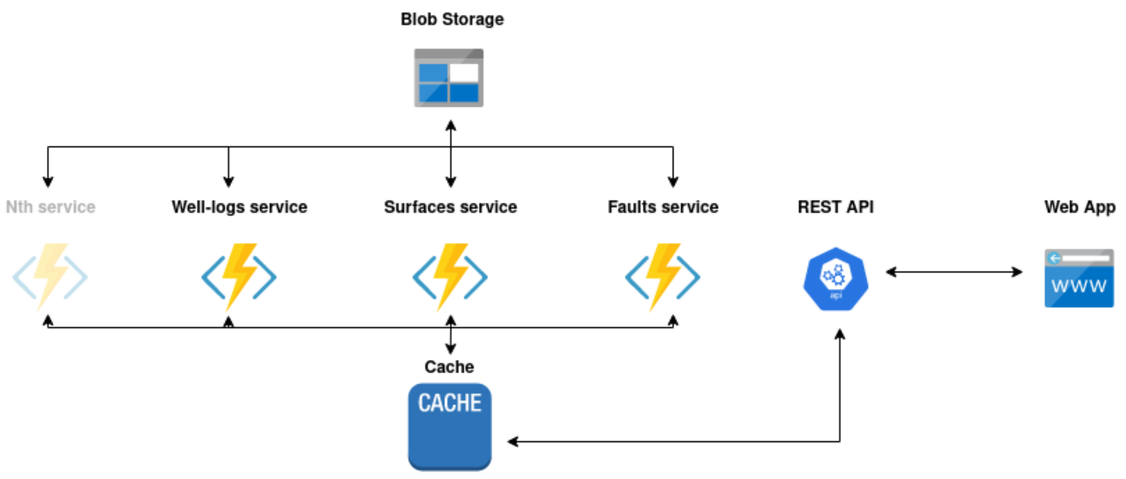


Figure 13: Alternative data flow diagram of the solution

There were some problems with this architecture: The first user to visit the page after a service restart would experience a much longer waiting time, as the geometric data would have to be re-generated. The same data would be processed over and over again, without any real need for it, as the data has most likely not been updated since the last processing. The reason we choose not to pursue this approach was due to the unnecessary data processing that this architecture would require.

5.3 Frontend

In the original project description, Glex proposed two alternative technologies for the frontend. Option one was using Unity, which is what Glex uses for their digital twins today. Unity is a full game engine, with a rich set of features, supporting a multitude of platforms, including desktop, mobile, and web[52]. Unity applications are developed in C#, with Unity providing an advanced interactive editor for making rich scenes. Going over the requirements for our specific project case, we also took into account importing geometry and plotting a chart with data on a 3D scene. Glex has extensive knowledge of the Unity engine. If we had chosen Unity, they would have been able to provide us with advice and knowledge. Picking Unity would have resulted in the exploratory parts of the bachelor's project being significantly reduced.

Option two was using a web based framework, like ThreeJS, to make a slightly different user experience than what Glex usually offers. This is a more ambitious option, where we take more of a risk by not knowing if the technology really supports the complexity of this project. Picking this approach would provide the group with a great learning opportunity. The knowledge gained from exploring this approach would be useful for Glex regarding further development of digital twins in the future.

Diving more into the second, option. ThreeJS is a lightweight, general-purpose 3D JavaScript library[50].

ThreeJS does not provide a built-in solution for UI creation, but since it is a JavaScript framework, we can utilize other web frameworks for this purpose. One solution is to combine the use of ThreeJS with the popular web development framework React. This will provide us with the best of both worlds, granting access to complex 3D rendering on one end, and React's performance, speed, and flexibility on the other end. Specifically, a library called React Three Fiber facilitates this combination of technologies in a sensible and efficient manner.

We researched whether or not the web based framework was capable of handling the requirements of this project. ThreeJS has many loaders for 3D models, but the documentation recommends importing GlTF files. This is because GlTF is focused on runtime asset delivery, it is compact to transmit and fast to load[49]. GlTF is a royalty-free specification for the efficient transmission and loading of 3D scenes and models by applications. It minimizes the size of 3D assets, and the runtime processing needed to unpack and use them[57]. Additionally, by using React with ThreeJS in React Three Fiber we get access to a collection of libraries, like Victory, which we can use to generate graphs using a collection of React components. Victory is a set of modular charting components for React and React Native. It let's us create one of a kind data visualizations with fully customizable styles and behaviors. It uses the same API for web and React Native applications, which makes it extensible to mobile in case that is an option in the future[4].

The main difference we identified between the two options, were how they impacted user experience. The main way to ship and distribute Unity based applications today, involves the user installing an application locally on their computer. What we want to explore with a web based approach is giving the user the ability to simply visit a website, and get the same information as before, but in a much more accessible form. This could perhaps make it easier to utilize the digital twin during a meeting, on a laptop, compared to having to utilize a desktop PC with preinstalled software to access the same information.

5.4 REST API

As established, we are building a solution based on microservices. Therefore, we needed a gateway for information to be communicated between the frontend, and the various backends and persistent storage, as we do not want the individual microservices to communicate with the outside world. This is based on the API Gateway pattern, which is a popular pattern in the world of microservices.

Since the concept of a REST API is so well established, we had a plethora of options when it came to the implementation languages and frameworks. There were mainly three options that were considered. Node based JavaScript implementation using *expressjs*, ASP.NET based implementation, or a Flask based implementation using Python. These options were all supported by Azure, which we needed for deployment. They all provide viable frameworks for implementing REST APIs. The choice eventually came down to economy of mechanism. We did not want to take on unnecessary complexity by adding a whole new language without proper reason, and we did not want to introduce unneeded task switching mental overhead for the group members. We knew that we had to utilize various Azure APIs from the other microservices already, so sticking with Python for interacting with these APIs made a lot of sense to us. Node and ASP.NET are both viable options, but for us, going with Python and Flask was the most sensible and economical option.

5.5 Persistent Storage

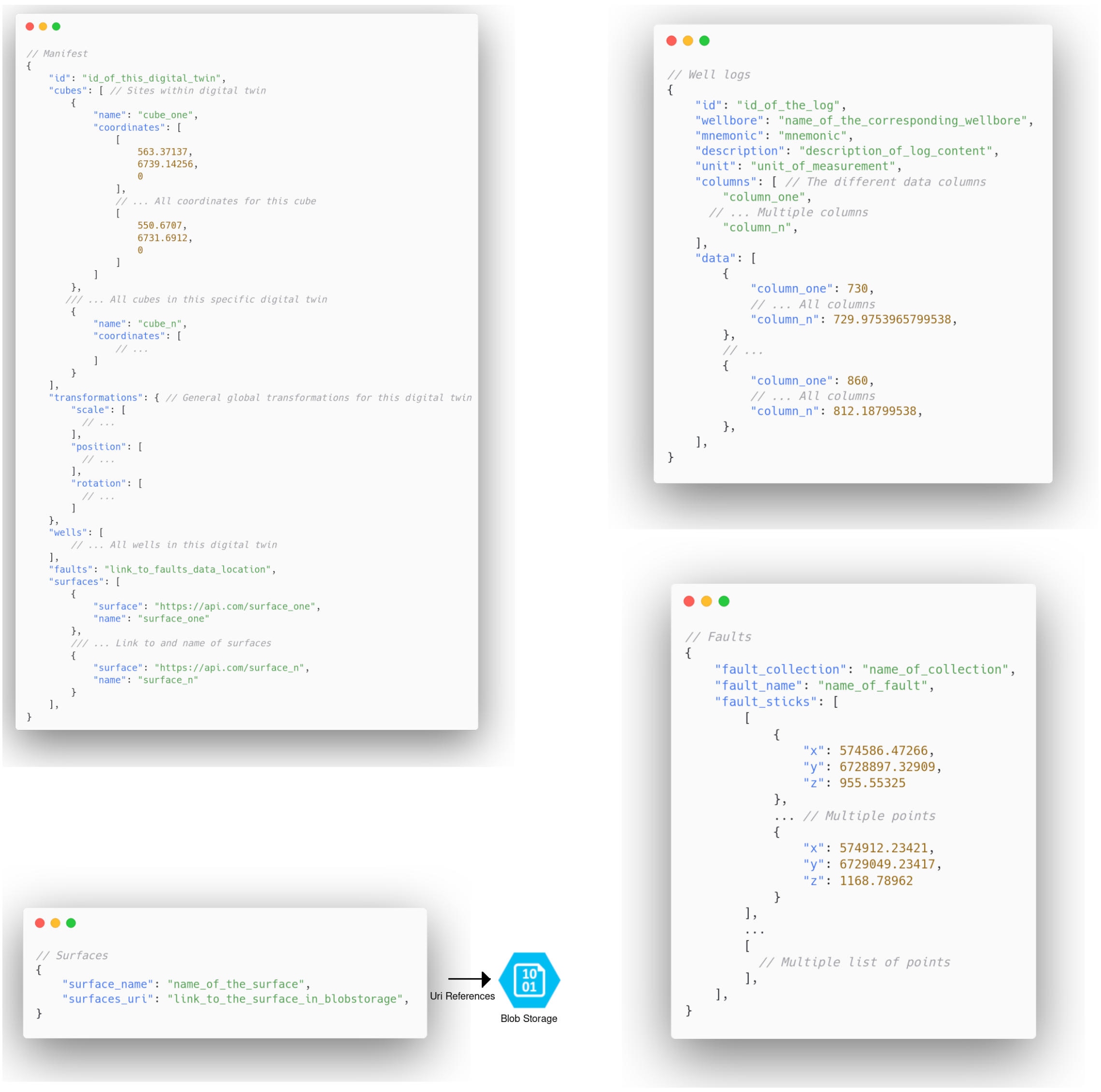
We had several different requirements regarding our choice of persistent storage. The first big question is whether or not a SQL or NoSQL based database should be utilized. We approached this questions by looking at the data we were going to store. Generally speaking, a SQL database style is a great choice when the data is relational. Relational data means that there exists some semi "flat" hierarchy between the data[16]. The data with have to work with in the Smeaheia dataset does not have much correlation between each other, an example of this is that the well logs don't directly relate to the faults in any sensible way. The biggest advantage of using a SQL database is its querying capabilities. This way of querying data is great when data is not initially coupled. If we have a service which receives 100 new transactions every second, it would make sense to store it into a transaction table. We could then easily query this data later, locating all transactions matching a user with a specific ID. In our case, all the data is precoupled. As an example, a fault file contains all the data necessary for making sense of a complete fault. No new data regarding a previous fault will be pushed to the database, we thus know all the information we need to know at the time of parsing/processing. Based on this, we decided that an NoSQL database would be the best choice for our project.

There exists four different types of NoSQL databases: Key-Value stores, Column-oriented databases, Graph databases and document databases. When deciding on which of these types were the most sensible for us, we took a look at our data once again. Our data is logically coupled together. For instance, all data regarding a fault collection is contained within a fault file. The first alternative we considered was a key-value store. A key-value store is the simplest form of NoSQL database, linking a key directly to a value. Our data is a bit more complex than this, and a key-value store database was therefore not a viable option. A column-oriented database focuses primarily on aggregating all the values in a given column together. The main strength of a column-oriented database is for analytical purposes, such as determining the total number of sales that have been completed during the last month. We do not need this analytical power in our project, and a column-oriented approach was therefore disregarded. Next up is a graph database. The main strength of a graph database is the relationship between nodes. The real power of graph databases comes into play when the joining of data is the primary focus. As already established, all of our data is "self-contained" meaning the faults are not particularly interested in the values of the well logs, as an example. The final type was the Document databases. A document database is theoretically a perfect match for our use case. The main benefit of document databases are the fact that objects can be retrieved in a form that closely resembles the way it is going to be utilized in the application itself. What this means is that the documents contains all the data required to utilize it directly. The problem with the data we have to work with in the dataset is that the data is not directly usable, we have to parse and process it into a data format that makes sense in a document database[30].

The next thing we had to do was to decide on a data format for our data. The logical way to approach this problem was to look at the different document database vendors, to determine what data format their databases utilizes. Since the client utilizes Azure as their cloud vendor, this was a natural place to start. Azure has a wide variety of different database solutions, one of which is CosmosDB[21]. CosmosDB provides several features that suits our project in a great way. CosmosDB scales in a automatic and limitless manner. The theoretical maximum size of CosmosDB is only limited by the budget of your service. Azure provides a serverless CosmosDB alternative, meaning that you do not have to have servers running around the clock when you don't have traffic. And finally, the most appealing feature: You can select from a wide variety of 'backends'. It is possible to use backends such as MongoDB, SQL, or Cassandra. We have a pretty solid foundation in the use of SQL from the course IDATG2204 - Data modeling and database systems. By utilizing CosmosDB, we get a hybrid solution that gives us the best of both worlds: Self contained documents with data available through a SQL-like querying syntax.

We needed to transform the data into a format available for storage in CosmosDB. CosmosDB Documents are of the JSON data format. JSON was a nice alternative in regard to data such as Faults and Well logs. Since we went for the web oriented approach of ThreeJS and React, the most sensible format for the surfaces was GlTF files. We solved this concern through the use of Blob Storage in combination with CosmosDB documents. We created documents with references to blobs in the blob storage, allowing the surfaces to be fetched with a Blob Client using the reference located in CosmosDB. The way this works is that the database is queried for surfaces with a given name, the surfaces matching this name returns the reference to the actual file in blob storage, which then can be fetched.

The different documents looked like this:



```
// Manifest
{
    "id": "id_of_this_digital_twin",
    "cubes": [ // Sites within digital twin
        {
            "name": "cube_one",
            "coordinates": [
                [
                    563.37137,
                    6739.14256,
                    0
                ],
                // ... All coordinates for this cube
                [
                    550.6707,
                    6731.6912,
                    0
                ]
            ]
        },
       /// ... All cubes in this specific digital twin
        {
            "name": "cube_n",
            "coordinates": [
                // ...
            ]
        }
    ],
    "transformations": { // General global transformations for this digital twin
        "scale": [
          // ...
        ],
        "position": [
          // ...
        ],
        "rotation": [
          // ...
        ]
    },
    "wells": [
        // ... All wells in this digital twin
    ],
    "faults": "link_to_faults_data_location",
    "surfaces": [
        {
            "surface": "https://api.com/surface_one",
            "name": "surface_one"
        },
        /// ... Link to and name of surfaces
        {
            "surface": "https://api.com/surface_n",
            "name": "surface_n"
        }
    ],
}
```

```
// Well logs
{
    "id": "id_of_the_log",
    "wellbore": "name_of_the_corresponding_wellbore",
    "mnemonic": "mnemonic",
    "description": "description_of_log_content",
    "unit": "unit_of_measurement",
    "columns": [ // The different data columns
        "column_one",
      // ... Multiple columns
        "column_n",
    ],
    "data": [
        {
            "column_one": 730,
            // ... All columns
            "column_n": 729.9753965799538,
        },
        // ...
        {
            "column_one": 860,
            // ... All columns
            "column_n": 812.18799538,
        },
    ],
}
```

```
// Faults
{
    "fault_collection": "name_of_collection",
    "fault_name": "name_of_fault",
    "fault_sticks": [
        [
            {
                "x": 574586.47266,
                "y": 6728897.32909,
                "z": 955.55325
            },
            ... // Multiple points
            {
                "x": 574912.23421,
                "y": 6729049.23417,
                "z": 1168.78962
            }
        ],
        ...
        [
          // Multiple list of points
        ],
    ],
}
```

```
// Surfaces
{
    "surface_name": "name_of_the_surface",
    "surfaces_uri": "link_to_the_surface_in_blobstorage",
}
```

Figure 14: Overview of persistent storage documents

5.6 Surfaces

In the Smeaheia dataset there were several surface files. A surface file consists of a point cloud that represents a surface in 3D space. A point cloud is a set of data points in space, which we can view as a collection of data points defined by a given coordinate system. In our case, the coordinate system is in 3D space and defined the shape of the surfaces. With these point clouds, we could create 3D meshes of the surfaces and visualize them in the scene.[45]

The process of choosing the technology for the implementation of the surfaces was a combination of looking at the documented SDK's for Azure Function Apps, and what libraries could meet our requirements of generating geometry from a point cloud. While researching our possibilities in Azure, we considered the

SDK's that were available in the documentation[23].

One of the libraries we tested for generation of geometry from point clouds was Open3D, which is a library that has a Python API. Below we will go into some of the technical design choices we made for implementing the surfaces of the project, focusing on the data processing and libraries that were chosen.

Adding a surface file in blob storage will trigger the parser code, which will process and generate a geometry file. When the Parser Code is finished, the generated geometry file will be stored back in blob storage, and a path to this file along with its name, will be stored in CosmosDB. The REST API can then serve the geometry file from blob storage, and the name from CosmosDB, for the Web app to fetch for visualization. Our final architecture for the surfaces:

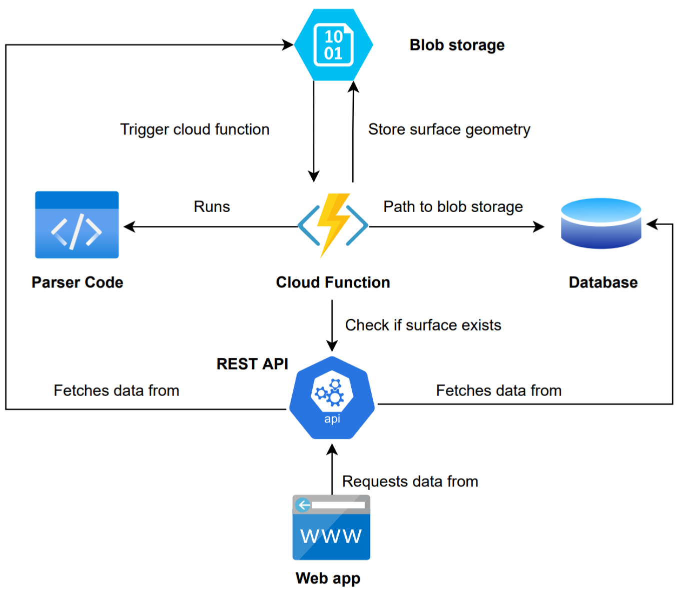


Figure 15: Overview of Surfaces service architecture

Data Processing

Processing the surface data meant that we had to organize the point cloud in such a way that Open3D could read it and create a point cloud object from it. This point cloud object would later generate a geometry file with one of Open3D's algorithms. This gave us some flexibility, as writing to a file can be implemented in several ways. However, in our case we had to take the Azure Functions into consideration. This meant that we had to try and not write the point cloud data to a file, but rather pass it to Open3D as an object. This is because Azure Functions are on-demand and serverless, which means that writing a file to disk is not ideal as the service does not provide any persistent storage[1].

Based on the knowledge we gained on how Open3D has to receive point cloud data, we decided to utilize the Python SDK for Azure's services.

Open3D

Open3D is an open-source library that supports rapid development of software that deals with 3D data. The

Open3D frontend exposes a set of carefully selected data structures and algorithms in both C++ and Python. The backend is highly optimized and is set up for parallelization[38]. In our case, we used the Python API for Open3D. This is further explained in the Implementation section of surfaces.

Pygltflib

While working with Open3D, a limitation was uncovered that would cause us difficulties in the web-app. Open3D would allow us to generate a GlTF file, but modifying it once created was not possible. One of the requirements we discovered from the web-app was that the GlTF file's geometry node had to be named in a specific way in order for them to be imported into the scene. Open3D did not allow us to accomplish this, so we needed a separate library for this purpose.

Pyglflib is a library for reading, writing, and handling GlTF files[41]. This library would allow us to name the geometry in the GlTF file for the web-app to read. This is further explained in the Implementation section of surfaces.

5.7 Well logs

The requirements specified that we needed to visualize the three wellbores of the Smeaheia area, as well as their well-logs. By wellbore we mean the borehole that has been drilled from the sea surface, to the aquifer that represents the potential carbon storage site. We are especially interested in the curvature and inner-surface properties of this wellbore, as well as other properties of the wellbore. All of this information that we would like to know about the wellbores have been recorded into well-logs of various formats, that we will need to parse and process, in order to eventually visualize.

When it came to choosing the technologies for implementing this parsing and processing, we had a few constraints. One of our constraints were that the implementation language had to be supported by Azure Functions and Azure SDKs. We wanted libraries capable of parsing the well-logs, as these come in quite archaic formats that would require a significant time investment to be parsed properly with a custom built solution.

Well-log parsing

Most of the options for parsing libraries were either immature or unmaintained, or lacked support for the particular formats we needed. However, based on an article provided by Glex[54], and our own research, we found two viable alternatives *Log I/O* and *lasio*. Log I/O is an enterprise grade parse for a multitude of well-log formats, it did however come with a price tag that quickly disqualified it for us. On the other hand, lasio is an open-source parser, written in Python, supporting the specific well-log format that we need. Since we had already decided to use Python for some of the other microservices at this point, we decided to use a Python based stack for parsing and processing well-logs as well.

Well-log processing

Additionally, we needed functionality for parsing Excel files and for processing tabular data in general. Python offers some excellent libraries in the form of *Pandas* and *NumPy* for these purposes. We have worked with these libraries in the PROG2051 - Artificial Intelligence course, and so they were the obvious choice. *Pandas* is a data analysis and manipulation library, which is well suited for working with tabular data[3]. *NumPy* is a fundamental library for a broad range of mathematical and scientific calculations in Python[2]. *Pandas* and *NumPy* work well together, as values produced by one library is accepted as input into the other.

Wellbore geometry generation

We tried to locate libraries that implements some of the calculations and geometry generation we required for this service, but we were unsuccessful in this regard. Therefore, the calculations and generation had to be implemented manually, with the help of Pygltflib for exporting the geometry, and Pillow for exporting the texture maps.

Service Architecture

The service, running as an Azure Function, waits for inputs from blob storage. When input arrives, the function runs, processing the well-logs, and finally outputting the results back to blob storage and CosmosDB, making them available through the REST API.

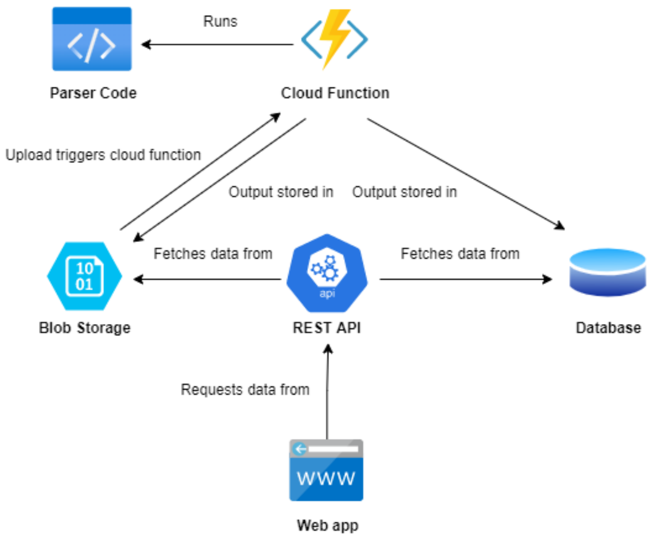


Figure 16: Overview of Well logs service architecture

5.8 Faults

When deciding on the technologies for parsing the faults, there were a few requirements. The most essential ones were that the language could run in Azure cloud functions, and have bindings for Azure services such as CosmosDB and Blob Storage. When parsing data, there often exist libraries for parsing the format of which the data is in. This was not the case with faults. The data format was of a non-standardized, unknown format. Preexisting libraries were thus not of any concern when considering the language for parsing/processing the faults.

Python provides SDKs for Azure's services, and runs in Azure Functions. The only drawback to Python is the relatively weak type system, making test writing a bit messy compared to languages such as Rust or Haskell. Considering the trade-off with the great support for Python in Azure against a bit messy tests, we decided that Python was the best choice for the faults parser.

The faults parser is written in Python, runs in an Azure Function, outputs its resulting data into CosmosDB, which in turn is queried by the REST API, and finally the geometry is generated in the web-app. The architecture for the faults service looks like this:

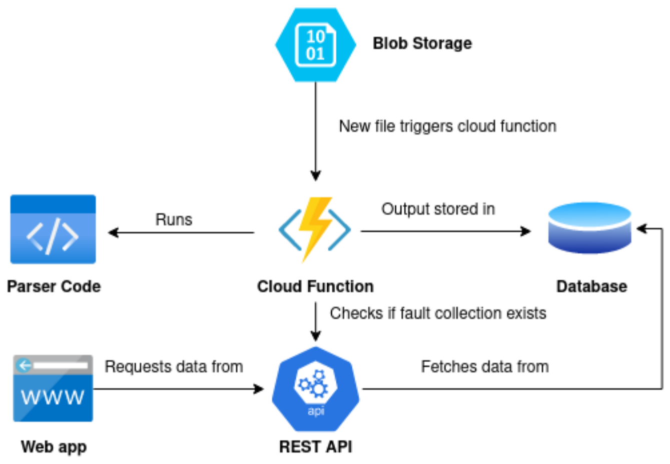


Figure 17: Overview of Faults service architecture

5.9 Horizons

The horizons in the dataset were not a part of the final product. We developed the parser and geometry generator for the horizons, but not the function app or visualization in the web-app. The horizon files in the Smeaheia dataset are shape files, consisting of multiple files in multiple formats. A shapefile is a simple, non-topological format for storing the geometric location and attribute information of geographic features. Geographic features in a shapefile can be represented by points, lines, or polygons[56]. The formats we needed to consider for this dataset, were .shp, shx, .dbf and .prj.

We had to extract the point cloud from the .shp and .shx files. Neither the .dbf or .prj contained information we needed for this visualization. Choosing a technology to accomplish this job boiled down to finding a library that could extract the point cloud from the shape files. Through our research, we tested *GeoPandas*. *GeoPandas* is an open source project to make working with geospatial data in Python intuitive. *GeoPandas* extends the data types used by pandas to allow spatial operations on geometric types[60]. We will go further into how the point cloud was extracted in the implementation part of the horizons.

After extracting the point cloud, we needed a tool for generating geometry. Here, we needed to either triangulate the geometry, as we did with the surfaces, or create point geometry. Having had some experience with Open3D, we tested this library first to check if it could meet our requirements. We already knew Open3D could triangulate a point cloud, but we found that the library was also able to generate point geometry in situations where the points in the point cloud were too far apart to triangulate properly.

5.10 Security

There are usually three main aspects to security in the cyberspace. Confidentiality, integrity, and availability. Confidentiality for us is mainly in regard to how we secure our secrets, since all the other data in the system is public. Integrity would be making sure the data we serve and visualize is accurate, and that it has not been manipulated, neither in transit nor at rest. And availability would be in regard to ensuring our solution is available to the users when then need it. In this section, we will explain how we secure these aspects of our solution.

By secrets, we mean different account credentials, connection strings, publishing profiles, and such that are used for authorizing and authenticating between different services in the system. All of these need to be kept confidential in order to prevent unauthorized access to our resources. Different services and environments deal with secrets differently, and so we had to be careful when handling these.

For our local development environments, we chose *dotenv* files as an intuitive and effective way of dealing with secrets. *Dotenv* files are plaintext files stored on individual developers machines that list individual secrets in key-value pairs. These files are usually sourced into the developers shells and made available to local processes via shell environment variables. This is by no means a perfect solution to managing secrets for developers, as these environment variables can be accessed by *any* process run by the shell after sourcing the *dotenv* file. Knowing this, we chose to adapt our secret consumers to source these *dotenv* files themselves, bypassing the need for developers to source them into their shell environments. There is still a risk posed by these files being plaintext files on disk, but we deemed this to be an acceptable risk, as theft of physical hardware or physical break-ins, necessary to compromise disk confidentiality, seemed unlikely in our situation.

Azure already makes several promises about availability. Based on the fact that none of the requirements state that we must achieve extra high availability, because downtime of this solution is not a business halting problem, we deemed Azures promises about availability good enough for this solution. Azure also provides several extra options for various availability related services, like DDOS protection[22], which can easily be incorporated into the solution if deemed necessary in the future.

6 Testing

In this section, we will talk about our approach to writing tests using Test Driven Development, and how we tested the various aspects of the solution.

6.1 Test Driven Development

One of our learning goals for this project was to learn more about test driven development. Test Driven Development is a software development practice that focuses on creating unit test cases before developing the actual code[62]. Therefore, we had a focus on writing unit tests throughout the project, completing the red/green/refactor cycle over and over. We felt this was an important measure in order to manage the complexity of this project. There are many parts that all have to interact in complex ways. Making sure that they all individually work as specified, helped us be more productive in our work and more confident in our solution as a whole.

6.2 Automated Testing

One of our goals for this project was to learn about test driven development. And in order to utilize TDD properly, we had to set up automated test suites for all our tests. By an automated test suite, we mean that we want to be able to run all the tests associated with a single project component, independent of all other components, and with all the configuration and set up parts automated. Meaning we should be able to run a single shell command, either locally or as part of the continuous integration, and be able to know if a component is working as intended or not.

The first step to setting up automated testing is to pick a test runner or a testing framework. Since our project uses multiple programming languages and frameworks, we needed to choose one testing framework for each programming language we used.

For the components written in Python, we chose to use PyTest as our testing framework. Alternatives here would have been Pythons built-in testing framework called *unittest*, or another test runner called *tox*. This was mostly a question of what level of abstraction we wanted to work at. *unittest* is a low level, rudimentary testing framework. It lacks many of the convenient features of PyTest, such as PyTests simple and reusable test fixtures and, PyTests improved output and debugability. *tox*[51] on the other hand attempts to solve a different problem. *tox* describes itself as a *development task automation tool* and the main selling point is its ability to run tasks targeting multiple Python implementations. This is not really relevant for us, since we know exactly what version and implementation of Python we are going to use ahead of time. PyTest also integrates nicely with *Flask*, which will be relevant when we describe the implementation of the REST API.

Setting up PyTest is as intuitive as writing a few tests. As long as they are in the 'tests' directory, and have the 'test_' prefix in the file name, PyTest will automatically discover the test and run them when invoked. See listing 1 for an example of what these test looked like. Additionally, PyTest has powerful support for fixtures, allowing us to create and reuse boilerplate setup and teardown code. See listing 2 for an example of how these fixtures are defined in code.

```python
def test_get_smeaheia(client: FlaskClient):
    """Test getting the manifest for the smeaheia area."""
    res = client.get("/manifests/smeaheia")

    assert res.is_json

    json = res.get_json()
    assert isinstance(json, dict)

    assert json["id"] == "smeaheia"
    assert len(json["wells"]) == 3
```

Listing 1: PyTest test example from REST API

```python
@pytest.fixture()
def app() -> Iterable[Flask]:
    app = create_app()
    app.testing = True
    yield app

@pytest.fixture()
def client(app: Flask) -> FlaskClient:
    return app.test_client()
```

Listing 2: PyTest fixtures from REST API

```
------- coverage: platform linux, python 3.9.12-final-0 -------
Name                             Stmts   Miss  Cover
--------------------------------
app/__init__.py                      7      0   100%
app/__main__.py                      3      3     0%
app/common.py                       27      4    85%
app/factory.py                      14      0   100%
app/faults.py                       22      3    86%
app/manifest.py                     24      0   100%
app/realtime.py                     40     18    55%
app/surfaces.py                     31      1    97%
app/well_blob_queries.py            30      6    80%
app/well_cosmos_queries.py          22      0   100%
app/wells.py                        46      7    85%
conftest.py                         16      1    94%
tests/test_faults.py                24      0   100%
tests/test_manifests.py             17      0   100%
tests/test_realtime.py               9      0   100%
tests/test_surfaces.py              28      0   100%
tests/test_wells.py                 37      0   100%
--------------------------------
TOTAL                              397     43    89%
```

Listing 3: REST API Test coverage report

```
------- coverage: platform linux, python 3.9.12-final-0 -------
Name                                 Stmts   Miss  Cover
-----------------------------------
composite-function/__init__.py          37     37     0%
excel-function/__init__.py              19     19     0%
tests/__init__.py                        0      0   100%
tests/test_composite.py                 14      0   100%
tests/test_excel.py                      3      0   100%
tests/test_trajectory.py                31      0   100%
tests/test_well_logs.py                  3      0   100%
well_logs/__init__.py                    1      0   100%
well_logs/composite.py                  19      0   100%
well_logs/excel.py                      52      0   100%
well_logs/gltf.py                       20      6    70%
well_logs/header.py                     16      0   100%
well_logs/log.py                        46      1    98%
well_logs/trajectory.py                102      9    91%
-----------------------------------
TOTAL                                  363     72    80%
```

Listing 4: well-logs test coverage report

```
------- coverage: platform linux, python 3.10.4-final-0 -------
Name                       Stmts   Miss  Cover
-------------------------------
dependencies/__init__.py       1      0   100%
dependencies/parser.py        46     10    78%
dependencies/types.py         43      5    88%
tests/__init__.py              0      0   100%
tests/test_parser.py          24      0   100%
tests/test_types.py           29      0   100%
-------------------------------
TOTAL                        143     15    90%
```

Listing 5: Faults test coverage report

```
------- coverage: platform win32, python 3.9.10-final-0 -------
Name                            Stmts   Miss  Cover
----------------------------------
common\__init__.py                  5      0   100%
common\color_generator.py          46      2    96%
common\data_processing.py          36      1    97%
common\geo_generate.py             65     12    82%
common\py_gltf.py                   5      3    40%
tests\__init__.py                   0      0   100%
tests\test_color_generator.py      72      0   100%
tests\test_data_processing.py      44      0   100%
tests\test_geo_generate.py         36      0   100%
tests\test_py_gltf.py               0      0   100%
----------------------------------
TOTAL                             309     18    94%
```

Listing 6: Surfaces test coverage report

```
------- coverage: platform win32, python 3.9.10-final-0 -------
Name                                 Stmts   Miss  Cover
-------------------------------------
color\__init__.py                        2      0   100%
color\color_generator.py                46      2    96%
geometry\__init__.py                     2      0   100%
geometry\gen_geometry.py                14      4    71%
processing\__init__.py                   2      0   100%
processing\data_processing.py           36     11    69%
shapefiles\__init__.py                   2      0   100%
shapefiles\shapefile_processing.py      19      3    84%
tests\__init__.py                        0      0   100%
tests\test_color_generator.py           72      0   100%
tests\test_data_processing.py           25      0   100%
tests\test_geo_generate.py              21      0   100%
tests\test_shapefiles.py                18      0   100%
-------------------------------------
TOTAL                                  259     20    92%
```

Listing 7: Horizons test coverage report

The web app is written in JavaScript, and therefore requires a different testing framework. There is a plethora of options when it comes to testing frameworks and libraries for JavaScript, making it difficult to choose without preexisting experience. We therefore based our choice on React's official recommendation[48] of using Jest as our testing framework, which allowed us to use React's built-in snapshot testing, and to perform DOM testing via *React Testing Library*.

Since we are using *create-react-app*, testing with Jest and all the configuration that comes with it, is already set up and done for us. We add tests by creating '*.test.js' files in the 'src' directory and define the tests using Jests testing primitives, the methods used for building tests. This allows us to write unit tests for our JavaScript code. However, if we want to write unit tests for our React components, we have the option of using React's built-in snapshot testing, and/or DOM testing via React Testing Library.

Snapshot testing is an approach to testing component driven web apps, and React apps in particular, where we test if rendering a component produces the same DOM tree, and attributes as it did last time. The info, stored as part of the tests, that defines how a component should render, is called a snapshot. The advantage of this approach is that it is intuitive to implement for React apps. The disadvantage is that it does not fully test all aspects of a component, only how it renders, and not aspects such as what side effects it generates.

DOM testing is a testing approach where the components are rendered into a virtual DOM tree. And tests are run against this virtual DOM. This gives us some advantages over snapshot testing, like the ability to run precise queries against the tree, but not needlessly fail if insignificant differences occur as a result of implementation details. In addition, we can also test what side effect the component generates as it is mounted, updated, and unmounted. However, we still can't test what the component looks like and other details related to styling.

There are many other alternatives to testing web apps and websites in general, like Selenium. These testing approaches are not well suited for testing ThreeJS applications, and were disregarded for this project.

6.3 Manual Testing

Since UIs are hard to fully test in an automated way, we also developed a strategy for manually testing the web app. Specifically, we had the following step-by-step procedure for testing the web app with good coverage.

Step	**Activity**	**Expected behavior**	**Status**
1.	Test camera controls	Camera can rotate, zoom, and pan	Working
2.	Test toggling menus	Inspector and toggle menu can be toggled	Working
3.	Test selecting wellbore	Clicking on a wellbore selects it in the inspector	Working
4.	Test well-logs	Well-logs render correctly and can be collapsed	Working
5.	Test switching wellbore texture	Chronostratigraphy, lithological formations and groups can all be projected onto the wellbore	Working
6.	Test toggling fault visibility	Faults can be shown and hidden	Working
7.	Test toggling surface visibility	Surfaces can be shown and hidden	Working

Table 2: Step-by-step procedure for testing wep app

6.4 User Testing

We conducted informal user testing as part of our meetings with Glex at the end of each sprint. We roughly based the testing procedure on the manual testing procedure above. The point of these tests was mainly to figure out if the functionality as implemented satisfied the specified requirements, and to gather any feedback Glex had for us. Based on this feedback, we revised the requirement specification. Clarifying some points, added some, and removing others that became irrelevant or out of scope. Therefore, one of the most important things the feedback was utilized for, was to verify that the work being performed was correct in terms of their expectations, and what parts of the specification were the most important for Glex.

Listing 3 shows a sample of the feedback given to us in one of these sessions

Problem	**Solution**	**Severity**	**Added to back-log**	**Status**
Well-logs are upside down	Flip the x-axis for the graph	High	Yes	Fixed
Some of the wellbores and well-logs are missing	Add the remaining of the wellbores and well-logs	High	Yes	Fixed
Wellbores should be labeled	Add floating label for wellbores in scene	High	Yes	Fixed
The faults disappear when the toggle menu is closed	Faults should stay visible	High	Yes	Fixed
We want to be able to toggle surfaces	Overhaul and extend toggle menu to include surfaces	Medium	Yes	Fixed
Seabed surface is missing	It isn't	High	Yes	Fixed
Some of the surfaces are duplicated	Remove duplicates	High	Yes	Fixed
The web-app performance is poor on low-end hardware	Look into optimizing the performance of the web-app	High	Yes	Improved
Camera controls are hard to use	Look at other alternatives for camera controls	Medium	No	In progress
Maybe add templates for the wells on the seabed	Maybe	Low	No	In progress

Table 3: Sample of feedback from Glex

7 Implementation

In this chapter, we will go further into the implementation of the different aspects of the project. We have divided the most central parts into sections, with each having several subsections within themselves.

7.1 Frontend

As explained in the technical design chapter, we decided to utilize React, ThreeJS, and React Three Fiber for the web app. The web app itself is structured into several different React components, some of which contains ThreeJS code, represented through the React Three Fiber library.

The web app consists of a root containing five subcomponents: The app loader, the viewport, the inspector, the toggle menu and the status bar component. The app loader is responsible for fetching all of the data of the service, and stores it into state. This allows the subcomponents to access the global state, ensuring that the entire web app operates on the same data.

7.1.1 Data loading

The data is loaded through the custom useData() hook. The first thing that happens is that the manifest is fetched. The manifest contains all the specific information related to a given digital twin area/site. If the manifest fails to load, the loading of the web app is aborted.

```
export function useData() {
const [data, setData] = useState(undefined)
const [done, setDone] = useState(false)

const f = useCallback(async () => {
    const area = "smeaheia"
    const manifest = await fetch(
        `https://dt-api.azurewebsites.net/manifests/${area}`
    )
        .then((res) => res.json())
        .catch(() => console.error("Failed to load manifest!"))

```

Listing 8: Wells are loaded using the manifest

After the manifest has been loaded, all the other sources of data is fetched. The manifest is responsible of defining where the data can be fetched from. An example of how this fetching looks, with wells in this particular case:

```
    const wells = manifest.wells.map(async (well) => {
        let ret = {
            logs: {},
            maps: {},
            realtime: 0.1,
        }

        ret.header = await fetch(
            `https://dt-api.azurewebsites.net/wells/${well}`
        )
            .then((res) => res.json())
            .catch(() => console.error("Failed to get well header!"))

```

Listing 9: The wells is loaded

7.1.2 The Viewport

The viewport contains the 3d world of which the digital twin resides. All ThreeJS code is written with a syntax similar to React. The code for the respective parts of the 3d world is covered in each subsection further down in the implementation chapter.

7.2 REST API

This section will cover the implementation of the REST API. The REST API serves mostly as a gateway to the database and the results of any processing the other microservices have done. As discussed in the technical design chapter, the REST API is implemented in Python using Flask as the main framework, in addition to utilizing Azure's Python SDK for accessing the database and blob storage.

7.2.1 Factory and Blueprints

Based on recommendations from Flask's own documentation[20], we decided to implement our Flask application using the factory pattern[5]. This means that there is one module dedicated to creating the Flask app, loading configuration, registering all the endpoints, and managing all global resources. This pattern is compatible with Python's WSGI specification[11], meaning that our application can be plugged into a variety of web servers, both development servers and production servers.

In order to make our REST API as modular and cohesive as possible, we followed Flask's advice on splitting related endpoints and functionality into their own modules, and making use of Blueprints[29] in order to mount these standalone modules into the Flask app. Listing 10 shows an example of the manifest module, and how it defines its own endpoints and implements the specific functionality it needs to function. This module can then be mounted into the rest of the API without regard for the other modules and implementation details.

```python
manifests_blueprint = Blueprint("manifest", __name__)

@manifests_blueprint.get("/manifests/<string:cube>")
def one_manifests(cube: str) -> Union[dict, tuple[str, int]]:
    """Return one manifest from cosmos db."""
    manifest = get_manifest(cube)

    if manifest is None:
        return "Manifest not found", 404

    return manifest

def get_manifest(cube: str) -> Union[None, dict]:
    """Return one manifest, by id, from cosmos db."""
    documents = well_logs_container.query_items(
        query="""
            SELECT c.id, c.cubes, c.transformations, c.wells, c.faults, c.surfaces
            FROM c
            WHERE c.id = @id
        """,
        parameters=[{"name": "@id", "value": cube}],
    )

    return first(documents)
```

Listing 10: The manifests module

7.2.2 Development vs. Production

Because we chose to implement the REST API as a generic WSGI server, this gave us options and flexibility when it came to what development server and what production server we chose to use. For local development, we used Flask's own development server[8], which provides features such as hot-reloading of modules, dotenv autoloading, and better debugging support. This greatly sped up development time. On the other hand, our production environment is using a standard Azure Python Web App, which is reality is *Gunicorn*[15] behind the scenes. *Gunicorn* is a much more scalable and production ready web server compared to Flasks built-in one, and allows for such things as multiple concurrent connections, due to supporting running multiple workers, which for servers dealing with a lot of I/O, which our server does, is a big advantage.

7.2.3 Testing

Flask provides guidelines[47] for how to implement API tests, in addition to conventional unit tests, that we used to guide our implementation of testing for the REST API. Specifically, we implemented a set of fixtures, like the *FlaskClient*, which helped us define succinct and to the point tests without too much boilerplate code, and used the APIs Flask provides in order to test for status codes, Content-Type, the schema of the JSON response, and in order to run queries against the response body. And with a test coverage of 90%, this gave us confidence that the API worked the way we intended it to.

```python
def test_get_smeaheia(client: FlaskClient):
    """Test getting the manifest for the smeaheia area."""
    res = client.get("/manifests/smeaheia")

    assert res.is_json

    # Check json format
    json = res.get_json()
    assert isinstance(json, dict)

    # Check some known info about manifest
    assert json["id"] == "smeaheia"
    assert len(json["wells"]) == 3
```

Listing 11: An API test showing how the manifest endpoint is tested

7.3 Surfaces

7.3.1 Surfaces Data Processing

As mentioned in the Technical Design chapter, we had to parse, process and generate geometry from the surface files. In this section we will cover how we analyzed the data, parsed, processed and made it ready for geometry generation.

Analyzing the Data

Our intention with analyzing the data was to get an overview over how it was structured. The structure was similar in all surface files, which allowed us to develop a parser that could handle the surface format in general. In 12 a snippet of a surface file is presented to illustrate the general structure.

```
# Type: scattered data
# Version: 6
# Description: No description
# Format: free
# Field: 1 x
# Field: 2 y
# Field: 3 z milliseconds
# Field: 4 column
# Field: 5 row
556399.694580 6716786.249595 -630.122375 123 2
556449.694580 6716786.249595 -630.604858 124 2
556499.694580 6716786.249595 -629.811157 125 2
556399.694580 6716836.249595 -629.808044 123 3
556449.694580 6716836.249595 -630.629639 124 3
556499.694580 6716836.249595 -630.128113 125 3
556549.694580 6716836.249595 -627.973328 126 3
556599.694580 6716836.249595 -626.480347 127 3
```

Listing 12: Parts of a surface file to illustrate the structure

When analyzing the data, we came to the conclusion that none of the comments, which are the lines that start with a hashtag, were needed. We also did not need the last two fields on each vertex because they represent the column and row of the vertex, as stated in the comments within the file.

Initial Tests

Our initial tests were done in Haskell. We decided to try this after doing some research on what technologies were good for data processing, and because we had previous experience with parsing using Haskell from the PROG2006 Advanced Programming course. Our initial idea was to parse the data to JSON format, making it storable in CosmosDB. We initially planned to generate the surface geometry in the web-app on-demand. In order to achieve this in Haskell, we had to first recognize the lines that started with a hashtag, and remove them. This was achieved through the removeComments function, which also used the checkComment function:

```haskell
checkComment :: Char -> [Char] -> Bool
checkComment a b = a `elem` b

removeComments :: Char -> [String] -> [String]
removeComments _ [] = []
removeComments c (x:xs)
    | checkComment c x = removeComments c xs
    | otherwise = x:xs
```

Listing 13: Two methods that work together in removing comments from surface files

After this, we could remove the last two fields of the vertex be running the init method twice, and later encode the x, y and z coordinates to JSON format.

After doing further analysis on the data and what we researched for the frontend, we decided that this would not be the optimal solution. It would be more beneficial to create a preprocessed GlTF file from the vertex data. Our reason for this came after reading the documentation of ThreeJS. It is stated there that the recommended format for import assets is GlTF, as explained in the frontend section of the Technical Design chapter. This would also ensure that no unnecessary data processing was performed. As mentioned in the surfaces section of the Technical Design chapter, we looked into *Open3D* and how the data would need to be organized in order to create a GlTF file from it.

The data format required to be inputted into *Open3D* had to be of the format illustrated in 14.

```
556399.694580 6716786.249595 -630.122375
556449.694580 6716786.249595 -630.604858
556499.694580 6716786.249595 -629.811157
556399.694580 6716836.249595 -629.808044
556449.694580 6716836.249595 -630.629639
556499.694580 6716836.249595 -630.128113
556549.694580 6716836.249595 -627.973328
556599.694580 6716836.249595 -626.480347
```

Listing 14: Point cloud data organizing for Open3D

Here, we can see the x, y and z coordinates of each vertex, which is what *Open3D* needs in order to create a point cloud object. There exists two different approaches for generating point cloud objects with *Open3D*. The first one is reading the necessary data directly from file. The other option is passing the data as a *NumPy* array. Since we ran the processing in a cloud function, we had to utilize the *NumPy* approach.

Point Cloud

Having the data processing running in a function app, meant that we had to read the data through an input stream. The input stream data type was converted into a list of strings after it was fetched from blob storage. As mentioned earlier, we needed to process the data into a point cloud for *Open3D* to be able to generate a GlTF file.

We needed to skip the lines that started with a hashtag. This filtering was performed while the reading of the input stream occurred.

```python
def read_from_inputstream(inputstream: list[str], symbol: str) -> list:
    """Reads data from file line by line

    Opens a file and reads each line by enumerating. Checks if a line has the
    # symbol and ignores it.

    Parameters
    -------
        inputstream : list[str]
                list of string from inputstream
        symbol : str
                the symbol that indicates a line to be ignored

    Returns
    -----
        list[float]
    """
    l_return = []

    for _, line in enumerate(inputstream):
        if symbol in line:
            continue
        else:
            l_return.append(line)

    return l_return
```

Listing 15: Read data from input stream

After reading each line, and skipping the comments, the method returns a list with all the vertex data. Next up we needed to remove the column and row values from each vertex, as well as reduce the value of each axis. More information on why we needed to reduce the value is covered in the *Open3D* section:

```python
def modify_vertex_data(l_input: list, r_num: int, reduce_amount: int) -> numpy.ndarray:
    """Modify the vertex data from file

    Takes in the list that is read from the data. Iterates through the list and splits
    up the elements into floats. Optionally, the remove_elements() method can be used
    to reduce the amount of data each vertice will have. And the reduce_amount variable
    can be used to reduce the size of the vertex position.

    Parameters
    ------
        l_input : list
                the list from reading data
        r_num : int
                the remove count on each vertex
        reduce_amount : int
                the amount to be reduced in vertex position

    Returns
    -----
    np.ndarray
        numpy array to be used for creating geometry
    """
    vertex_list = []

    for i, _ in enumerate(l_input):
        l_temp = [round(float(j) / reduce_amount, 9) for j in l_input[i].split()]

        if not l_temp:
            continue
        else:
           l_temp = remove_elements(l_temp, r_num)
           l_temp += c.color_generator(l_temp)
           vertex_list.append(l_temp)

    # convert the vertex list to a numpy array
    numpy_vertex_array = numpy.array(vertex_list, dtype='float')

    return numpy_vertex_array
```

Listing 16: Method to modify vertex data

In order to mitigate any potential inconsistencies in the point cloud, we decided to round the resulting value after the reduction of the vertex value had been performed. We chose the value "9" here to make sure that all of the original value were included after rounding.

We also added a remove_elements() method, to remove the column and row values from each vertex. This method runs the pop() method on the list a certain of times to remove unneeded values. And finally, the color_generator method is used to add a color to the vertex before the result is appended to the vertex list, which we will talk about in the next section.

Surface Color

Adding color to the surface meshes was important in order to indicate what depth level the surface had. We decided to implement the color palette from the Smeaheia website[43].

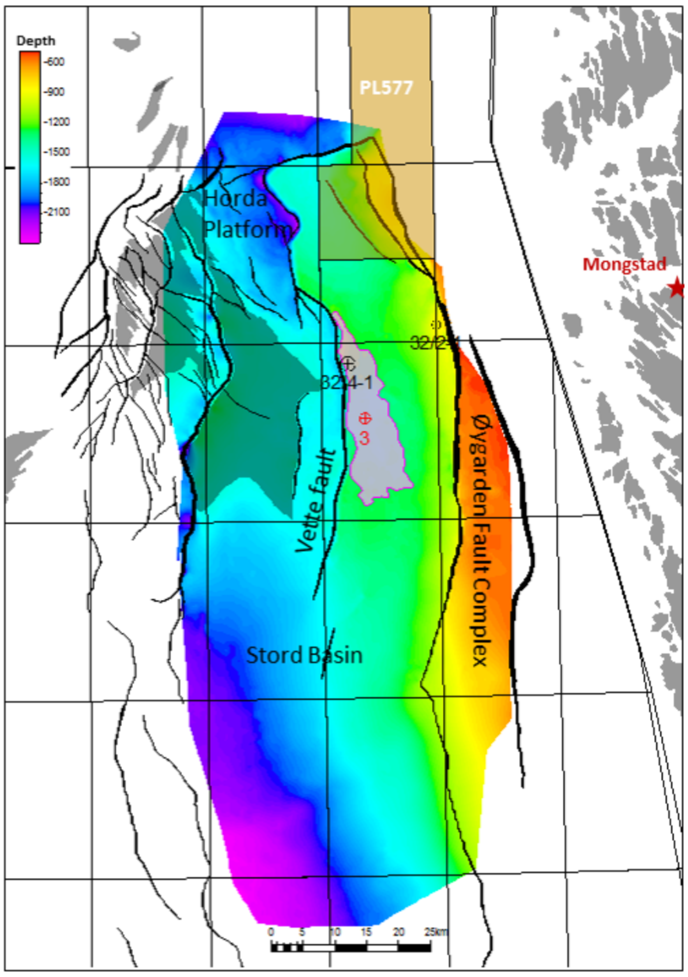


Figure 18: Shows the area around Smeaheia and the wells, indicating the depth of the sea bottom

The colors represent the depth level that a surface has. Our initial implementation had some problems, which will be explained further below. The method starts by extracting the z-axis from the vertex, and utilizes it to determine the color of the surface.

```
    z_value = z value from vertex in parameter

    # Depth level
    depth_level_0 = -300
    depth_level_1 = -600
    depth_level_2 = -900
    depth_level_3 = -1200
    depth_level_4 = -1500
    depth_level_5 = -1800
    depth_level_6 = -2100

    # Colors for the surfaces
    color_level_0 = [200,32,0]  # Red
    color_level_1 = [255,132,0] # Orange
    color_level_2 = [255,245,0] # Yellow
    color_level_3 = [66,255,0]  # Green
    color_level_4 = [0,255,208] # Teal
    color_level_5 = [0,130,255] # Blue
    color_level_6 = [201,0,255] # Purple
```

Listing 17: Depth level values and pre-defined colors.

After this, we checked the z value of the vertex against the depth level, and set a color for the vertex. We then calculate the color range between the depth levels, and utilized the add color method to interpolate between them.

```
if z_value > depth_level_1:
        input_color_range = (z_value - depth_level_0) / (depth_level_1 - depth_level_0)
        return add_color(color_level_0, color_level_1, input_color_range * -1)

elif z_value > depth_level_2:
        input_color_range = (z_value - depth_level_1) / (depth_level_2 - depth_level_1)
        return add_color(color_level_1, color_level_2, input_color_range * -1)
...
```

Listing 18: Parts of the depth level check to illustrate adding color

This value is then checked against each depth level. The add_color method takes in the start color, target color and what range between them is to be used. These values are later used to interpolate and return a list with the accurate color between two depth levels.

```
def add_color(start_color: list, target_color: list, color_percentage: float) -> list[float]:
    """Returns a color

    Lerps a color between the start and target color with a certain percentage.

    Parameters
    -------
    start_color : list
                the start color
    target_color : list
                the target color
    color_range : float
                the percentage between the start and target color

    Returns
    -----
    list[float]
        the calculated color between start and target
    """
    return_color = []

    for i, _ in enumerate(start_color):
        return_color.append(color_lerp(start_color[i], target_color[i], color_percentage))

    return return_color
```

Listing 19: Method to add color to vertex

And finally, the color_lerp method simply holds the formula for interpolating between two values

```
a + (b - a) * x
```

where a is the starting value, b is the target value and x is the value between a and b. This gave us the following result on a surface:

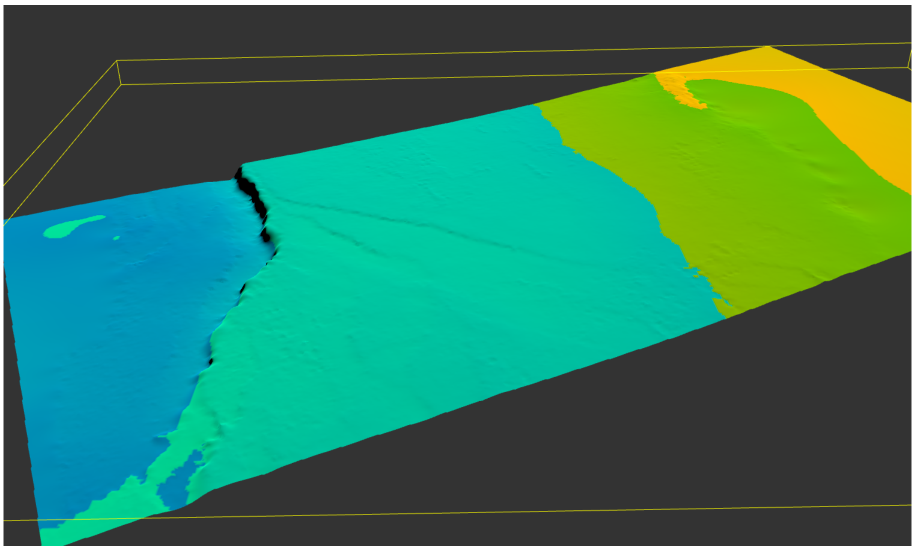


Figure 19: Surface with incorrect color interpolation

As can be seen here, there is no real interpolation between the depth levels. Another problem here is that the colors were too bright, which is not hard to spot since the light intensity is dialed down to 0.01. Both problems were handled by setting the RGB range to be between 0 and 1, and through the removal of the -1 that was multiplied into the color range. So the final color result looked like this:

```
# Colors for the surfaces
color_level_0 = [0.784, 0.125, 0.0]  # Red
color_level_1 = [1.0, 0.517, 0.0] # Orange
color_level_2 = [1.0, 0.960, 0.0] # Yellow
color_level_3 = [0.258, 1.0, 0.0]  # Green
color_level_4 = [0.0, 1.0, 0.815] # Teal
color_level_5 = [0.0, 0.509, 1.0] # Blue
color_level_6 = [0.788, 0.0, 1.0] # Purple
```

Listing 20: RGB values set in range between 0 and 1

This yielded a much better result, both in the interpolation and the brightness of the surface.

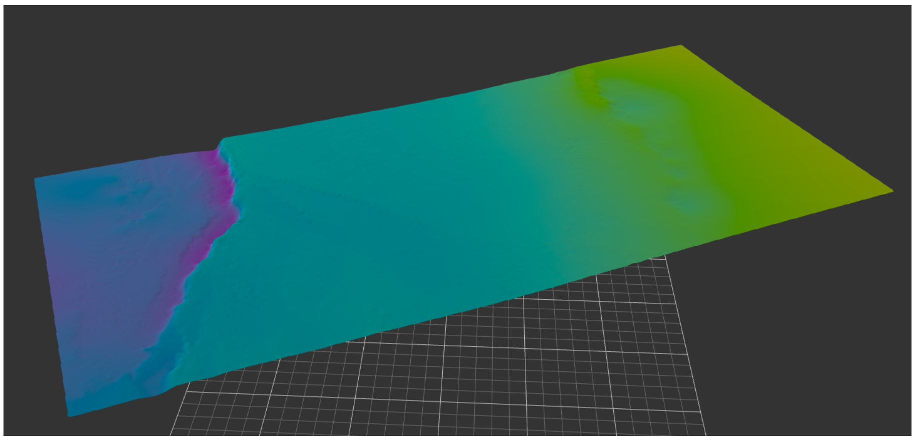


Figure 20: Surface with correct color interpolation

7.3.2 Surfaces Geometry Generation

While working on the data processing we researched how we could generate geometry from the point cloud. As mentioned in the data processing section, we discussed doing this on demand in the web-app. This generation process is relatively heavy, and it was thus not sensible to do this generation every single time the page loads. Because of this, we went with the approach of pre-generated GlTF files as stated in the Technical Design chapter.

In this section we will talk about how we used the geometry generation tools we presented in the Technical Design chapter. As well as what challenges we faced, how we tackled them and what the results were.

Open3d

Our purpose with using *Open3D* was to generate a GlTF file from a surface point cloud. The first thing we analyzed was the input data format of *Open3D*'s point cloud triangulation algorithms. We had to estimate the normals of the point cloud, this was an additional input requirement for the triangulation algorithm.

As mentioned in the data processing section, we had to organize the point cloud in a way that *Open3D* was able to process 14. *Open3D* reads point clouds through the use of *NumPy* arrays. We had to specify where the points are located within the *NumPy* array.

```python
def point_cloud_data(point_cloud_array: np.ndarray) -> open3d.cpu.pybind.geometry.PointCloud:
    """Read the point cloud data from file

    Reads the point cloud from the array that is sent in and assigns the position and color
    from the point_cloud_array. Calculates the distances between the points in the point cloud
    to determine the difference betweeen the minimum and maximum distance. The radius with which
    normals are calculated with are determined by the min_max_diff value.

    Parameters
    -------
        point_cloud_array : np.ndarray
                numpy array with vertex data

    Returns
    -----
        o3d.cpu.pybind.geometry.PointCloud
                the point cloud that will be used for generating geometry
    """
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(point_cloud_array[:,:3])
```

Listing 21: Create point cloud object

The next part that had to be performed was the estimation of normals.

```python
    pcd.estimate_normals(search_param=
            o3d.geometry.KDTreeSearchParamHybrid(radius=1.1, max_nn=30))
```

Listing 22: Estimate normals for point cloud

The estimate normals method iterates through all of the points and calculates the normals in relation to one another. The parameter radius and max_nn are the radius and the maximum amount of neighbors the method will take into consideration.

After processing the point cloud and estimating normals, we had to apply an algorithm that was able to triangulate the point cloud into geometry. The two alternatives we considered for this triangulation were the Poisson and Ball-Pivoting algorithms. There are advantages and disadvantages with both algorithms. Below we will outline how each of them work and what went into our final decision regarding which one to apply.

Poisson Algorithm

The Poisson algorithm considers all the points at once, without resorting to heuristic spatial partitioning or blending, and is therefore highly resilient to data noise. Unlike radial basis function schemes, the Poisson approach allows a hierarchy of locally supported basis functions, and therefore the solution reduces to a well conditioned sparse linear system. We describe a spatially adaptive multiscale algorithm whose time and space complexities are proportional to the size of the reconstructed model[40]. Because of the fact that the Poisson algorithm considers all the points at once, it proved to give fast results when it came to the triangulation. The method we used for this was:

```python
def triangle_mesh_poisson(pcd: PointCloud, depth: int, scale: float, linear_fit: bool) -> TriangleMesh:
    """Triangle the point cloud into a mesh with the poisson algorithm

    Generates the mesh with the poisson algorithm. Creates a bounding box after and crops the mesh.

    Parameters
    -------
        pcd : o3d.cpu.pybind.geometry.PointCloud
                the point cloud to be made into a mesh
        depth : int
                max depth of tree for surface reconstruction
        scale : float
                ratio between the diameter of the cube used for reconstruction
        linear_fit : bool
                if true, use linear interpolation to estimate position of iso-vertices

    Returns
    -----
        o3d.cpu.pybind.geometry.TriangleMesh
                the generated mesh
    """
    poisson_mesh = o3d.geometry.TriangleMesh.create_from_point_cloud_poisson(
                    point_cloud, depth=9, scale=1.1, linear_fit=True)[0]

    bbox = pcd.get_oriented_bounding_box()
    p_mesh_crop = poisson_mesh.crop(bbox)

    return p_mesh_crop
```

Listing 23: Generate mesh with Poisson Algorithm

We can see that the poisson algorithm method takes the point cloud as a parameter. The algorithm also expects the parameters depth, scale and linear fit. The scale describes the ratio between the diameter of the cube used to reconstruction and the diameter of the samples bounding box. The greater the scale, the less detail will be retained. If the scale is big enough the geometry will collapse onto itself, creating a watertight mesh. The depth parameter is used to defined how detailed the mesh will be, higher value equals more details. The linear fit allows the reconstruction to use linear interpolation to estimate the positions of iso-vertices, if set to true.

We also needed to crop the mesh. This was done by retrieving the bounding box of the point cloud with the "get_oriented_bounding_box" method, and then cropping the generated mesh with the bounding box. The result looked like this:

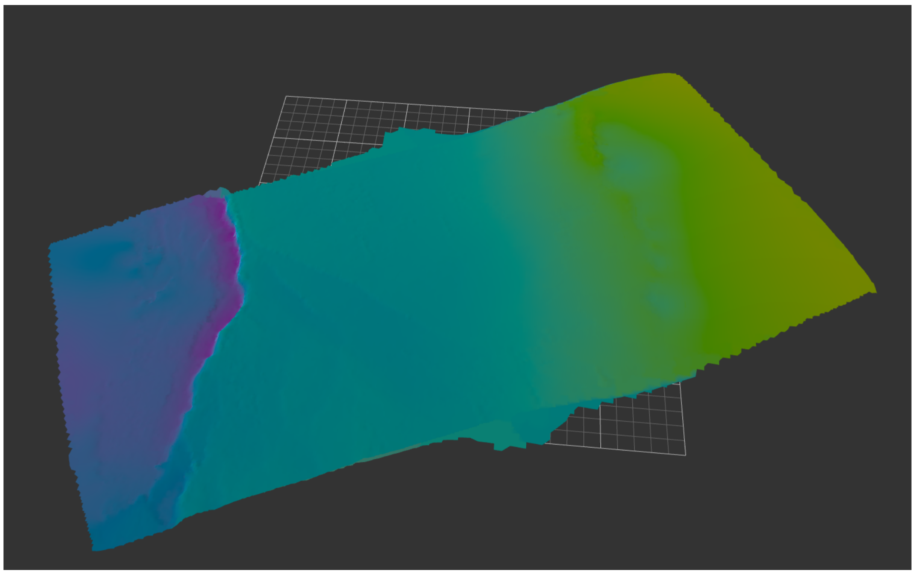


Figure 21: Poisson Surface Reconstruction

There are some downsides to the Poisson algorithm, mainly jagged edges and loss of detail, which is evident in 21. One of the upsides is that the triangulation is fast. Generating this mesh took 24 seconds. The Poisson algorithm was ruled out, in favor of the Ball-Pivoting algorithm, discussed in the next section. A detailed and consistent mesh is vital for a geologist to be able properly utilize a digital twin, which is a requirement the mesh generated through the Poisson algorithm did not satisfy.

Ball-Pivoting Algorithm

The Ball-Pivoting algorithm is named for the simulated use of a virtual ball to help reconstruct a mesh from a point cloud.

In the Ball-Pivoting algorithm, we can image a tiny ball rolling across the surface of points. This ball will have a radius which determines the distance the ball will take into consideration when rolling across the points. The radius should be slightly higher than the average space between points. When the ball has rolled on three points, it forms a triangle. From that location, the ball can roll into any direction[7].

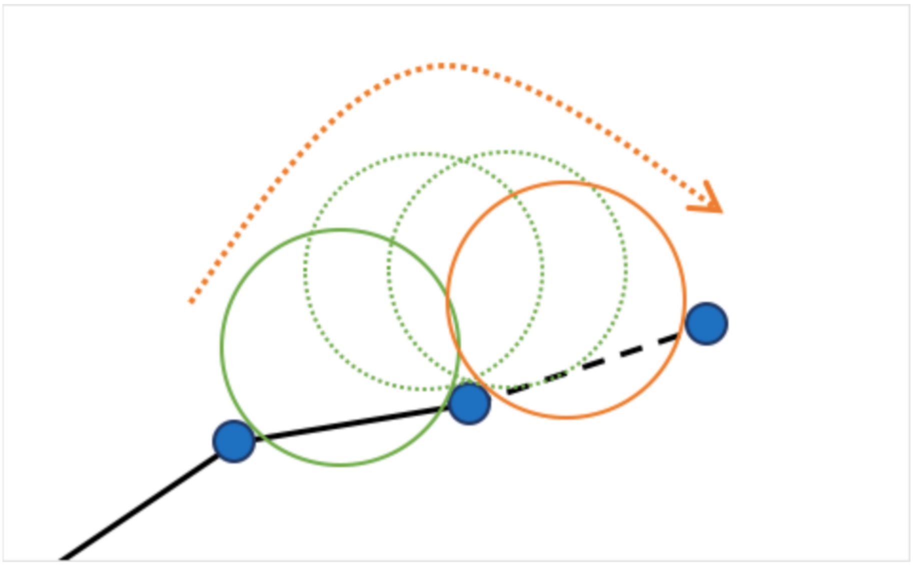


Figure 22: Illustration showing ball-pivoting algorithm

The ball will be pivoting along the triangle edge formed from two points. This will then settle the ball in a new location which will form another triangle, utilizing the point it just located alongside the two previous points. This process is repeated until the mesh is fully formed.

The Ball-Pivoting algorithm requires a radius which determines the distance to be taken into consideration by the ball while rolling across the surface. We calculated the radius with the following method:

```python
def calculate_radius(pcd: PointCloud) -> list:
    """Calculate the radius of the point cloud points

    Calculate the radius of each point in the point cloud. Used later on the open3d TriangleMesh method.

    Parameters
    -------
        pcd : o3d.cpu.pybind.geometry.PointCloud
                the point cloud to calculate point radius

    Returns
    -----
        list
    """
    distances = pcd.compute_nearest_neighbor_distance()
    avg_dist = np.mean(distances)
    radius = avg_dist * 2.0

    return radius
```

Listing 24: Method to calculate radius

We use the *Open3D* method to compute the nearest neighbor distances, and then *NumPy* to calculate the average. The average was multiplied with two, and the resulting value was set as the radius. The value used for the multiplication of the average would prove to cause some issues later, which we will explain in the Problem areas section. This radius is then used in the Ball-Pivoting algorithm to triangulate the surface point cloud.

```python
def triangle_mesh_bpa(pcd: PointCloud, radius: list) -> TriangleMesh:
    """ Triangle the point cloud into a mesh with the ball pivot algorithm

    Generates a mesh with the ball pivoting algorithm. Cleans up the geometry with several methods from open3d.
    Calculates different distances between the points in the point cloud to verify is the merge_close_vertices method
    is needed.

    Parameters
    -------
        pcd : o3d.cpu.pybind.geometry.PointCloud
                the point cloud to be made into a mesh
        radius : list
                the radius to be used in the algorithm

    Returns
    -----
        o3d.cpu.pybind.geometry.TriangleMesh
                the generated mesh
    """
    bpa_mesh = o3d.geometry.TriangleMesh.create_from_point_cloud_ball_pivoting(
                        point_cloud, o3d.utility.DoubleVector(radius))

    # Clean up geometry
    bpa_mesh.remove_degenerate_triangles()
    bpa_mesh.remove_duplicated_triangles()
    bpa_mesh.remove_duplicated_vertices()
    bpa_mesh.remove_non_manifold_edges()

    return bpa_mesh
```

Listing 25: Generate mesh with Ball-Pivoting algorithm

We also utilized several *Open3D* methods to clean up the mesh, by removing degenerate and duplicated triangles, duplicate vertices and non-manifold edges. This would also make the mesh file smaller in size because unneeded geometry was removed.

Using the Ball-Pivoting algorithm yielded much better results than the Poisson algorithm. Using it on the same surface data as shown in the Poisson mesh image, resulted in this mesh:

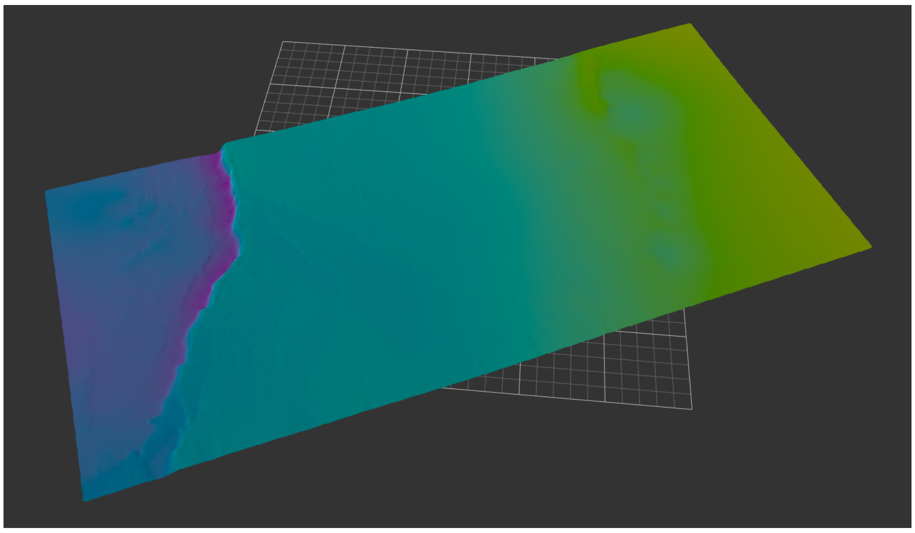


Figure 23: Ball-Pivoting Surface Reconstruction

Using this algorithm also presented some issues. We will talk more about them, and how we solved them, in the section below.

Problem areas

One of the main issues we discovered when using the Ball-Pivoting algorithm was that it was slow in comparison to the Poisson algorithm. When generating the mesh with default settings, the average surface would be processed in 6 minutes and 45 seconds. At first, we thought this was going to be a significant issue for the solution. But looking more into the issue, we found that the time it would take to generate a surface geometry would not matter much. The reason for this is that all of the surface data was static, so it would only have to be generated once. And if ever updated, the web-app would only load it after the geometry generation had been completed. As we go into some of the other issues, we will see that the generation time was improved alongside other issues.

Another notable issue we had was holes in the generated surface meshes. These came from inconsistencies with the value that the average distances between points was multiplied with. It resulted in issues that looked like this:

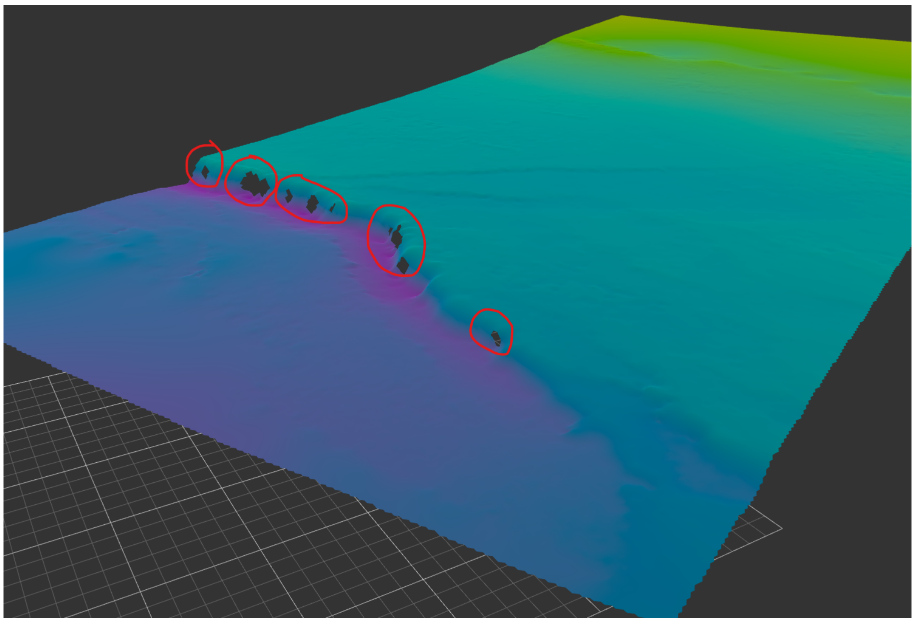


Figure 24: Mesh holes

We found that the inconsistencies were mainly a result from the radius value that the Ball-Pivoting algorithm was being provided. This value could be fixed by trial and error, but what we found later was that each surface required a custom value in order to be generated without holes. This meant that we had to find a way to generate a custom radius based on the requirement that a specific surface would have.

To mitigate this issue, we started calculating the difference between the minimum and maximum distance in the points of the point cloud. This helped us create a minimum and maximum threshold, which was a consistent way of recognizing the requirements that different surfaces could have. We also needed a value that could amplify the radius, similar to what the value did in our first iteration of radius. We used the average, minimum and maximum distance value to create this by adding them together and then multiplying them by 10. The multiplication was needed because the point cloud had been scaled down significantly to be able to estimate normals correctly. The final implementation of the radius method looked like this:

```python
def calculate_radius(pcd: PointCloud) -> list:
    """Calculate the radius of the point cloud points

    Calculate the radius of each point in the point cloud. Used later on the open3d TriangleMesh method.

    Parameters
    -------
        pcd : o3d.cpu.pybind.geometry.PointCloud
                the point cloud to calculate point radius

    Returns
    -----
        list
    """
    distances = pcd.compute_nearest_neighbor_distance()

    avg_dist = np.mean(distances)
    max_dist = max(distances)
    min_dist = min(distances)

    min_max_diff = max_dist - min_dist

    max_threshold = 0.032
    min_threshold = 0.001

    # Above the threshold of difference between min and max distances, a bit
    # extra amplify is needed to avoid holes in the computed mesh.
    if min_max_diff > max_threshold or min_max_diff < min_threshold:
        amplify = (avg_dist + min_dist + (max_dist * 1.3)) * 10
    else:
        amplify = (avg_dist + min_dist + max_dist) * 10

    radius =  [max_dist * amplify]

    return radius
```

Listing 26: Calculate radius final version

We also added an if statement to recognize a couple of surfaces that needed an extra amplifier. These surfaces had characteristics that were either over the maximum threshold, or below the minimum threshold. This method would adjust according to the distance between points on each surface, and make sure there were no holes in the meshes.

There were some issues with normal estimation as well. The orientation of the normals was flipped or in the wrong direction, this caused some minor holes to appear in the mesh. While the issue with holes in the mesh was fixed by the radius method, the orientation needed more work.

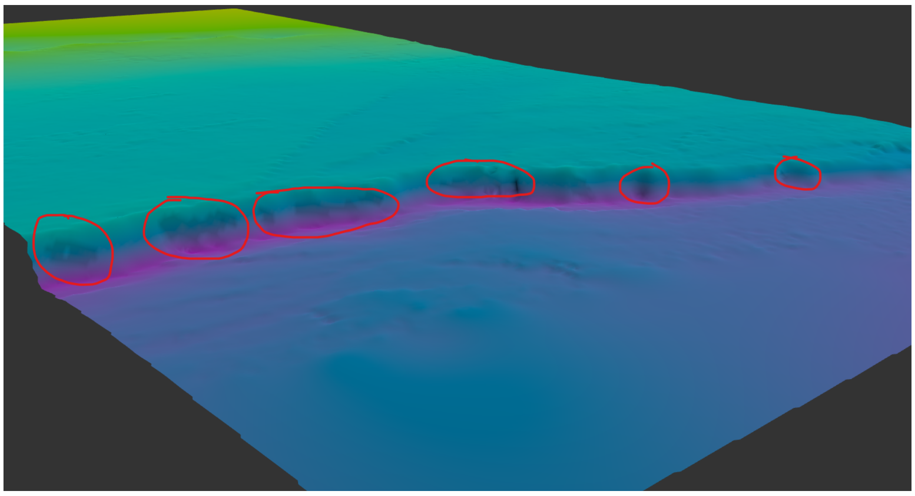


Figure 25: Normals with wrong orientation

As we can see here, the orientation of the normals does not follow the orientation of the surface. This makes it so that certain parts, like the ones outlined in the image, will point horizontally and not be affected by light in the correct way. This would leave marks similar to "patches" on the mesh. Our solution to this was to use the distance between points again to calculate the correct radius for normals estimation. The final method looked like this:

```python
def point_cloud_data(point_cloud_array):
    """Read the point cloud data from file

    Reads the point cloud from the array that is sent in and assigns the position and color
    from the point_cloud_array. Calculates the distances between the points in the point cloud
    to determine the difference between the minimum and maximum distance. The radius with which
    normals are calculated with are determined by the min_max_diff value.

    Parameters
    -------
        point_cloud_array : np.ndarray
                numpy array with vertex data

    Returns
    -----
        o3d.cpu.pybind.geometry.PointCloud
                the point cloud that will be used for generating geometry
    """
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(point_cloud_array[:,:3])
    pcd.colors = o3d.utility.Vector3dVector(point_cloud_array[:,3:6])

    distances = pcd.compute_nearest_neighbor_distance()
    max_dist = max(distances)
    min_dist = min(distances)

    min_max_diff = max_dist - min_dist

    min_threshold = 0.030
    increase_radius = 0.01

    # Increase the radius to calculate normals by a bit if the min max difference
    # is below the minimal threshold.
    if min_max_diff < min_threshold:
        radius = max(distances) + increase_radius
    else:
        radius = max(distances)

    max_neighbor = int(len(distances) / 100)

    pcd.estimate_normals(search_param=o3d.geometry.KDTreeSearchParamHybrid(
            radius=radius, max_nn=max_neighbor), fast_normal_computation=False)

    pcd.orient_normals_consistent_tangent_plane(100)
    pcd.normalize_normals()

    return pcd
```

Listing 27: Method to handle point cloud final version

As mentioned, we used the difference between the minimum and maximum distance between points to determine what the radius would be. The difference here is that we only used this to recognize the surface in an if statement. For the actual radius we took the maximum distance of the points, and set that as the radius. In some cases, as detected by the if statement, the radius needed to be increased a bit in order to return the optimal result. We also utilized a *Open3D* method to orient the normals correctly and normalize them.

When the holes and normals issues were resolved, we saw a significant improvement in how little time the geometry generation took. The surface that initially took 6 minutes and 45 seconds to generate previously, was now down to 1 minute and 35 seconds. The geometry generation was thus much more efficient.

7.3.3 Surfaces Cloud Function

After finishing the processing and geometry generation for surfaces, we needed to deploy this to a Azure cloud function. This function app would be triggered every time a new surface file was added to the Storage Account, and generate a geometry file which would then be stored for further use by the web app.

```
def main(blobIN: func.InputStream, blobOUT: func.Out[func.InputStream]):
   """Main function of blob trigger

   Is triggered when a file is added to azure blob storage. Gets the name of the file that triggered
   and checks the extension. Will currently only process .surface files.

   Parameters
   -------
   blobIN : func.InputStream
            file that triggered function read as inputstream
   blobOUT : func.Out[func.InputStream]
            processed file saved to storage with inputstream

   Returns
   -----
   No value
   """
   # Log some information about the blob
   logging.info(
        f"Python blob trigger function processed blob \n"
        f"Name: {blobIN.name}\n"
    )

   # grab the file with basename method
   data_file = os.path.basename(blobIN.name)

   # check whether the file that triggered function has file extension .surface
   if data_file.endswith('.surface'):
      # decode the bytes that are read from blob
      input_lines = bytes.decode(blobIN.read()).split("\n")

      # Send a request to grab if the surface name already exists
      req = requests.get("https://dt-api.azurewebsites.net/surfaces/exists",
      params={"surface_name": data_file},
      )

      # Check if the surface already exists in the database
      if req.content.decode('UTF-8') == "false":
         # process inputstream to a point cloud array
         point_cloud_array = process.process_inputstream(input_lines, '#', 2, 1000)

         # get the temp directory where the function app is running to temporarily store gltf file
         dir_path = tempfile.gettempdir()
         temp_store_geometry = dir_path + "/" + data_file + ".gltf"

         # generate the geometry and store the file in the temp folder
         geometry.generate_geometry(point_cloud_array, temp_store_geometry)

         # load and name the geometry node in the gltf file. needed for web-app
         pygltf.load_and_name_gltf(temp_store_geometry, data_file)

         # read the binary data from temp folder and read it to blobOUT for blob storage
         data = open(temp_store_geometry, "rb")
         blobOUT.set(data.read())

         surface_name = data_file.split(".")[0]
         surface_uri = blobIN.name.replace("data", "geometry").replace(".surface", ".gltf")

         surface = {
            "surface_name" : surface_name,
            "surfaces_uri" : surface_uri
         }
         CosmosDB.set(func.Document.from_json(json.dumps(surface)))

         logging.info(f"surface name : {surface_name} \n"
                     f"surface uri : {surface_uri} \n")

         logging.info(f"Data written to blob storage : {temp_store_geometry} \n")
      else:
         logging.info(
                f"Surface file {data_file} already exists. "
            )
```

Listing 28: Method that runs when function app is triggered for surfaces

The filename is retrieved with the help of the basename method. This is then used to verify that the file extension is ".surface", so that we can be sure that the type of the file being processed is correct. If the file extension is correct the input stream is read with the decode method and split for each new line it encounters. We also added an if statement to verify whether or not the file being processed exists in the database already or not. The processing will only run if the file does not already exist. From here, we can use the methods we created to process and generate the GlTF file. This is temporarily stored in local disk, due to further processing with *pygltflib*. *pygltflib* adds the name to the geometry node in the GlTF file. This is a requirement to be able to load the file in the web app.

The function app uses bindings to both trigger the function, and store the file in the Storage Account. This is done through the function.json file:

```json
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "name": "blobIN",
      "type": "blobTrigger",
      "direction": "in",
      "path": "rawstorage/Surfaces/data/{name}.surface",
      "connection": "ntnuccsstorage_STORAGE"
    },
    {
      "name": "blobOUT",
      "type": "blob",
      "direction": "out",
      "path": "rawstorage/Surfaces/geometry/{name}.gltf",
      "connection": "ntnuccsstorage_STORAGE"
    }
  ]
}
```

Listing 29: Bindings in function.json

The name, type, direction, path and connection of the binding is set. In this case we have a *blobTrigger* for input and an out binding pointing to blob storage for output. These are both connected to the 'ntnuccsstorage_STORAGE' which is the storage account located in Azure.

7.3.4 Surfaces Testing

Since one of our project goals was to learn more about test-driven development we created tests for the surface section. We planned out the methods we needed for parsing and processing purposes based on the data in the dataset. This contributed to the test driven approach, where we would create tests before implementing the methods. The implementation of tests in the surfaces section was a hybrid approach where some methods were developed on a test-first basis, and some with functionality-first basis. The most challenging aspect of testing surface generation was the *Open3D* section, and because of this most of the testing regarding *Open3D* was conducted manually in the beginning.

Testing the data processing and color generator for the surfaces were performed as planned. The tests were written before the implementation of the methods. To illustrate how we went about testing surfaces, we will showcase the process for the remove_elements method:

The first thing we had to identify was what functionality the method had to accomplish. We needed a method that would remove a certain amount of elements from a list. The list and the amount value would be passed as parameters, and the pop() method would run the operation "number" amount of times on the list. Writing the test case to achieve this functionality looked like this:

```python
def test_remove_elements():
    """remove elements from list"""

    l_test_1 = [1,2,3,4,5]
    l_test_2 = ['one', 'two', 'three']

    l_test_1 = remove_elements(l_test_1, 2)
    l_test_2 = remove_elements(l_test_2, 1)

    assert(len(l_test_1) == 3)
    assert(len(l_test_2) == 2)
```

Listing 30: Test method for remove_elements method

We started off by created two test lists. We would then run the method, removing a certain amount of elements, and then asserting the length of these lists. Having created the test, developing the method was a straightforward process, as it was possible to determine when the functionality was sufficiently implemented (When the test passed).

```python
def remove_elements(l_input: list[float], amount: int) -> list[float]:
    """Remove elements from a list

    Takes in a list and amount, and runs pop() on it amount times.

    Parameters
    -------
        l_input : list[float]
                the list to be modified
        amount : int
                amount of times to run pop()

    Returns
    -----
        list[float]
    """
    for _ in range(amount):
        l_input.pop()

    return l_input
```

Listing 31: Method to remove elements from list

Testing the geometry generation involved checking the generated files, and validating the point cloud object.

7.3.5 Surfaces Web-app

The surfaces are loaded through the use of the manifest, as explained in the frontend section of this chapter.

The surface method loads the GlTF file that is fetched through the API call.

```javascript
const Surface = memo(function Surface({ name, surface }) {
    const ref = useRef()
    const model = useGLTF(surface)
    const geometry = model.nodes[name].geometry

    geometry.computeVertexNormals()

    return (
        <mesh ref={ref} scale={1} geometry={geometry}>
            <meshStandardMaterial
                side={Three.DoubleSide}
                transparent={true}
                opacity={0.75}
                vertexColors={true}
            />
        </mesh>
    )
})
```

Listing 32: Surface method to load GlTF and compute normals

Here, we can see how the name of the surface file is used. This is needed in order to access the geometry within the GlTF file and properly display it in the scene. We utilize a ThreeJS method for the computation of vertex normals. This is a safeguard to make sure that the orientation of the normals are correct on the scene. The material of the surface is also set here, giving it some transparency and activating the usage of the vertex color generated from depth levels. The surfaces are then exported with:

```
function Surfaces({ surfaces, surfacesState }) {
    return (
        <group>
            {surfaces
                .filter(({ name }) => surfacesState[name])
                .map((item, index) => {
                    return (
                        <Surface
                            key={index}
                            surface={item.surface}
                            name={item.name}
                        />
                    )
                })}
        </group>
    )
}
```

Listing 33: Surfaces method that loads all surfaces

These are then passed into the "Surfaces" object in the viewport, which draw the surfaces on the scene.

7.4 Wells and well-logs

7.4.1 Well and Well-logs data

In order to visualize wells and well-logs, first we will need to read, parse and process a broad range of different data, provided both as part of the Smeaheia dataset, and other datasets available to Glex.

Wellbore 32/2-1 and 32/4-1 both have somewhat minimal data available for them, but wellbore 32/4-1 T2 on the other hand has the most data available, with 17 different logs and data curves to extract and visualize.

There are mainly two data formats we had to deal with for this project. Compiled logs, which are tabular logs manually compiled into spreadsheets by data managers at Glex. And composite logs, which are logs generated in the process of drilling and probing a wellbore, here in the form of LAS files.

We went back and forth on the format of the compiled logs quite a bit, the reason being that we needed a machine-readable format that was also easy for non-technical actors to author. We ended up with a custom excel sheet format. Essentially a single excel file per well, where the different sheets inside the excel file represent different well logs, some of which are mandatory, and others optional.

Listing 34 shows a truncated example of a permeability log. This showcases how most of the compiled data is formatted. The data is tabular, where each column represents a different measurement, and each row represents each measurement at different depths. By depth, we mean measured depth in meters relative to the drilling platform, and the curvature of the wellbore. This will be relevant later, as this depth is not the true vertical depth that we might expect it to be. Each log also has a set of measurements or labels for this row, in this example a measurement of permeability, which is what we want to visualize.

```
depth    permeability
1220.5    0.224
1221.5    0.097
1222.5    0.882
1223.5    0.215
1224.5    0.131
1225.5    0.123
1226.5    0.819
1227.5    0.102
1228.5    0.086
1229.5    0.091
1230.5    0.128
1231.5    0.316
1232.5    1.14
1233.5    0.021
1234.5    0.063
1235.5    0.053
```

Listing 34: Truncated example of well-log from 32/4-1 T2 showing depth vs. permeability

Listing 35 shows a simplified raw LAS file. The actual file is about 18 000 lines long, but this excerpt still highlights the overall structure of the file, and some of the most interesting information. We can see that the file consists of information blocks, indicated by the ˜symbol. These blocks contain various pieces of information about the well-log. For example, we can see the general information about the well and wellbore in the *well information block*. In the *curve information block*, we can see descriptions and units for the individual curves, or data points for the log. And in the *ASCII* block, we see the raw data, formatted not too dissimilar to a CSV file.

```
~VERSION INFORMATION
VERS.                                                 2.0:   CWLS Log ASCII Standard-VERSION 2.0
WRAP.                                                  NO:   One line per depth step
~Well Information Block
STRT.M                                           389.8880:   Top Depth
STOP.M                                          3164.3300:   Bottom Depth
STEP.M                                             .15240:   Depth Increment
NULL.                                            -999.250:   Null Value
FLD .                                                 Q32:   Field Name
WELL.                                             32/4-1:   NAME
WBN .                                          32/4-1 T2:   WELLBORE
NATI.                                                 NOR:   COUNTRY
~Curve Information Block
#MNEM.UNIT                                        API CODE    No Description
#=========                                        ========    == ===========
DEPT.M                                       00 001 00 00:     1  DEPTH
HAC.US/F                                                 :     2  Sonic Transit Time (Slowness)
HCAL.IN                                                  :     3  Caliper
HGR.GAPI                                                 :     4  Gamma Ray
HRD.OHMM                                                 :     5  Deep Resistivity
HRD1.CPS                                                 :     6  Far Thermal Neutron Count Rate
HRD2.CPS                                                 :     7  Far Thermal Neutron Count Rate
HRS.OHMM                                                 :     8  Micro Resistivity
~ASCII
  389.8880  -999.2500  -999.2500 -8219.5742  -999.2500  -999.2500  -999.2500 -8223.6055
  390.0404  -999.2500  -999.2500    22.0639  -999.2500  -999.2500  -999.2500      .3882
  390.1928  -999.2500  -999.2500    21.2065  -999.2500  -999.2500  -999.2500      .4003
  390.3452  -999.2500  -999.2500    20.5128  -999.2500  -999.2500  -999.2500      .3776
  390.4976  -999.2500  -999.2500    20.5980  -999.2500  -999.2500  -999.2500      .3799
  390.6500  -999.2500  -999.2500    21.4558  -999.2500  -999.2500  -999.2500      .4349
  390.8024  -999.2500  -999.2500    21.8847  -999.2500  -999.2500  -999.2500      .5474
  390.9548  -999.2500  -999.2500    22.2225  -999.2500  -999.2500  -999.2500      .7874
  391.1072  -999.2500  -999.2500    23.3564  -999.2500  -999.2500  -999.2500      .8468
  391.2596  -999.2500  -999.2500    26.9873  -999.2500  -999.2500  -999.2500      .9354
  391.4120  -999.2500  -999.2500    24.9395  -999.2500  -999.2500  -999.2500      .9342
  391.5644  -999.2500  -999.2500    23.4503  -999.2500  -999.2500  -999.2500      .9652
  391.7168  -999.2500  -999.2500    27.3499  -999.2500  -999.2500  -999.2500      .9842
  391.8692  -999.2500  -999.2500    25.1465  -999.2500  -999.2500  -999.2500      .9930
  392.0216  -999.2500  -999.2500    22.8582  -999.2500  -999.2500  -999.2500     1.0146
  392.1740  -999.2500  -999.2500    17.9492  -999.2500  -999.2500  -999.2500      .9905
  392.3264  -999.2500  -999.2500    19.0793  -999.2500  -999.2500  -999.2500      .9937
```

Listing 35: Simplified example of well-log from 32/4-1 T2 showing raw LAS file

7.4.2 Well and Well-logs parsing and processing

In order to work with well-logs, we first needed to obtain some metadata about the well and wellbore. This information is represented by the header and trajectory logs in the compiled well-logs. The well-log header gives us information such as the wellbore ID, for example *32/2-1*, and where the wellbore is located in the world. The trajectory log, on the other hand, gives us information about the curvature of the wellbore, which we will need later.

With this, the following needs to be completed in order to meet the requirements provided to us by Glex:

- Calculate true vertical depth for any measured depth.
- Generate a mesh representing the wellbore for visualization.
- Generate texture maps in order to visualize stratigraphy logs.
- Output logs as JSON documents for consumption by the web app.

Foremost, though, we had to validate and remove incorrect data. Mostly this involved removing null, NaN, and empty values, but we also tried to validate if the data was within realistic limits, like checking for negative values where they should not be. This proved difficult, as we did not always know what the data represented, and what these limits were. Therefore, we went with a more minimalistic approach where the data is represented mostly as given, except removing some values as mentioned. We then also wrote a set of tests for parsing and processing data, which we made use of through the rest of this implementation.

To calculate *true vertical depth*, (TVD, TVDKB, TVDSS), which was identified as an important piece of information by our requirements, Glex recommended we use the *minimum curvature method*. The minimum curvature method is a mathematical method for calculating the minimum curvature of a graph, or a wellbore in our case, based on a series of data points consisting of measured depths, angles, and azimuths, which is the data we had, from the wellbore survey, as part of the compiled logs.

However, the minimum curvature method only gives TVDs corresponding to the depth measurements it was given. In order to calculate TVD for any arbitrary depth measurement, we needed to find some continuous function for mapping measured depth to TVD. We could use regression in order to fit the function. The problem was determining the degree of the function to be utilized. Here we chose to experiment by using first linear, and then progressively higher degrees of polynomial functions, in order to see when the accuracy tapered off. Accuracy was measured and tested by splitting the dataset into a training dataset, and a test dataset. This way, we could test if the predictions the model makes for value we already know the answer to were close or not. This approach was inspired by the linear regression models we made as part of the PROG2051 - Artificial Intelligence course.

We ended up using a second degree polynomial function, as this provided a small advantage over linear regression, and the same accuracy as higher degree ones. We tested this on the limited data we had, though, and it is likely that wellbores that are much more curvy than the one we had, will require higher degree functions. Alternatively, a high fidelity solution would be to figure out how many curves the wellbore has ahead of time, as this is linked to the degree of polynomial function you will need to create this mapping. But considering we had already found a solution that was good enough for our purposes, we chose not to prioritize this, sacrificing a bit of accuracy, for the time to explore other more interesting aspects of this project.

Finally, this function could then be used to predict the TVD of any depth measurement in any of the other logs. With reasonable accuracy.

The minimum curvature calculations not only yields the true vertical depth, but also the relative deviations of the wellbore, or its curvature. Based on this data, we generated a polygon mesh representing the wellbore trajectory. This is useful for visualization purposes in the web app. We struggled to find any geometric shapes that fully captured the curvature of the wellbore, without becoming way too complicated to implement. Therefore, we had to come up with an algorithm for constructing this geometry ourselves. Since our input was a list of points, representing the points on the wellbore with the calculated deviation as x and y coordinates and the calculated true vertical depth as the z coordinate, we could form a pipe, by calculating vertex coordinates in a circle around each point, and by drawing triangles between these vertices. These vertices and triangles, being defined by indices, as well as appropriate texture coordinates, are then combined into a mesh. This forms a mesh, representing the wellbore as a pipe, made up of cylinder sections that go together without overlap. Listing 36 shows a simplified view of how this algorithm was implemented.

```python
def vertex_circle(
    x: float, y: float, z: float, segments: int, radius: float
) -> list[list[float]]:
    vertices = []
    for i in range(segments + 1):
        angle = (math.pi * 2.0) * i / segments
        vertex = [(math.cos(angle) * radius) + x, (math.sin(angle) * radius) + y, z]
        vertices.append(vertex)

    return vertices


def indices(segments: int, row: int) -> list[list[int]]:
    # We have segments+1 number of vertices in each circle
    # We want to make 2 triangles/1 quad per segment
    vertices_per_row = segments + 1
    indices = []
    for i in range(vertices_per_row):
        i = i + (vertices_per_row * row)
        indices.append([i, i + 1, i + vertices_per_row])
        indices.append([i + 1, i + 1 + vertices_per_row, i + vertices_per_row])

    return indices


def uvs_circle(z: float, max_z: float, segments: int):
    # Can't generate UVs for cylinders with fewer than 2 segments. (What would the shader blend between?)
    assert segments >= 2

    uvs = []
    for i in range(segments + 1):
        u = float(i) / float(segments)
        v = z / max_z
        uvs.append([u, v])

    return uvs

def make_pipe(
    points: np.ndarray, segments: int, radius: float
) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    vertices = []
    faces = []
    uvs = []

    for [x, y, z] in points:
        vertices.extend(vertex_circle(x, y, z, segments, radius))

    rows = len(points)
    for row in range(rows - 1):
        faces.extend(indices(segments, row))

    max_z = points[-1][2]  # The max z value will be the last one
    for [_, _, z] in points:
        uvs.extend(uvs_circle(z, max_z, segments))

    return (
        np.array(vertices, dtype="float32"),
        np.array(faces, dtype="uint32"),
        np.array(uvs, dtype="float32"),
    )
```

Listing 36: Code for generating wellbore geometry

Then, we generated texture maps, based on the lithostratigraphy and chronostratigraphy data from the compiled logs. These texture maps can be applied onto the wellbore mesh, in order to visualize the lithological units and groups, and what ages the different strata that the wellbore intersects. Which was identified as information of interest to Glex. The texture is generated by drawing colored bands on a palette, where each unique color represent either a lithological unit, or a geologic age. Listing 37 shows the code for generating these textures, and figure 26 shows an example of the resulting texture.

```python
    # series here is the log data itself in the format [(color, top, base)]

    for (_, row) in series.iterrows():
        color: tuple[int, int, int] = row["color"]  # type: ignore

        # Convert 0..max_depth into 0..height range
        start = math.floor((float(row["top"]) / max_depth) * heigth)  # type: ignore
        end = math.floor((float(row["base"]) / max_depth) * heigth)  # type: ignore

        # Draw a stripe into a section of the image, representing a formation, group, or age
        for y in range(start, end):
            # print(y, color)
            for x in range(width):
                img.putpixel((x, y), color)
```

Listing 37: Code for generating texture maps

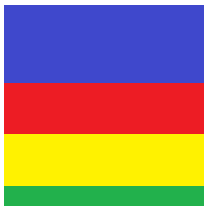


Figure 26: Resulting texture map

Finally, since these well-logs will be stored in a document database, we needed to convert them from various idiosyncratic tabular formats, into a unified JSON format. This was achieved by extracting only the minimal relevant metadata provided by all logs, and placing this as fields in a sort of header to the document. And then by extracting the tabular data that represents the log itself, converting these into a records format, and placing it at the end of the document.

```json
{
    "id": "32-4-1 T2-permeability",
    "wellbore": "32/4-1 T2",
    "mnemonic": "permeability",
    "description": "Measured depth vs. permeability",
    "unit": "kl-h",
    "columns": [
        "depth",
        "permeability",
        "tvd"
    ],
    "data": [
        {
            "depth": 1220.5,
            "permeability": 0.224,
            "tvd": 1220.4591135430153
        },
        {
            "depth": 1221.5,
            "permeability": 0.097,
            "tvd": 1221.4590599015503
        },
        {
            "depth": 1222.5,
            "permeability": 0.882,
            "tvd": 1222.4590061768924
        },
        {
            "depth": 1223.5,
            "permeability": 0.215,
            "tvd": 1223.4589523690418
        },
        {
            "depth": 1224.5,
            "permeability": 0.131,
            "tvd": 1224.458898477998
        },
        {
            "depth": 1225.5,
            "permeability": 0.123,
            "tvd": 1225.4588445037614
        },
        {
            "depth": 1226.5,
            "permeability": 0.819,
            "tvd": 1226.458790446332
        },
        {
            "depth": 1227.5,
            "permeability": 0.102,
            "tvd": 1227.4587363057096
        },
        {
            "depth": 1228.5,
            "permeability": 0.086,
            "tvd": 1228.4586820818943
        },
        ...
    ],
}
```

Listing 38: A truncated example of processed well-log

In order to implement all of this functionality, we tried to reuse existing components and libraries as best we could. We found *lasio* for parsing LAS files, *pyglftlib* for generating GlTF files, *wellpathpy* for doing minimum curvature calculations, and *Pillow* for encoding image files/textures. In addition to a set of Python libraries we had previous experience with from the PROG2051 - Artificial Intelligence course for doing data processing, namely Pandas, *NumPy*, *Matplotlib*, and *SciPy*.

7.4.3 Well and Well-logs cloud functions

There are actually two separate cloud functions for processing well-logs. There is one for compiled well-logs, that is triggered by '*.xlsx' files being uploaded to the 'well-logs' directory in blob storage. And the other is for composite well-logs, and is triggered by '*.las' files being uploaded into the same directory.

These functions both take the well logs as input, use the internal modules that implement the functionality we talked about earlier, process the well-logs into a suitable data format, in addition to generating auxiliary assets. Which are both then uploaded to the database and blob storage respectively.

7.4.4 Well and Well-logs visualization in web app

Visualizing the well-logs was done by generating graphs directly in the frontend, see the implementation section for the frontend for more details. Figure 27 shows what these graphs ended up looking like in the frontend.

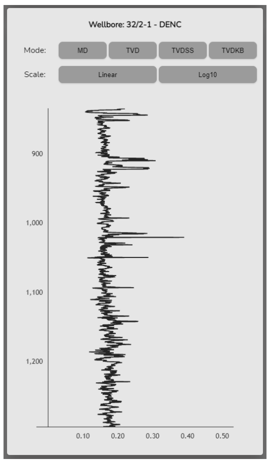


Figure 27: A well-log as visualized by the frontend

Since the frontend implements full 3D graphics capabilities, the wellbore geometry we generated can be visualized by rendering a mesh, placed at the correct location for this well. Figure 28 shows what this looks like in the frontend. Admittedly, it is difficult to make out the fact that this wellbore actually curves. In order to visualize the mesh, it was scaled by increasing the radius to 100 meters, making any of the 1-5 meter deviations that the wellbore actually has difficult to spot in this case. However, it is important to implement with respect to this being a generic well and well-log processing service, that could potentially have to handle a wellbore with significant curves and deviation, which the service will now handle gracefully.

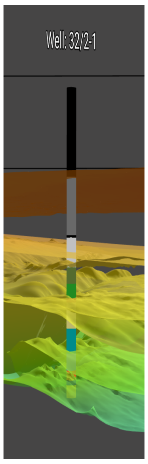


Figure 28: A wellbore, intersecting with surfaces, and with a lithostratigraphy map applied

7.5 Faults

One of the data sources that needed to be parsed and displayed was Faults. Fault sticks are lines consisting of several points located in the world. By combining these points, drawing a line between them, we get a "stick". We can further combine these sticks, by creating a plane between them. We then end up with something called a fault. A fault is a fracture between two "blocks" of landmass. These faults allow the different "blocks" to move relative to each other[53].

7.5.1 The faults data

The faults data was structured in text files with space separated values. Each file represents an individual fault collection. Within each file, there are several lines, each of which represents a single point in the world. All points have some metadata associated with it, namely a fault name and a point index. The fault name clarifies which fault any given fault stick belongs to. By combining the fault name and the point index, one can tell which fault stick any one point belongs to.

The data looks like this (note that the row with column explanations is not part of the actual data):

```
#unused    unused  unused  x            y             z          fault name             index
INLINE-    1140     976    554563.82213 6734960.79353 1576.22278 Fault_interpretation_1 1
INLINE-    1110     976    554874.79295 6735143.23622 1930.71108 Fault_interpretation_1 1
INLINE-    1098     976    555012.32603 6735223.92516 2202.36315 Fault_interpretation_1 1
INLINE-    1130    1008    554461.82192 6735364.71100 1576.22278 Fault_interpretation_1 2
INLINE-    1110    1008    554681.50972 6735493.59908 1794.81449 Fault_interpretation_1 2
INLINE-    1094    1008    554837.29941 6735584.99894 1939.17816 Fault_interpretation_1 2
INLINE-    1078    1008    555028.99375 6735697.46362 2081.28394 Fault_interpretation_1 2
INLINE-    1108    1040    554488.22650 6735843.96194 1559.28863 Fault_interpretation_1 3
INLINE-    1100    1040    554589.24637 6735903.22904 1693.06845 Fault_interpretation_1 3
INLINE-    1076    1040    554844.83882 6736053.18194 1962.46262 Fault_interpretation_1 3
INLINE-    1060    1040    555000.62851 6736144.58180 2113.17660 Fault_interpretation_1 3
```

Listing 39: The fault data format

The first three columns contains information regarding how the data was captured and is thus not relevant for the visualization we are going to do. The last 5 columns contains the x coordinate, y coordinate, z coordinate, fault name and the point index. In the data example, we can see that there are three different fault sticks, from one fault: *Fault_interpretation_1*.

7.5.2 Testing Faults

The first thing that was defined when the Faults were developed was the tests. By defining the tests beforehand, the actual development work became easier. The general idea behind the tests was to verify that the correct types of data had been parsed correctly. Since the language chosen for the faults parser was Python, verifying that the data had been parsed to the correct format was non-intuitive. The problem of verifying that the data types had been parsed to the correct type was solved through the use of classes. The classes had attributes, and a constructor was utilized to ensure that the correct data was assigned to the correct variables.

An example of how point creation was tested:

```python
def test_point_creation():
"""Verifies that a single point is created successfully"""
point = types.Point(1,2,3)
assert point.x == 1
assert point.y == 2
assert point.z == 3
```

Listing 40: Point class test

With the test defined, it was intuitive to develop the actual class that served as a Point:

```python
class Point:
"""Contains information about a single point"""

def __init__(self, x: float = 0, y: float = 0, z: float = 0) -> None:
    """Initializes a new Point object
    Parameters
    -------
    x : float, optional
        The x coordinate of the point, by default 0
    y : float, optional
        The y coordinate of the point, by default 0
    z : float, optional
        The z coordinate of the point, by default 0
    """
    self.x = x
    self.y = y
    self.z = z
```

Listing 41: Point class implementation

7.5.3 The faults parser

The faults parser transforms the plain text fault files into a more usable JSON format. The first thing we did when creating the parser was defining the different data types. From our previous experience creating a parser in Haskell during the PROG2006 - Advanced Programming course, we found that having specific types for the objects we were parsing was a really powerful feature. Although no such thing as the Haskell type system exists in Python, we created classes that were meant to serve somewhat of the same purpose. By creating classes for each of the data types to be parsed, testing became easier.

The fault stick parser accepts two different types of input: A file path to a file containing fault data or a list of fault stick strings. If the class receives a file path, the lines contained within the file are read. All the lines are then sent onward to the *parseFaultSticks* function, where the parsing occurs.

```python
def parseFaultSticks(self, lines: list[str]):
    """Converts lines of Fault Sticks into faults

    Parameters
    -------
    lines : list[str]
        List of lines (strings) that is going to be converted into faults
    """

    self.faults = types.Faults(fault_collection_name=self.filename)

    current_fault_name = ""

    for line in lines:
        [x, y, z, fault_name, stick_number] = line.split()[3:]

        # If the line is a new fault
        if fault_name != current_fault_name:
            current_fault_name = fault_name
            self.faults.faults.append(types.Fault())
            current_fault = self.faults.faults[-1]
            current_fault.faultName = current_fault_name
            current_fault.faultSticks = []
            current_stick_number = 1
```

Listing 42: New fault encountered

The parser processes one line at a time. The first thing that happens is that the parser splits the line into 5 different variables. The *current_fault_name* variable stores the name of the current fault being processed. The newly parsed fault's name is checked against this variable, this is to check if the line currently being parsed belongs to the fault of previous iterations, or is an entirely new one. If the newly parsed fault name is not equal to the previous one, we have encountered a new fault. When a new fault is encountered, the *current_fault_name* is updated to reflect the change, a new Fault gets added to the faults list, and the current stick number is set back to 1.

```python
        # If the line is a new faultstick
        if stick_number != current_stick_number:
            current_stick_number = stick_number
            current_fault.faultSticks.append(types.FaultStick())
            current_faultstick = current_fault.faultSticks[-1]
            current_faultstick.points = []

        # Adds the point to the current faultstick in the current fault
        current_faultstick.points.append(types.Point(float(x), float(y), float(z)))
```

Listing 43: New fault stick encountered

The next thing that gets checked is the line's stick number. As explained previously, all lines have a number indicating which fault stick within a fault it belongs to. If the *stick_number* is different from the *current_stick_number*, we have encountered a new fault stick. When a new fault stick is encountered, the *current_stick_number* gets updated to reflect this change, and a new fault stick is appended to the fault stick list of the current fault. The *current_faultstick* is the variable that always points to the current fault's current fault stick, the fault stick that should receive the next point.

Finally, the x, y and z coordinates are added as a point to the correct fault stick.

7.5.4 Faults cloud function

The parser runs in an Azure cloud function. This function monitors the blob storage, and when a new fault collection is uploaded, the function is triggered. The function takes the filename of the newly uploaded file, connects to the blob storage container, and fetches the content of the provided file. After the content has been read, the lines of the file are separated into a list of strings.

```python
def main(myblob: func.InputStream, CosmosDB: func.Out[func.Document]):
    """Takes in a blob (filestream), parses it into Faults, then writes them to CosmosDB

    Parameters
    -------
    myblob : func.InputStream
        The blob file containing the data to be read
    CosmosDB : func.Out[func.Document]
        The binding to CosmosDB, used to write the data
    """

    # If the blob is of the correct file extension
    if os.path.splitext(myblob.name)[1] == ".faultsticks":

        # Extract the filename of the blob
        filename = os.path.basename(myblob.name)

        # Fetch the connection string
        CONNECTION_STRING = os.environ["ntnuccsstorage_STORAGE"]

        # Create the container client
        blob = BlobClient.from_connection_string(conn_str=CONNECTION_STRING,container_name="rawstorage",blob_name='Fault_Sticks/data/'+filename)

        # Read the blob
        lines = bytes.decode(blob.download_blob().readall()).split("\n")
```

Listing 44: Connecting to the Fault's blob client

The function then verifies that the newly uploaded fault collection does not previously exist in the database. The checks are conducted by sending a request to the REST API. The reason we choose to utilize the REST API for this purpose, was to avoid creating an unnecessary amount of connections to the database. The lines are then consumed by the parser, and subsequently converted into CosmosDB documents, before finally being inserted into the database.

```python
    # Check if the fault collection already exists in the database
    req = requests.get(
        "https://dt-api.azurewebsites.net/faults/exists",
        params={"fault_collection": filename},
    )

    if req.content.decode('UTF-8') == "false":

        # Parse lines to Faults
        fsp = parser.FaultStickParser(lines=lines, filename=filename)

        documents = []
        # Create list of documents to insert into DB
        for doc in fsp.faults.toSeparatedJSON():
            documents.append(func.Document.from_json(doc))

        # Set the DB output to the generated list of documents
        CosmosDB.set(func.DocumentList(documents))
    else:
        logging.info(
            f"Fault collection that already exists: {filename} was attempted parsed"
        )
```

Listing 45: Parsing and inserting Faults data into CosmosDB

7.5.5 Faults in the web app

The REST API has endpoints available for fetching faults data. When the web app loads, the fault data is fetched from the endpoints. There is some further geometry generation that has to be done before the faults can be displayed to the user. The reason we chose to generate this geometry on demand was due to the file sizes of the generated geometry. The data stored about the fault sticks in the database is restricted to the bare minimum required for the front end to be able to generate proper geometry. By structuring our data this way, we reduced the size of the data being transmitted from the REST API to the web app. The generation work that needs to be done is cheap from a system resource point of view, ensuring that this generation is fast and efficient.

Two types of geometry are generated for the faults. Lines are generated, representing the fault sticks. The other type of geometry is the mesh, consisting of a "plane" illustrating the surface between the fault sticks in a given fault. In this particular case, the fault sticks are black, and the "surface" is red.

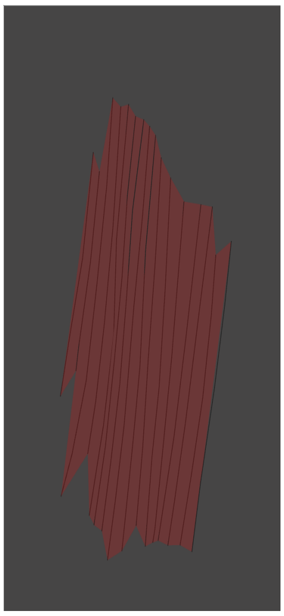


Figure 29: Example of a fault

The generation of these geometries are done on demand. We utilize state to ensure that only the faults that have been selected for displaying are processed. This ensures that we do not generate any unnecessary geometry. This is done through a map function, where the list of all faults (*data*) are iterated over. Each element in this list contains fault data, this includes the name of the fault, the fault sticks with its accompanying points, etc. We utilize the name of the fault to check against the *faultState*, verifying whether or not this particular fault is currently toggled for visualization.

```
function Faults({ faultsState, data }) {
    // Filter fault data to only what is enabled via faultsState
    const faults = []
    data.map((fault,) => {
        if (faultsState[fault.fault_name])
            faults.push(fault.fault_sticks)
    })
```

Listing 46: Add all faults that have been enabled using the toggle menu

After all the list of faults to be further processed has been generated, it is iterated over, creating a Fault component for each fault in the list.

```
    return faults.map((fault_sticks, index) => (
            <Fault key={index} fault_sticks={fault_sticks} />
    ))
}
```

Listing 47: Creating fault components for each Fault

Inside the fault component, the JSON data is further simplified. The reason this has to be done is because of the geometry that is going to be generated. To visualize the fault sticks, we are utilizing ThreeJS Line geometry. Line geometry expects a list of points. This list of points will also be utilized for generating the "fault surface" mesh.

```
const Fault = memo(function Fault(props) {
    let fault_sticks = []
    for (let i = 0; i < props.fault_sticks.length; i++) {
        fault_sticks.push([])
        for (let j = 0; j < props.fault_sticks[i].length; j++) {
            fault_sticks[i].push([
                props.fault_sticks[i][j]["x"] / 1000,
                props.fault_sticks[i][j]["y"] / 1000,
                -props.fault_sticks[i][j]["z"] / 1000,
            ])
        }
    }
```

Listing 48: Fault data complexity reduction

The result of this operation is the nested list *fault_sticks*. All point values were divided by 1000, to match the rest of the digital twin (assuming we have covered the reason behind this number somewhere else.). Due to the fact that the input data organized its z coordinate in the wrong direction, we also had to flip the z values.

The next step was to utilize the nested list to generate the actual geometry. For the line geometry, we created a *Fault_Stick* component. This component takes a *points* argument, these supplied points (A list of points) will be transformed into line geometry.

```
const Fault_Stick = memo(function Fault_Stick(props) {
    return (
        <Line
            points={props.points}
            color="black"
            transparent={true}
            opacity={0.5}
            lineWidth={0.5}
        />
    )
})
```

Listing 49: Fault stick geometry generation

Next up was the "surface" of the faults, which consists of mesh geometry. We needed to generate both the necessary indices in addition to the vertices. The mesh consists of several squares. Each square is drawn between the start and end points of two subsequent fault sticks. For this reason, we needed to have access to both the current fault stick, alongside the previous one. We solved this by assigning the current fault to the *previous_fault* variable, to be used in the next iteration of the processing.

```
let previous_points = null

return fault_sticks.map((points, index) => {
    const indices = new Uint32Array([0, 1, 2, 3, 2, 1])

    if (previous_points != null && index < fault_sticks.length) {
        let vertices = new Float32Array([
            points[0][0],
            points[0][1],
            points[0][2],
            points[points.length - 1][0],
            points[points.length - 1][1],
            points[points.length - 1][2],
            previous_points[0][0],
            previous_points[0][1],
            previous_points[0][2],
            previous_points[previous_points.length - 1][0],
            previous_points[previous_points.length - 1][1],
            previous_points[previous_points.length - 1][2],
        ])

        previous_points = points

        return (
            <group key={index}>
                <Fault_Stick points={points} />
                <mesh>
                    <bufferGeometry>
                        <bufferAttribute
                            array={indices}
                            attach={"index"}
                            count={indices.length}
                            itemSize={1}
                        />
                        <bufferAttribute
                            attachObject={["attributes", "position"]}
                            count={vertices.length / 3}
                            array={vertices}
                            itemSize={3}
                        />
                    </bufferGeometry>
                    <meshBasicMaterial
                        side={Three.DoubleSide}
                        transparent={true}
                        opacity={0.2}
                        color="red"
                        flatShading={true}
                    />
                </mesh>
            </group>
        )
    }

    previous_points = points
    return <Fault_Stick key={index} points={points} />
})
```

Listing 50: Fault geometry generation

To limit edge conditions, we included an if statement to ensure that *previous_points* is assigned, and that the index has not gone out of range. In the situations where these conditions apply, only the line geometry (fault sticks) are displayed.

7.6 Realtime data

7.6.1 Realtime data generation

Since we had to generate realtime data ourselves, we had to define some characteristics that the algorithm had to satisfy. The algorithm had to be deterministic, meaning that a specific input would always provide the same output. The reason we had to provide this trait was due to our choice of not storing realtime data in the database, by creating a deterministic algorithm we could regenerate data at any point in time, whenever we want to. The second requirement was that the data interval was to be between 0 and 3.

The function for generating the realtime data looked like this:

```python
def flowrate(timestamp: float) -> float:
    """Mocks a flowrate value for a given point in time

    Parameters
    -------
    timestamp : float
        A UNIX timestamp (seconds elapsed since Jan. 1 1970, exc leap seconds)

    Returns
    -----
    float
        The flowrate at any given second
    """
    # Seed random so we will always get the same output
    random.seed(timestamp)

    # Generate a random offset
    offset = (random.random() - 0.5) / 5

    # Convert the time to the interval 0-119
    modtime = timestamp % 120

    # Convert modtime to the interval 0-1
    t = modtime / 120

    # Calculate the current flow rate
    rate = quadbeziercurve(0, 0, 2 + offset, t)

    # Returns the flowrate, or 0 if the flowrate is negative (as this is not possible)
    return max(rate + offset, 0)
```

Listing 51: Function for generating realtime flow rate

The first thing that happens is that the random is seeded, ensuring that the rest of the generation is deterministic. From this seeded random an offset is generated, this is to simulate an unpredictable offset in the general flow rate, due to some natural condition. Since the flow rate moves in cycles, the timestamp is converted to an interval of 0-119, further this value is converted to the interval 0-1. We do this conversion to get a value we can use in the interpolation function. We set the interpolation midpoint to be the maximum value, and the start and endpoint to be 0.

7.6.2 Realtime data endpoint

The realtime data endpoint takes a timestamp as input, and runs the generation algorithm on it. This gives a flow rate for any given second in time, which is returned as a response.

```python
@realtime_blueprint.route("/realtime/flowrate/<float:ts>", methods=["GET"])
def flowrate_timestamp(ts: float):
    """Generates a flowrate for a given point in time

    Parameters
    -------
    ts : float
        UNIX timestamp

    Returns
    -----
    json
        Returns time and a flowrate
    """
    timenow = datetime.fromtimestamp(ts)
    return jsonify(
        {"time": timenow.strftime("%Y-%m-%d %H:%M-%S"), "flowrate": flowrate(ts)}
    )
```

Listing 52: Realtime endpoint

7.6.3 Realtime data chart

Whenever the diagram for the realtime data component is active, requests are continuously sent to the REST API to fetch the current flowrate. This is done through a useEffect() hook, which sets an interval of 1000ms between each request.

```javascript
    useEffect(() => {
        function fetchRealtimeData() {
            let timestamp = Number(Math.floor(Date.now() / 1000) + props.offset).toFixed(2)

            fetch("https://dt-api.azurewebsites.net/realtime/flowrate/" + timestamp)
                .then(res => res.json())
                .then(data => {
                    setFlowdata(flowdataRef.current.slice(-100).concat(data));
                })

        }
        const intervalID = setInterval(() => {
            fetchRealtimeData()
        }, 1000)

        return () => clearInterval(intervalID)

    }, [props.offset])
```

Listing 53: Hook for fetching realtime data continously

The data is limited to 100 entries, meaning older entries gets overwritten once 100 timestamps worth of realtime data has been collected.

When the data has been collected, it is displayed in a Victory graph.

```
return (
    <div className='graph'>
        <h3>{props.wellname} Flowrate</h3>
        <div
            style={
                {textAlign: 'center'}
            }
        >Current flowrate: </div>
        {flowdata.length > 1 && (
            <div
                style={{
                    backgroundColor: "lightgrey",
                    paddingTop: '0.5rem',
                    paddingBottom: '0.5rem',
                    textAlign: 'center',
                    textJustify: 'center',
                }}
            >
                {flowdata[flowdata.length - 1]["flowrate"]} / s
            </div>
        )}

        {flowdata.length > 1 && <VictoryChart>
            <VictoryLine
                x="time"
                y="flowrate"
                data={flowdata}/>
            <VictoryAxis tickFormat={(t) => t.split(" ")[1].replace("-",":")}
                         tickCount={5}
                         domain={{x: [0,3]}}/>
            <VictoryAxis dependentAxis={true}/>
       </VictoryChart>}
    </div>
)
```

Listing 54: Code for generating a line graph with Victory

7.7 Horizons

As mentioned in the technical design section, the function app and visualization part of the horizons were not implemented. In this section, we will briefly go over the implementation that was done, which is the extraction of point cloud from the shape files, parsing, and generating of point geometry.

Extracting the point cloud from the shape files was done with GeoPandas. Since we did not get to the implementation of a function app for the horizons, the implementation was done locally, saving the output to files. The implementation of reading the shape file and retrieving the point cloud looked like this:

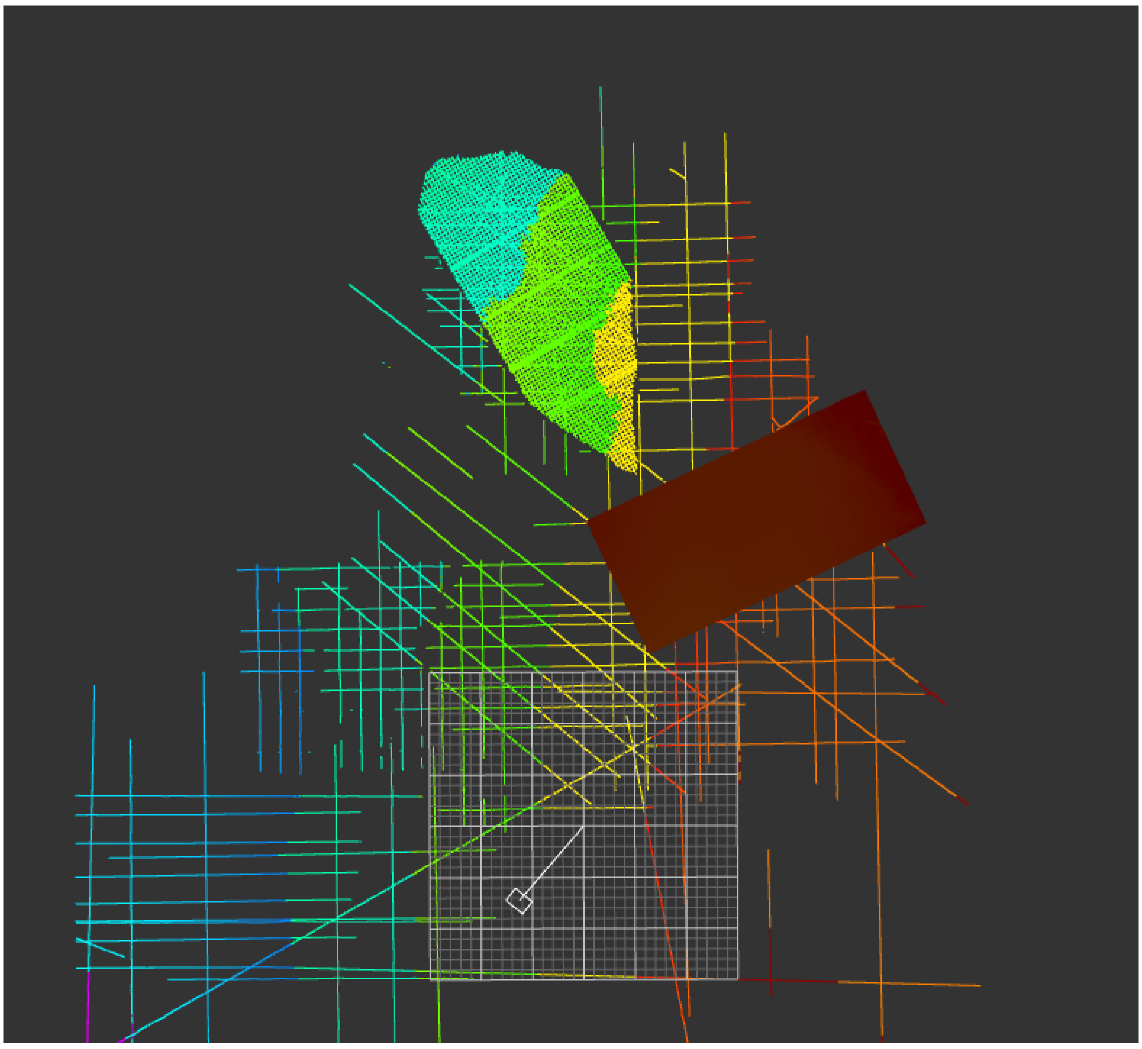
```
def read_shape_file(filepath):
    gdf = gpd.read_file(filepath)

    coordinates = gdf["geometry"]

    return coordinates

def retrieve_coordinates(coordinates):
    l_return = []

    for i, _ in enumerate(coordinates):
        l_return.append([coordinates[i].x, coordinates[i].y, coordinates[i].z])

    return l_return
```

Listing 55: Code for reading the shape file

Here, we extract the coordinates with the help of GeoPandas read_file method, to find the geometry element. And in the next method we extract each coordinate and append them to a list. This is later written to a file as a point cloud for further processing.

The data processing and color generation is done the same way as with the surfaces implementation. The only difference is that this was not adjusted to work as a function app. The geometry generation is done with

Open3D. Here, we read the processed point cloud and colors, and then write the point cloud as a geometry file. This generated point geometry, and the result can be seen below.



8 Deployment

When we were given this project, it came with the requirement that the solution had to be deployable to Microsoft Azure. This chapter explains how the different components of the project were deployed.

8.1 Deploying to Azure

Before looking at our deployment, let us present some of the nomenclature around Azure. First of all in order to run anything in Azure, we needed a subscription. The subscription serves to facilitate paying Azure for their services, keeping track of costs and where they originate from, and for setting budgets and warnings, so that you do not end up paying too much.

Second, we required a resource group for our resources. Resources in Azure are any of the individual services we spin up. Like a virtual machine, a database, or anything else. A resource group is a folder for these resources to live in, and facilitates things such as keeping track of all the resources, sharing common properties, and deleting them all when we are done with them. Resource groups are not strictly mandatory, but based on Microsoft's advice[25], it is best practice to use them to organize your resources. We experienced ourselves how useful they were when we had to migrate which region our resources were located in due to some needed functionality not being available in our original region of choice.

With a subscription and resource group set up, we could now create the individual resources we needed for our project. That being, an *Azure Static Web App* for our frontend, an *Azure Web App* for our REST API, a *Storage Account* for blob storage, a *Cosmos DB* instance as our database, and three different *Azure Functions* for the different data processing components.

Figure 30 shows how all the resources and how the data flows between them. Green arrows indicate reading, blue arrows indicate writing, and orange arrows indicate waiting on triggers. We can also see that all the resources are in one resource group, managed by one subscription.

These triggers are one of the really interesting things about our solution. The cloud functions are all set up to watch for new files being uploaded to blob storage. When a new file is uploaded, the relevant function, based on file type, will run, processing the data in whatever way appropriate.

The diagram also indicates how the two main actor archetypes interact with the deployment. These being, the geologists and other users, who mainly interact with the frontend, none of which require any access to Azure at all. And the data managers, who upload data to the system, which requires read-write access to blob storage in Azure. Not included in the diagram are the administrators and developers, both of whom require access to create, destroy and manage resources in Azure. We incorporated this information into the user roles we created in Azure, in order to effectively secure access to our resources.

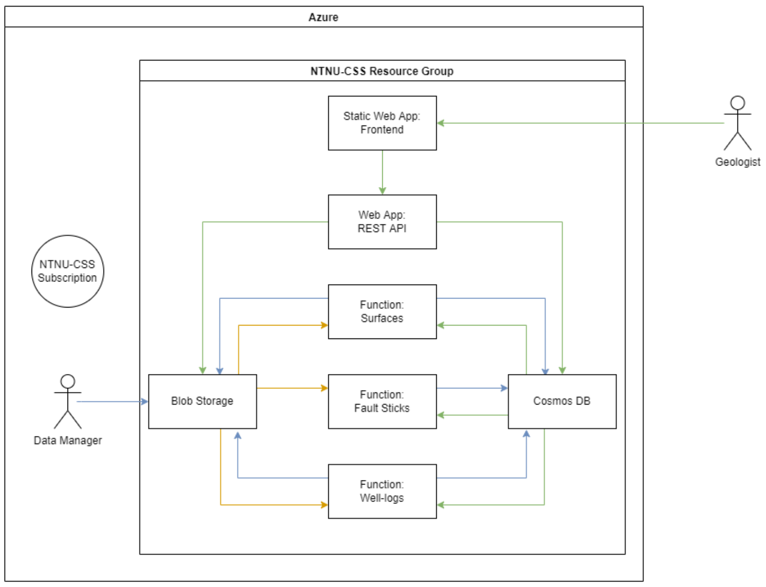


Figure 30: Diagram showing the architecture in Azure

8.2 Continuous Integration and Continuous Deployment

Throughout this project, we have made use of continuous integration[10] and continuous deployment[9]. Continuous integration, or CI, is the practice of automatically, and continuously, building, testing and asserting that new code integrates correctly before merging into a central repository or branch. Continuous deployment, or CD, takes this a step further, and automates the deployment of code into production, if and only if the previous tests and assertions were successful.

CI and CD are useful as they automate the precarious and finicky process of building and deploying software to the cloud, without losing the ability to perform quality assurance before pushing code into production.

The system we used to automate CI and CD is GitHub Actions. GitHub Actions comes built-in to GitHub and is one of the most intuitive CI/CD systems to incorporate with GitHub. It does cost a little bit of money to use, but it did not amount to much throughout our project.

GitHub Actions consists of a set of primitives which can be composed into a CI/CD system. These are:

- Actions: Pre-made *apps* to run on your code to perform some action. (Like install python, run tests, lint the code, etc.)
- Workflows: A workflow is a specification for how to automate the CI and CD related to one component of the overall project.

- Jobs: Jobs are a set of steps to perform in order to achieve some high level objective, isolated from other jobs.
- Steps: Steps are part of a job and perform some specific action. Either through using actions or by running shell commands.

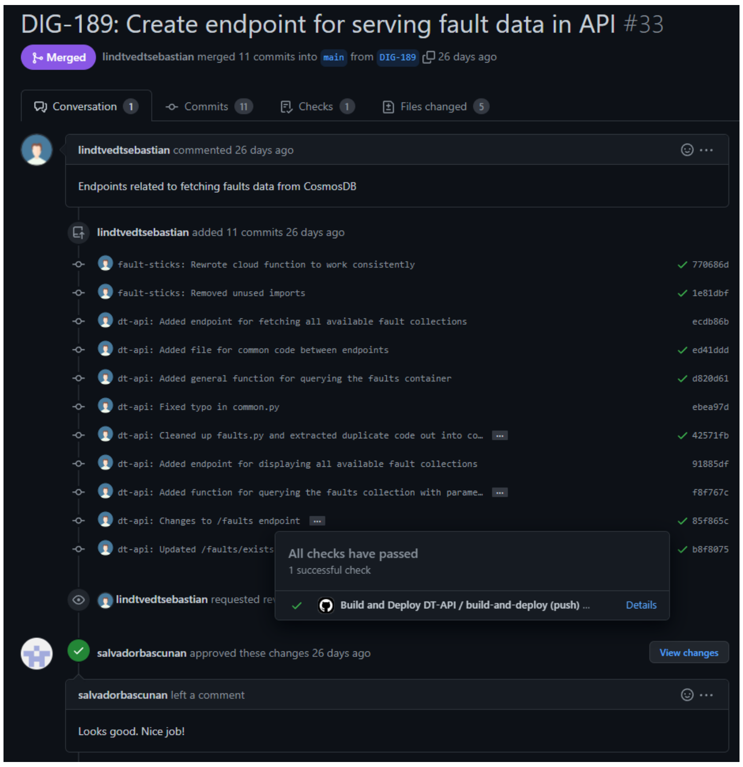
There are multiple workflows as part of our CI and CD. But they are all very similar, so we will only showcase the workflow for the REST API. The other workflows follow more or less the same structure and sequence, only differing in some key configuration.

The workflow only runs when it should. This results in the CI running when we open a pull request, as seen in figure 31.

We only want to invoke the workflow when we have actually made relevant changes. To do this, we instruct GitHub to only invoke the workflow when code in the *dt-api* directory changes, or when the workflow itself changes. We also instruct GitHub to invoke the workflow both for when we push to or merge to the main branch, and when we are working on a pull request.

```yaml
on:
  push:
    branches:
      - main
    paths:
      - "dt-api/**"
      - ".github/workflows/dt-api.yml"
  pull_request:
    types: [opened, synchronize, reopened, closed]
    branches:
      - main
    paths:
      - "dt-api/**"
      - ".github/workflows/dt-api.yml"
```

Listing 56: Branch selectors



DIG-189: Create endpoint for serving fault data in API #33

Merged lindtvedtsebastian merged 11 commits into `main` from `DIG-189` 26 days ago

Conversation 1 | Commits 11 | Checks 1 | Files changed 5

lindtvedtsebastian commented 26 days ago

Endpoints related to fetching faults data from CosmosDB

lindtvedtsebastian added 11 commits 26 days ago

```
fault-sticks: Rewrote cloud function to work consistently                    ✓ 770686d
fault-sticks: Removed unused imports                                         ✓ 1e81dbf
dt-api: Added endpoint for fetching all available fault collections            ecdb86b
dt-api: Added file for common code between endpoints                         ✓ ed41ddd
dt-api: Added general function for querying the faults container             ✓ d820d61
dt-api: Fixed typo in common.py                                                ebea97d
dt-api: Cleaned up faults.py and extracted duplicate code out into co… …     ✓ 42571fb
dt-api: Added endpoint for displaying all available fault collections          91885df
dt-api: Added function for querying the faults collection with parame… …       f8f767c
dt-api: Changes to /faults endpoint …                                        ✓ 85f865c
dt-api: Updated /faults/exists                                               ✓ b8f8075
```

lindtvedtsebastian requested re

All checks have passed
1 successful check
✓ Build and Deploy DT-API / build-and-deploy (push) … Details

salvadorbascunan approved these changes 26 days ago View changes

salvadorbascunan left a comment

Looks good. Nice job!

Figure 31: Pull request with CI status indicating successful CI deployment

The workflow is smart about cancelling redundant runs, and avoiding race-conditions. For example if we merge two pull requests one after the other, which results in the workflow being invoked twice. In the event of two simultanious pull requests, we only want to deploy the latest one. The developer merging the pull requests expects to see the changes from both PRs, but depending on how long the individual jobs takes to run, we may end up with only the changes from the first pull request being pushed to production, as which invocation of the workflow that ends up pushing its changes to production depends not on when it was invoked like we might expect, but on when the invocation finishes.

To avoid this problem of race conditions and redundant workflow runs, we use a concurrency group. This is a fancy concurrency primitive for synchronizing runs one after another, and optionally cancelling redundant invocations. A redundant workflow invocation being an invocation that will be immediately superseded by another. See listing 57.

```yaml
# Only allow one running workflow.
# Cancel the previous workflow when a new one is started.
# This avoids race-conditions for deployments.
concurrency:
  group: dt-api-production
  cancel-in-progress: true
```

Listing 57: Concurrency group

In listing 58 we see the individual jobs and steps in the workflow. In listed order, they are, the step for checking out the code from the relevant commit. A step for installing the correct version of the Python

```yaml
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    environment: dev
    steps:
      # Checkout the repo
      - uses: actions/checkout@v2

      # Setup python installation
      - name: Setup Python
        uses: actions/setup-python@v1
        with:
          python-version: ${{ env.PYTHON_VERSION }}

      # Install dependencies into virtual environment
      - name: Python install
        working-directory: ${{ env.WORKING_DIRECTORY }}
        run: |
          sudo apt install python${{ env.PYTHON_VERSION }}-venv
          python -m venv --copies antenv
          source antenv/bin/activate
          pip install setuptools
          pip install -r requirements.txt

      # Run PyTest
      - name: Run Tests
        working-directory: ${{ env.WORKING_DIRECTORY }}
        env:
          FLASK_ENV: "production"
          FLASK_APP: "app.factory:create_app()"
          ACCOUNT_URI: ${{ secrets.ACCOUNT_URI }}
          ACCOUNT_KEY: ${{ secrets.ACCOUNT_KEY }}
          AZURE_STORAGE_CONNECTION_STRING: ${{ secrets.AZURE_STORAGE_CONNECTION_STRING }}
        run: |
          source antenv/bin/activate
          pytest

      # Deploy to production when main updates
      - name: Deploy
        if: github.event_name == 'push'
        uses: azure/webapps-deploy@v2
        with:
          app-name: ${{ env.AZURE_WEBAPP_NAME }}
          package: ${{ env.WORKING_DIRECTORY }}
          slot-name: "Production"
          publish-profile: ${{ secrets.AZUREAPPSERVICE_PUBLISHPROFILE_7D9FAB1C04EB4DAEBA5AC430D48296D6 }}
```

Listing 58: Workflow jobs

interpreter. A step for setting up a virtual environment and installing all the needed dependencies. A step for running the test suite, which fails and aborts the job if any of the tests fail. And a step for finally deploying the new code to the production environment. You can also see that we use an *if* in the *deploy* step to only deploy to production if the invocation was caused by pushing or merging to main, not when working on a pull request. See figure 34 for an example of what this looks like in GitHub, after a workflow has finished.

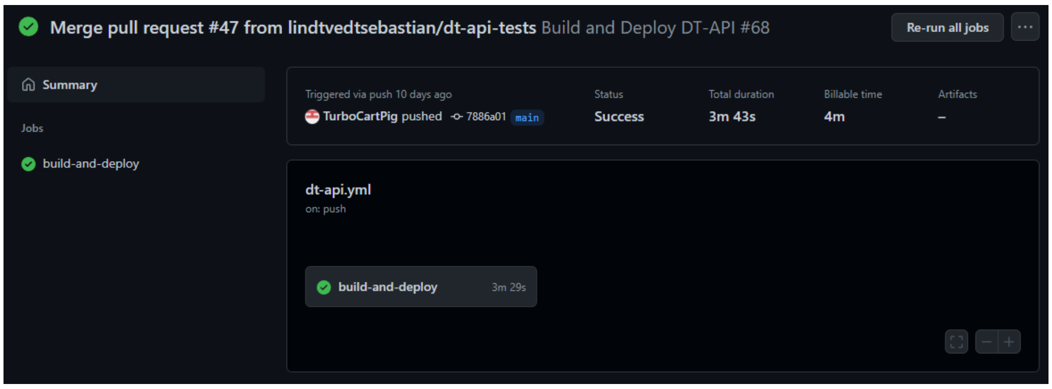
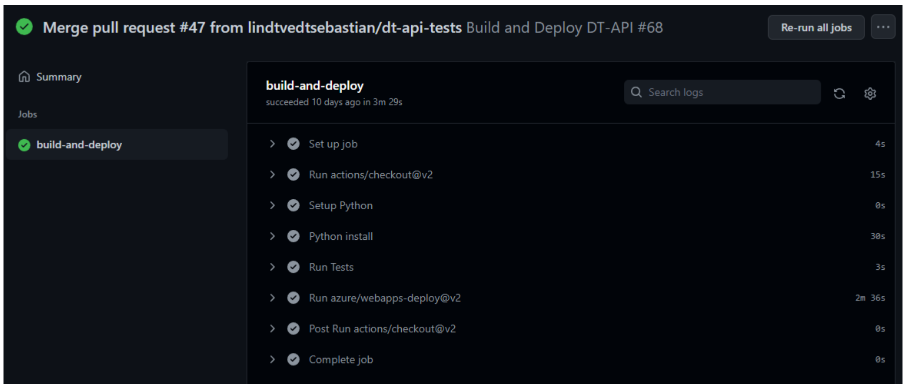
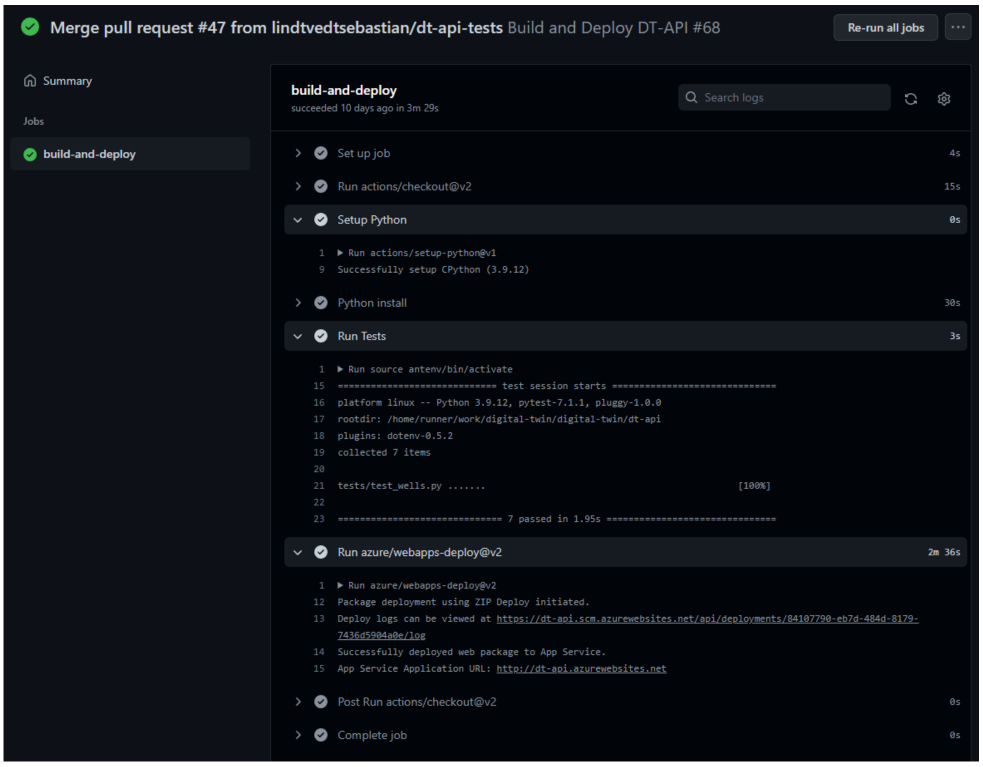

Merge pull request #47 from lindtvedtsebastian/dt-api-tests Build and Deploy DT-API #68
Re-run all jobs
Summary
Jobs
build-and-deploy
Triggered via push 10 days ago
TurboCartPig pushed 7886a01 main
Status
Success
Total duration
3m 43s
Billable time
4m
Artifacts
–
dt-api.yml
on: push
build-and-deploy 3m 29s

Figure 32: Overview of CI workflow



```
Merge pull request #47 from lindtvedtsebastian/dt-api-tests Build and Deploy DT-API #68      Re-run all jobs

Summary        build-and-deploy
               succeeded 10 days ago in 3m 29s
Jobs
build-and-deploy
               > Set up job                              4s
               > Run actions/checkout@v2                15s
               > Setup Python                            0s
               > Python install                         30s
               > Run Tests                               3s
               > Run azure/webapps-deploy@v2        2m 36s
               > Post Run actions/checkout@v2            0s
               > Complete job                            0s
```

Figure 33: Overview of CI jobs



```
Merge pull request #47 from lindtvedtsebastian/dt-api-tests Build and Deploy DT-API #68      Re-run all jobs

Summary        build-and-deploy
               succeeded 10 days ago in 3m 29s
Jobs
build-and-deploy
               > Set up job                              4s
               > Run actions/checkout@v2                15s
               v Setup Python                            0s
                 ▶ Run actions/setup-python@v1
                 Successfully setup CPython (3.9.12)
               > Python install                         30s
               v Run Tests                               3s
                 ▶ Run source antenv/bin/activate
                 ============================ test session starts =============================
                 platform linux -- Python 3.9.12, pytest-7.1.1, pluggy-1.0.0
                 rootdir: /home/runner/work/digital-twin/digital-twin/dt-api
                 plugins: dotenv-0.5.2
                 collected 7 items

                 tests/test_wells.py .......                                          [100%]

                 ============================= 7 passed in 1.95s ==============================
               v Run azure/webapps-deploy@v2        2m 36s
                 ▶ Run azure/webapps-deploy@v2
                 Package deployment using ZIP Deploy initiated.
                 Deploy logs can be viewed at https://dt-api.scm.azurewebsites.net/api/deployments/84107790-eb7d-484d-8179-
                 7436d5904a0e/log
                 Successfully deployed web package to App Service.
                 App Service Application URL: http://dt-api.azurewebsites.net
               > Post Run actions/checkout@v2            0s
               > Complete job                            0s
```

Figure 34: Overview of CI jobs expanded

9 Discussion

We will discuss our development process as a whole, and dive deeper into parts like the initial project plan, our technology choices, how scrum worked for us, how we worked as a group, how our choices affected the final solution, what we think of the final product, and more.

9.1 Development Process

During our initial planning phase, we decided that writing was something we were going to do concurrently with the development of the project. Although, we did write some parts, we see in hindsight that even more writing would have been beneficial. The development work became a much bigger task than initially planned. As explained in earlier parts of this document, our decision to create a reusable, more general 'pipeline' for digital twins had its cost. Considering that a shifted weight on writing at the cost of reduced focus on development would ultimately have led to a weaker final product, we believe that the way things turned out was for the better.

The last month of the project period was very heavily focused on writing. Even though we had a much bigger writing job in the final phase, we believe that the thorough development work that we conducted laid a solid foundation for the writing phase, making this part easier than if it had been done earlier in the project work.

We initially wanted to perform the development work in an even more sequential manner than how it actually turned out. We estimated that different components of the project would be finalized much quicker than they were in reality. The result of the more iterative approach we ended up with was in the end a much more well thought through solution. The reason for this was that new issues arose during development that we went back and fixed. One example of this was with the faults parser. Here we went back and customized how data was converted into JSON, this allowed the JSON to be much more sensible in regard to how the web-app and CosmosDB had been set up.

9.1.1 Scrum

Scrum served as a great framework for keeping the development work structured. The routine Scrum provided, with clearly defined events occurring every single sprint served as a great way to keep the up the progress of the project. Although the specific meetings, such as retrospective, were very handy in the start of the project, we do feel that their usefulness declined over time. Towards the end of the project the sprint retrospective, for example, yielded no useful information anymore, as all the issued we had in earlier retrospectives had been resolved. If the project were to keep on for much longer, we should have considered rethinking the frequencies of the different scrum meetings. Though, for the project we had, the frequencies we chose were appropriate.

9.1.2 Meetings

As touched upon in the previous section, we followed our initial plan of meetings defined in the project planning phase. The frequency of these meetings was fine. The two weekly stand-up meetings were more than plenty to keep in touch within the team, the reason for this is that we communicate frequently within the team on platforms such as Discord in addition to these meetings. We established a Slack channel with the client early on. This communication channel was a great way to communicate outside of the meeting we had every two weeks. We met with our supervisor almost every two weeks. These meetings were an excellent way to receive feedback, both on our writing, but also on our the overall project progress. Overall, we would say that the meeting structure we followed in this project is sensible to adhere to in further work as well.

9.1.3 Atlassian suite

We utilized the Atlassian suite of software for almost all project process related issues, except for version control and communication. It was nice to have a platform that served several purposes, as everything worked together nicely. In previous projects we have used the issue tracker built into GitHub, this allowed the issues to be a bit more connected to the code in our opinion. The advantage of Jira was that we could link project pages and time tracking directly to specific issues. Overall, the advantage of the integration of tools was worth the trade-off between less connection between issues and code.

Project pages

Atlassian has a tool for writing project pages called Confluence. Confluence allows for live collaboration on documents, as well as integrating tightly with Atlassian's other products. It was nice to have all notes related to meetings located right next to Jira, and be able to reference relevant issues. An alternative here would have been to use tools such as Google Docs, where similar cloud collaboration features are available, or GitHub Wiki, which provides an alternative way of creating a knowledge base and documentation. Overall, we think Confluence was a great tool for our purpose, as we felt like it was easy to organize our common project notes using this tool.

Time tracking

In previous projects, time tracking has been a consistent issue. The different group members have all utilized different tools for time tracking, making the final phase of combining all tracking together to a common format a real struggle. Tempo was a nice tool for tracking time. It had extensions for editors such as VSCode, which allows for easy time tracking. Tempo also integrates nicely with Jira as a whole, making time tracking of specific issues a breeze. Overall, we were happy with Tempo.

9.2 Test Driven Development

This was the first time the group had worked with a test driven development workflow. It was a steep learning curve to get the hang of the test-first, red/green/refactor cycle. Testing was difficult in the beginning, but became easier as the project progressed. These difficulties lead to some situations where functionality was developed before the tests. The habit of testing first, implementing second grew on the group members throughout the project period. The value of TDD became more apparent as the backlog of tests grew.

9.3 Technical Design

9.3.1 Azure

This project came with a strong recommendation to use Azure as the cloud vendor. Even so, we think it is important to give Azure some due criticism in this part of the paper, as Azure has been a source of frustration throughout the project.

During the project, Azure suffered a series of outages and service disruptions that left us unable to deploy any of our Python services for two days. The source being a misconfiguration on Microsoft's part, that made Onyx attempt to build our services with the wrong version of the Python interpreter[18]. This is in turn caused by Azure not being very flexible when it comes to the exact version of Python a project uses, only allowing the choice between a few select versions, none of which are the latest one. You are also not allowed to select a patch version, only major and minor versions.

Microsoft's way of structuring their documentation has caused some issues throughout the development process. Microsoft does have a massive library of documentation, but the problem is that it is difficult to navigate at times. The documentation sometimes repeats itself, while failing to mention some topics. An example of this is how to query for an exact CosmosDB document based on captured expressions in Azure Function bindings. The upside of Microsoft's documentation is their SDKs, which provide a solid interface for interacting with their different services, such as CosmosDB, Blob Storage, etc.

9.3.2 REST API

Flask is a minimal framework. We were a bit worried that there might be some features required for the REST API that Flask would not be able to satisfy, but these concerns proved themselves unfounded. Flask provided a quick and straightforward way to set up different endpoints. The trade-off between simplicity and features was perfectly balanced in our case, as we did not feel limited by Flask. We were able to implement all the necessary functionality for our REST API using Flask, and we would gladly use Flask again in further projects.

9.3.3 Persistent storage

CosmosDB proved to be a great persistent storage solution. The ease of querying the data using SQL like syntax was really handy. The SQL way of thinking is something we as a group has had a lot of previous experience with, and the opportunity to utilize this knowledge was a time saver in the development work. The option to run CosmosDB in a serverless manner suited our project perfectly, as queries against the database happened at non-scheduled, unpredictable times.

9.3.4 Surfaces

Python proved to be a good choice for us when it came to the implementation of the surfaces service. It gave us the flexibility of writing both the parser and the geometry generation in the same language. Because of Python's mature ecosystem, we had good alternatives for libraries to solve our problems.

Choosing Open3D to generate glTF files from point clouds proved to be a good choice. Open3D is a well established library with good documentation. The few issues we faced with the geometry, as mentioned in the implementation, were manageable. Early in the development process, we briefly considered developing our own geometry generator. Although this sounded interesting, we quickly decided to use a library, in this case Open3D, which we believe saved us valuable development time.

In our case, choosing to utilize Open3D for our geometry generation is what determined how we needed to parse the surface data. As explained in the implementation, Open3D needs a NumPy array. This can either be received as an object, or as a file. How Azure Functions work influenced our choice of creating the parser in Python. As explained in the implementation, we tested parsing the data in Haskell, which would have been an interesting subject to explore further

9.3.5 Horizons

Python proved to be a good choice for the implementation parts we performed on the horizon files. It gave us great flexibility to have the point cloud extraction, parser, and geometry generator be using the same technology. GeoPandas, together with Open3D, allowed us to cover the requirements well. The horizons were not a part of the final product because Glex did not see any value in including them.

9.3.6 Well logs

As far as the microservice implementation goes, we think utilizing Python with the libraries that we did was a good choice, and we would not have done anything differently.

When it comes to visualization of the wellbores and well-logs, things could have been better. The visualization of the wellbores themselves is correct, but it is a bit hard to see that we have implemented full wellbore geometry generation when the wellbore are so straight, however we felt like the scaling and transformations we applied were still correct in the context of visualization of the well-logs on the wellbore.

The visualization of the well-logs did not go quite to plan. Visualizing well-logs in 3D proved to be a much harder, and more importantly time-consuming task than originally estimated. The choice of using ThreeJS over Unity might not have been the best. ThreeJS does not have any built-in functionality for building graphs, and although we made a good attempt at doing this manually, we eventually had to down prioritize the 3D graphing functionality in favor of a more complete 2D graphing and visualization functionality.

9.3.7 Faults

Python was a nice language for the faults parser. The only drawback to Python is the relatively weak type system in comparison to languages such as Rust or Haskell. The code was written in a way that attempted to recreate some of the benefits of languages with a better type system, with classes serving as different data types. This worked to some degree, although not very optimally. A better alternative would perhaps have been to utilize a language such as Haskell. Since Haskell has a strong type system, it would have resulted in cleaner tests in our opinion. We believe that a language such as Haskell or Rust would have been a better fit for this service.

9.3.8 Frontend

Our choice to develop a web-app to visualize the digital twin proved to be one of the more challenging parts of the project. Early in the development process we discussed whether we would use a web-based frontend framework like React in combination with ThreeJS, or Unity, a standalone game engine. Our discussing revolved mainly around the fact that our client already used Unity in their preexisting projects. Here, we will discuss the strength and weaknesses we found in using a web-based approach with React Three Fiber, among other tools, compared to Unity.

Unity vs React Three Fiber

Using React Three Fiber meant that we could use React components together with ThreeJS. This was beneficial for us because we had some previous experience with React from other projects. Loading the generated geometry was not an issue and the performance proved to be acceptable. The issues began when we tried to implement the plotting requirements alongside the well trajectory. This requirement meant that certain well data needed to be visualized in the 3D scene. We tried using Victory, but the implementation ended up being a lot more time-consuming than originally estimated. Even though we did implement a low fidelity prototype of this, we ended up focusing on improving the 2D graphs to display the well data, which we think was the right call in the end.

It is regarding the 3D graph plotting requirement, we think Unity might have worked out better, considering Unity has built-in support for this. Other advantages of using Unity would have been its interactivity, inspectability, and the write-build-debug cycle being faster, as these were low points for React Three Fiber. We are also not sure how big of an advantage a purely web based solution has over using Unity's Webgl backend, even if we know the latter has disadvantages and constraints as well, like not being able to mix well with HTML.

Frontend Conclusion

Having discussed these comparisons and seen the results we achieved, we think that our web based approach had its upsides, but making it work even better would require a significant investment in order to get 3D graph functionality comparable with what Unity already has. On the other hand, using web technologies made the frontend much more accessible than we believe a Unity based approach would have been.

9.4 Product

The presentation layer of the final product is the component we are the least satisfied with. The reason the web-app turned out suboptimal was due to the focus on the general pipeline, as opposed to the more specific digital twin. Our choice of creating reusable components instead of custom tailoring the service to suit the Smeaheia dataset is apparent.

The overall quality of the architecture as a whole however is of a much higher quality than it would have been, had we gone for the specific approach. All the different microservices are written in a way that easily enables a data manager to create new digital twins within ocean spaces, containing data from a totally different area in the world. This pipeline is really what our project has been all about, and we are proud of how it turned out.

The time required to create new digital twins have been significantly reduced by courtesy of our pipeline. We are satisfied with how our architecture easily facilitates further expansion of new services and features. The architecture is highly scalable, modular, and ready for additional microservices.

9.4.1 Revisiting Project Result Goals

In this section, we will revisit the result goals we set in the Introduction. We will discuss to what degree these goals were met, and what we could have done different in the situations where a certain goal was not met.

- **A general processing solution that facilitates generation of geological digital twins from data.**

  This goal was met through the solution we developed. We implemented this through the use-case of a data manager, where a data file is uploaded to blob storage and subsequently triggers the appropriate service for that data type. Once finished, the result can be visualization in the web-app.

- **A digital twin, generated through the developed solution using the Smeaheia dataset and made**

accessible through a web interface.

The pipeline proved itself more than capable of processing the Smeaheia dataset. The resulting visualization is available through the web interface we developed. The web app runs on several different devices, including mobile. The performance is generally speaking pretty good.

- **The solution should be able to process different data types, such as Surfaces, Horizons, Faults and Well logs.**

  Our goal of creating services to handle different data types from the dataset was accomplished. The solution provides a set of services for each data type that runs when new files are added to blob storage. The cloud service part of Horizons was not implemented because Glex did not see the value of visualizing this in the web-app.

- **The solution should be able to support real-time data.**

  Mocking of real-time data was implemented and visualized in the form of a graph in the UI panel. The solution supports live real-time data, if this ever comes into existence.

- **The solution should have plotting and visualization functionality to display processed data in graphs, both in UI panel and scene.**

  Plotting functionality was implemented, although only partially in the 3d scene. The UI panels facilitates access to *Victory*, a powerful graphing library for React, which was utilized to create the necessary graphs. The 3D plotting functionality, as touched upon earlier in the well logs section of the Discussion, was implemented in a limited manner.

- **The solution must be modular and expandable.**

  The different parts of the solution are modular, which makes the solution as a whole expandable. Adding new services to handle more types of data is only a matter of creating the actual microservices, as the solution itself is readily expandable.

9.5 Group collaboration

As a group, we are happy with our collaboration in this project. We have collaborated on multiple projects throughout our time here at NTNU, so we are well-aware of our level of ambition. We have also established a certain level of trust within the group, which allows us to collaborate very effectively.

Communication

For the majority of our time here at NTNU, we have had to take COVID-19 into consideration. This has created a more digital approach to collaboration and general school work. Discord has been our main communication tool during our time here at NTNU, and was our main communication tool in this project as well. We created a dedicated channel for this project and shared thoughts, problems, and possible solutions there.

Group Work

At the beginning of the project period, we established some ground rules regarding workdays. Every weekday, all group members would be required to be available on Discord from 10 am to 6 pm. This was to establish an environment with workday continuity so that we could rely on each other. Having our two weekly stand-up meetings allowed us to stay up-to-date on each other's progress as well. This enabled us to quickly discuss and solve any issues within the group or the development process.

9.6 Time allocation

In regard to time spent during the project period, we utilized labels to categorize different types of activities. As can be seen in Figure 35, we spent more than 50% of our time doing actual development work. Overall, we are happy with the time and effort invested into the project, and we feel like this is well reflected in both the thesis and final product. Table 4 showcases how many hours went into each activity.

Type of activity	**Hours spent**
Client meetings	34.42
Supervisor meetings	39.93
Internal meetings	227.93
Thesis writing	432.38
Development work	937.45

Table 4: Overview of time spent on various activities

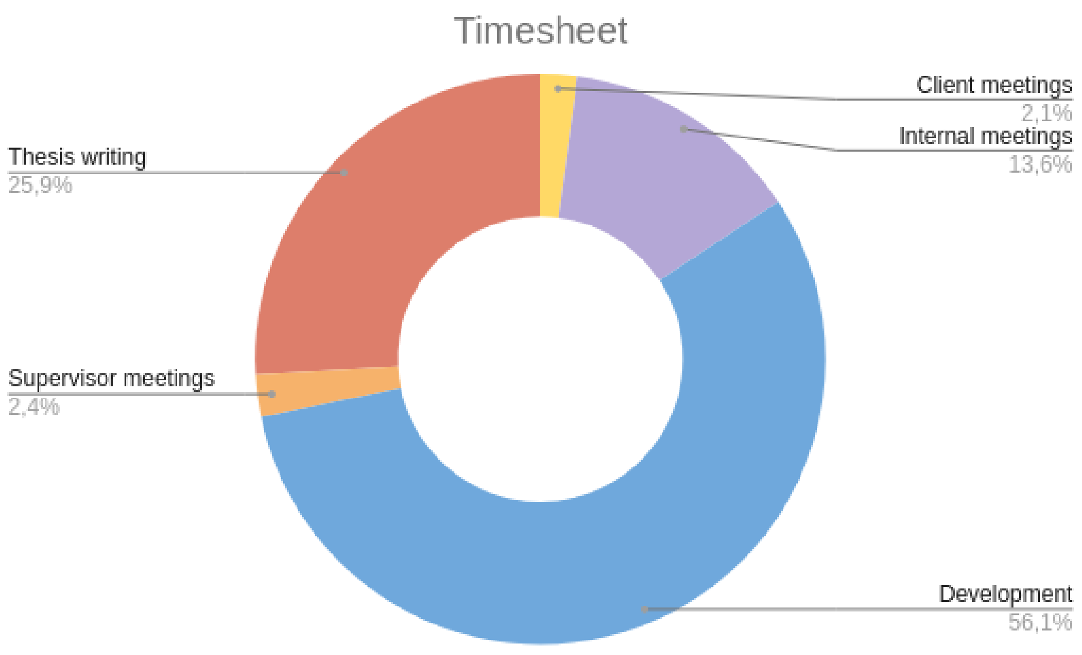


Figure 35: Overview of percentage use of time

10 Conclusion

We will write a small summary of this project and how the process was for us, and how the product looks and works from our perspective. We will also mention some possibilities regarding future development of this project

10.1 Process

Our way of structuring and working on this project was a success. The structure provided by Scrum helped keeping the progress consistent. Both the product and the thesis were finalized in time. The group members were able to conduct work on consistent levels throughout the entire project period, resulting in a satisfactory amount of time invested in the project as a whole. We conducted the entire project digitally, as the group have grown accustomed to this workflow after several years of Covid-19 pandemic.

10.2 Product

We are satisfied with the resulting product. The purpose of the finalized solution is to provide Glex with a simplified workflow when creating digital twins. Due to time constraints, we have not made a user interface for the data manager role, meaning the solution assumes technical experience and knowledge with regard to Azure's data management tools, which the client has. The solution is highly scalable, and readily facilitated for further service integration and expansion.

10.3 Future Development

Although the work that has been done creates a solid foundation, there are several potential additions that would increase the value of the solution if it were to be developed further:

- Create a service/interface for Data managers, allowing them to upload, modify, and delete data related to the digital twins.
- Further development on the user interface of the web-app. Our focus on this project has been the system architecture and cloud service, so further development on the UI would benefit the solution as a whole.
- Explore the discoveries made regarding mobile devices. Glex informed us that they tested the web-app on their phones, and achieved good performance. This opens up a whole specter of possibilities for web-based digital twins in the future.

10.4 Final words

During our time working on our bachelor’s thesis, we have acquired new knowledge within fields such as cloud, digital twins, project management and overall software development. It has been both exciting and a great learning experience to work on such a big project. Being able to contribute within the field of CCS has felt rewarding, knowing we are contributing to a greener tomorrow.

It has been exciting to create something with potential impact in the relatively new field of digital twins. By experimenting with new technologies such as ThreeJS, we are exploring a new approach within the domain of digital twins as a whole. In addition to this, discovering the possibilities the web-based approach has for mobile devices, opens up a whole new way of displaying and utilizing digital twins.

In conclusion, we are very satisfied with the project as a whole. Due to the continuous efforts of the group, we were able to finalize both a thesis and a solution we are proud of.

References

[1] *About Azure Functions*. Microsoft. URL: https://docs.microsoft.com/en-us/azure/azure-functions/ (visited on 18th May 2022).

[2] *About NumPy*. NumPy. URL: https://numpy.org/about/ (visited on 18th May 2022).

[3] *About Pandas*. Pandas. URL: https://pandas.pydata.org/about/ (visited on 18th May 2022).

[4] *About VictoryChart*. Victory. URL: https://formidable.com/open-source/victory/about (visited on 18th May 2022).

[5] *Application Factories*. Flask. URL: https://flask.palletsprojects.com/en/2.1.x/patterns/appfactories/ (visited on 14th May 2022).

[6] Atlassian. *Atlassian Jira*. Atlassian. URL: https://www.atlassian.com/software/jira (visited on 4th May 2022).

[7] *Ball Pivoting Algorithm*. Brett Rapponotti, Michael Snowden and Allen Zeng. URL: https://cs184team.github.io/cs184-final/writeup.html (visited on 19th May 2022).

[8] *Command Line Interface*. Flask. URL: https://flask.palletsprojects.com/en/2.1.x/cli/ (visited on 14th May 2022).

[9] *Continuous Deployement*. IBM. URL: https://www.ibm.com/cloud/learn/continuous-deployment (visited on 1st May 2022).

[10] *Continuous Itegration*. Atlassian. URL: https://www.atlassian.com/continuous-delivery/continuous-integration (visited on 1st May 2022).

[11] P.J. Eby. *PEP 3333 Python Web Server Gateway Interface*. Python. URL: https://peps.python.org/pep-3333/ (visited on 15th May 2022).

[12] Marco Macchi Elisa Negri Luca Fumagalli. *LaTeX: A review of the roles of Digital Twin in CPS-based production systems*. 2017.

[13] Gassnova. *The world comes together to tackle climate change*. Gassnova. URL: https://gassnova.no/en/why-ccs (visited on 26th Jan. 2022).

[14] *Glex Energy*. Glex. URL: https://glex.no/product (visited on 18th May 2022).

[15] *Gunicorn*. Gunicorn. URL: https://gunicorn.org/ (visited on 15th May 2022).

[16] Sakshi Gupta. *What is SQL?* Springboard. URL: https://www.springboard.com/blog/data-analytics/what-is-sql/ (visited on 8th May 2022).

[17] IBM. *What is a digital twin*. IBM. URL: https://www.ibm.com/topics/what-is-a-digital-twin (visited on 26th Jan. 2022).

[18] *Installing python 3.9 fails as of 2022-04-06 20:00 EST*. Microsoft. URL: https://github.com/microsoft/Oryx/issues/1330#issuecomment-1091739027 (visited on 28th Apr. 2022).

[19] *Intro Faults*. Geology Page. URL: https://www.geologypage.com/2017/10/what-is-a-geologic-fault.html (visited on 14th Jan. 2022).

[20] *Large Applications as Packages*. Flask. URL: https://flask.palletsprojects.com/en/2.1.x/patterns/packages/ (visited on 14th May 2022).

[21] Microsoft. *Azure databases*. Microsoft. URL: https://azure.microsoft.com/en-ca/product-categories/databases/ (visited on 8th May 2022).

[22] Microsoft. *Azure DDoS Protection Standard overview*. Microsoft. URL: https://docs.microsoft.com/en-us/azure/ddos-protection/ddos-protection-overview (visited on 25th Apr. 2022).

[23] Microsoft. *Azure Function Apps*. Microsoft. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview (visited on 8th May 2022).

[24] Microsoft. *Gateway Routing pattern*. Microsoft. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/gateway-routing (visited on 5th May 2022).

[25] Microsoft. *Manage Azure resource groups by using the Azure portal*. Microsoft. URL: https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal (visited on 22nd Apr. 2022).

[26] Microsoft. *Materialized Views*. Microsoft. URL: https://docs.microsoft.com/en-us/azure/data-explorer/kusto/management/materialized-views/materialized-view-overview (visited on 5th May 2022).

[27] Microsoft. *Microservice Architecture*. Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture (visited on 5th May 2022).

[28] Microsoft. *N-layer Architecture*. Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures#traditional-n-layer-architecture-applications (visited on 6th May 2022).

[29] *Modular Applications with Blueprints*. Flask. URL: https://flask.palletsprojects.com/en/2.1.x/blueprints/ (visited on 14th May 2022).

[30] MongoDB. *Types of NoSQL databases*. MongoDB. URL: https://www.mongodb.com/scale/types-of-nosql-databases (visited on 8th May 2022).

[31] CO2 Team NPD. *CO2 Atlas for The Norwegian Continental Shelf*. Ed. by Eva K. Halland & Jasminka Mujezinovi & Fridtjof Riis. The Norwegian Petroleum Directorate. 2014. URL: https://www.npd.no/en/facts/publications/co2-atlases/co2-atlas-for-the-norwegian-continental-shelf/ (visited on 18th Jan. 2022).

[32] Nrwl. *What is a Monorepo?* Nrwl. URL: https://monorepo.tools/#what-is-a-monorepo (visited on 7th May 2022).

[33] *Oilfield Glossary - chronostratigraphy*. Schlumberger - Oilfield Glossary. URL: https://glossary.oilfield.slb.com/en/terms/c/chronostratigraphy (visited on 14th Jan. 2022).

[34] *Oilfield Glossary - lithostratigraphy*. Schlumberger. URL: https://glossary.oilfield.slb.com/en/terms/l/lithostratigraphy (visited on 14th Jan. 2022).

[35] *Oilfield Glossary - strata*. Schlumberger - Oilfield Glossary. URL: https://glossary.oilfield.slb.com/en/terms/s/strata (visited on 14th Jan. 2022).

[36] *Oilfield Glossary - well log*. Schlumberger. URL: https://glossary.oilfield.slb.com/en/terms/w/well_log (visited on 14th Jan. 2022).

[37] *Oilfield Glossary - wellbore*. Schlumberger. URL: https://glossary.oilfield.slb.com/en/terms/w/wellbore (visited on 14th Jan. 2022).

[38] Open3D. *Open3D Documentation*. Open3D. URL: http://www.open3d.org/docs/release/introduction.html (visited on 9th May 2022).

[39] Ministry of Petroleum and Energy. *Announcement of Longship by Norwegian Government*. Norwegian Government. URL: https://www.regjeringen.no/no/dokumentarkiv/regjeringen-solberg/aktuelt-regjeringen-solberg/smk/pressemeldinger/2020/regjeringa-lanserer-langskip-for-fangst-og-lagring-av-co2-i-noreg/id2765288/ (visited on 26th Feb. 2022).

[40] *Poisson Surface Reconstruction*. Michael Kazhdan, Matthew Bolitho and Hugues Hoppe. URL: https://hhoppe.com/poissonrecon.pdf (visited on 19th May 2022).

[41] Pygltflib. *Pygltflib Documentation*. Pygltflib. URL: https://gitlab.com/dodgyville/pygltflib (visited on 9th May 2022).

[42] *Smeaheia Dataset*. Gassnova & Equinor. URL: https://co2datashare.org/dataset/smeaheia-dataset (visited on 18th Jan. 2022).

[43] *Smeaheia Dataset*. Smeaheia. URL: https://co2datashare.org/dataset/smeaheia-dataset (visited on 15th May 2022).

[44] *Smeaheia Dataset License*. Gassnova & Equinor. URL: https://co2datashare.org/view/license/26af9426-203f-4993-9d41-2e1bf191ceaf (visited on 18th Jan. 2022).

[45] Tech27. *What are point clouds?* Tech27. URL: https://tech27.com/resources/point-clouds/ (visited on 5th Feb. 2022).

[46] *Template Project NTNU*. Jon Arnt Kårstad. URL: https://no.overleaf.com/latex/templates/template-project-ntnu/zjystqvqztpg (visited on 17th May 2022).

[47] *Testing Flask Applications*. Flask. URL: https://flask.palletsprojects.com/en/2.1.x/testing/ (visited on 15th May 2022).

[48] *Testing Overview*. Facebook. URL: https://reactjs.org/docs/testing.html#tools (visited on 15th Apr. 2022).

[49] ThreeJS. *Loading 3D models in ThreeJS?* ThreeJS. URL: https://threejs.org/docs/index.html#manual/en/introduction/Loading-3D-models (visited on 2nd May 2022).

[50] ThreeJS. *ThreeJS*. ThreeJS. URL: https://threejs.org/ (visited on 7th May 2022).

[51] *Tox*. Tox. URL: https://github.com/tox-dev/tox (visited on 2nd May 2022).

[52] Unity. *Unity*. Unity. URL: https://unity.com/solutions/game (visited on 5th July 2022).

[53] USGS. *What is a fault?* USGS. URL: https://www.usgs.gov/faqs/what-fault-and-what-are-different-types?qt-news_science_products=0#qt-news_science_products (visited on 1st Mar. 2022).

[54] Erlend Viggen. *DLIS Files*. URL: https://erlend-viggen.no/dlis-files/ (visited on 11th Jan. 2022).

[55] *WebGL*. Khronos. URL: https://www.khronos.org/webgl/ (visited on 20th May 2022).

[56] *What are Shape files?* ArcMap. URL: https://desktop.arcgis.com/en/arcmap/latest/manage-data/shapefiles/what-is-a-shapefile.htm (visited on 15th May 2022).

[57] *What is a glTF file?* Khronos. URL: https://www.khronos.org/gltf/ (visited on 19th May 2022).

[58] *What is a surface in geology?* Mindat. URL: https://www.mindat.org/glossary/surface_geology (visited on 15th May 2022).

[59] *What is CCS*. Equinor. URL: https://www.equinor.com/energy/carbon-capture-utilisation-and-storage?e93a3fa409d4=0 (visited on 19th May 2022).

[60] *What is GeoPandas?* GeoPandas. URL: https://geopandas.org/en/stable/ (visited on 15th May 2022).

[61] *What is lithological*. Schlumberger. Oilfield Glossary. URL: https://glossary.oilfield.slb.com/en/terms/l/lithologic (visited on 19th May 2022).

[62] *What is Test Driven Development*. IBM. URL: https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/ (visited on 19th May 2022).

[63] *What is TVD*. Oilfield Glossary. URL: https://glossary.oilfield.slb.com/en/terms/t/true_vertical_depth (visited on 19th May 2022).

[64] *What is TVDKB*. Oilfield Glossary. URL: https://help.dugeo.com/m/faq/l/167077-height-datums-and-abbreviations (visited on 19th May 2022).

[65] *What is TVDSS*. Research Gate. URL: https://www.researchgate.net/figure/Seismic-depth-slices-at-2304-m-TVDSS-True-Vertical-Depth-Sub-Sea-showing-faults-in-the_fig15_304066028 (visited on 19th May 2022).

[66] Wikipedia. *What is a unit test*. Wikipedia. URL: https://en.wikipedia.org/wiki/Unit_testing (visited on 28th Jan. 2022).

[67] Wikipedia. *What is an integration test*. Wikipedia. URL: https://en.wikipedia.org/wiki/Integration_testing (visited on 28th Jan. 2022).

Appendix

A Project Plan

A Background

Glex is a company mainly based in Bergen, with ambitions to expand their offices to Oslo. Their focus areas are a unique mix of game development, geology and geospatial technology. Glex focuses on building solutions to solve complex problems for some of the most demanding customers in the world.

Glex main building blocks consist of Oil & Gas Exploration, Deep Sea Minerals, Renewable Energy, and Carbon Capture & Storage. With this unique toolkit, Glex is able to give their customers an end-to-end software for managing and staying up to date with their exploration and/or renewable portfolios.

Currently, Glex has a couple of customers using their main software. This is within the Oil & Gas Exploration industry. For the time being, this is Glex's main area of focus as they are in contact with more potential customers in this segment.

For this project, Glex wants us to develop a digital twin of the Smeaheia Carbon Storage site. This will entail having a graphical user interface with the geometry that the digital twin represents, and a user interface displaying the data. We will be exploring new ways of visualizing the data on the frontend, with ThreeJS running on top of WebGL in the browser. So far we have planned to serve the data processed on the backend with a variety of different microservices.

B Domain

There is a growing need to be able to safely dispose of excess CO_2 in the world today. This is especially important in the Oil and Gas industry. Carbon capture and storage, CCS, is a solution to this. CCS is a process where excess CO_2 is captured at a production site and transported to a storage site where the CO_2 is stored permanently. Interest in this area has increased following The Norwegian Government's announcement of project Longship [39], Equinor's project Northern Lights, and Blue Barents CSS project. All projects dedicated to realizing the CSS potential present on The Norwegian Continental Shelf.

According to *CO2 Atlas*[31]:

> *Depending on their specific geological properties, several types of geological formations can be used to store CO_2. In the North Sea Basin, the greatest potential capacity for CO_2 storage will be in deep saline-water saturated formations or in depleted oil and gas fields.*
>
> *CO_2 will be injected and stored as a supercritical fluid. It then migrates through the interconnected pore spaces in the rock, just like other fluids (water, oil, gas).*

And according to *Gassnova*[13]

> *We need to increase the efficiency of our energy consumption and promote sources of non-fossil fuel energy. Yet despite advances in these areas and technological leaps forward in renewables such as solar, wind and hydropower, the world is in danger of failing to meet those targets. In turn, we are in danger of failing our responsibility to future generations. Energy intensive industry accounts for 25% of global CO2 emissions and cannot go down to zero without CCS.*

An area with potential for CO_2 storage on the Norwegian Continental Shelf, is the Smeaheia area, located on the Hordaland Platform in the North Sea near Mongstad. This area has been evaluated by both Gassnova and Equinor, and together they have released and open-sourced a dataset containing subsurface data, reports, and geomodels, for the purpose of encouraging research and learning around CCS. [42]

Before reading the rest of this paper, it is important to also know what we mean by a digital twin.

According to *IBM: What is a digital twin?*[17]

> *A digital twin is a virtual model designed to accurately reflect a physical object. The object being studied for example, a wind turbine is outfitted with various sensors related to vital areas of functionality. These sensors produce data about different aspects of the physical objects performance, such as energy output, temperature, weather conditions and more. This data is then relayed to a processing system and applied to the digital copy.*

C Project Goals

C.1 Result Goals

- Create a digital twin of the Smeaheia carbon storage site based on historical and real-time data, from the processing of raw data all the way to visualization.
- Explore a new workflow within visualizing digital twins in web browsers.
- Create a user interface for the digital twin that runs in the browser.
- The digital twin will be accurate enough to represent the physical area for further research.

C.2 Effect Goals

Qualitative goals:

- Gain experience about the advantages and disadvantages of our technology choices versus those currently being utilized by Glex.

- Increase the availability of high fidelity visualizations.

Quantitative goals:

- Reduce the time it takes to show implementations to clients.
- Reduce the threshold for clients to run the web application
- Lower hardware requirements compared to previous similar solutions

C.3 Learning Goals

- Improve our knowledge within Scrum based development.
- Improve our knowledge within Test Driven Development (TDD).
- Improve our knowledge within git best practices, continuous development and continuous deployment.
- Learn and apply Three.js, a WebGL library for 3d graphics on the web.
- Learn more about microservices.
- Learn about complex cloud architectures and improve our knowledge of cloud in general.
- Improve our knowledge about project management in general, and specifically learn Jira, a cloud collaboration tool.

D Delimitation

- The software will only feature a specific geographic area.
- The visualization will be limited by the available data in the Smeaheia dataset, and those supplied by Glex.
- Only two wells will be visualized.
- The full-scale CCS value chain can be defined as 3 sub-systems; Capture, Transport, and subsurface Storage, including injection location of the CO_2. The bachelor thesis will focus only on the sub-system Subsurface Storage.
- The software's geographic data will not be updated live.
- The software is designed for desktop devices, not mobile ones, like phones and tablets.

E Framework

Together with Glex we have formulated the following framework requirements.

- The web app will work on the latest version of Chrome, Edge, and Firefox respectively.
- The back-end services will be deployable to Microsoft Azure.

F Case Description

This project will be based on a subset of the existing historic data available, and real-time data. The digital twin will feature a 3D visualization of the area, including the 32/4-1 well trajectory, 2D and 3D seismic, horizons, surfaces, fault sticks and well data such as pressure and porosity. This data will be supplied mainly from the Smeaheia dataset, and in addition it will be supplemented with data from Glex.

The digital twin should be made available to the end-users via a web-based interactive application, where users can access historical information, operational information, and key performance indicators, by interacting with different parts of the visualized 3D environment.

The project as a whole will showcase the potential that are enabled by creating a digital twin for carbon storage. Our end users will be any and all actors within the CCS segment on the Norwegian Continental Shelf today. e.g. Shell, Equinor, Horisont Energi, Northern Lights JV, Vår Energi.

Glex have divided the project into several separate requirements for us to complete in sequence.

F.1 Requirement 1: Historical data

In order to create the 3D visualization as specified above, data needs to be fetched and extracted from multiple sources.

- 3D Cube outlines for TNE01 and GN1101. Source: Excel sheet provided by Glex.
- Interpreted surfaces from 3D seismic. Surfaces to be in depth. Source: Files provided.
- Fault sticks (depth-conversion required?). Source: Files provided.
- Well locations and Trajectory surveys for wellbores 32/2-1, 32/4-1 and 32/4-1 T2 (technical side-track). Source: Excel sheet provided.
- Composite log data for all wellbores. Source: LAS files provided.
- Lithostratigraphy and Chronostratigraphy data. Source: Excel sheet provided.
- Formation Pressure and Core Porosity/Permeability data. Source: Excel sheet provided.

F.2 Requirement 2: Plotting and Visualization

The ability to visualize the 3D cube outlines, surfaces, fault sticks and wellbores.

Data to be visualized alongside the well trajectory in 3D:

- Composite Log Data.
- Lithostratigraphy.
- Chronostratigraphy.
- Porosity/Permeability.
- Formation Pressure.

Data to be visualized using plots and graphs (Lithostratigraphy data should be indicated in these plots):

- Formation Pressure vs Depth (MD and TVDSS). [Most important]
- Porosity vs Permeability.
- Porosity vs Depth (MD and TVDSS).
- Permeability vs Depth (MD and TVDSS).

F.3 Requirement 3: Real-time data

In order to make a proper digital twin, real-time data needs to be fetched and integrated into the environment. These pieces of data are much harder to source and might need to be mocked or faked for this project, as opposed to basing ourselves on real data the way we are doing for the above requirements.

These are the most critical pieces of real-time data:

- Well pressure.
- Flow rate.

G Project Organization

G.1 Responsibilities and Roles

G.1.1 Glex

Glex is the client and product owner, where our contact person is Jørgen Engen Napstad. We will have weekly meetings where we will update them regarding the progress, as well as ask questions. We have established a Slack channel where we can communicate frequently.

G.1.2 Supervisor

Our supervisor for this project is Tom Røise. Initially we have established weekly meetings and will adjust according to the need for supervision.

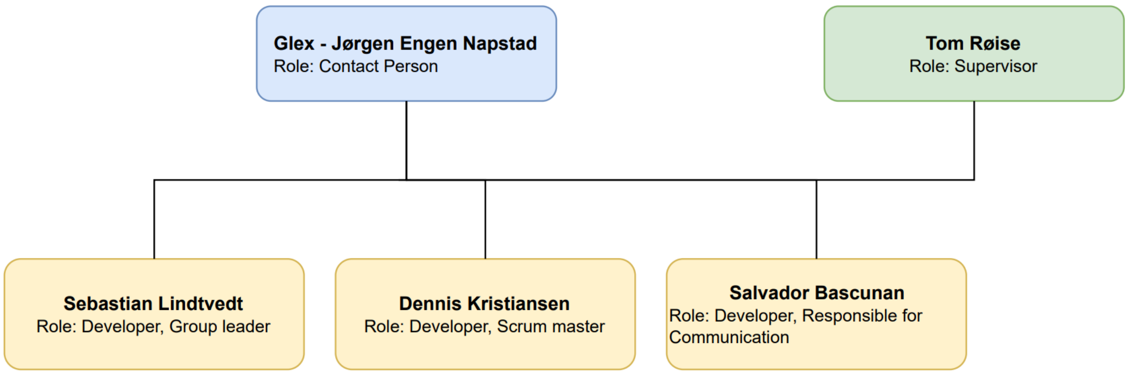
G.1.3 Group roles

Sebastian Lindtvedt is the group leader. This entails making sure the project's progress is moving forward. The group leader will also be responsible for solving internal conflicts

Dennis Kristiansen is the Scrum master. The responsibilities of a Scrum master includes: maintaining the issue backlog, training and coaching the other members on Scrum and agile development, removing barriers for the Scrum team, and making sure the Scrum events take place and serve their purpose.

Salvador Bascunan is mainly responsible for communication with Glex. This will entail keeping track of our weekly meeting invitations, schedule, change date or time if needed, and bringing topics between us and Glex if needed.

All the group members are also developers. This of course means that they are members of the Scrum team, and are responsible for taking on, and completing, issues from the sprint backlog, moving the project forward. The completion of all issue in each sprint is a shared responsibility for the whole Scrum team.


Glex - Jørgen Engen Napstad
Role: Contact Person
Tom Røise
Role: Supervisor
Sebastian Lindtvedt
Role: Developer, Group leader
Dennis Kristiansen
Role: Developer, Scrum master
Salvador Bascunan
Role: Developer, Responsible for Communication

G.2 Routines and Rules

Basic rules for group work:

- All members must attend daily Scrum meetings on Mondays and Thursday.

- All members attend project meetings, status meetings and meetings with the supervisor.
- All members are expected to work at least 30 hours each week on the project.
- All members should be available during the hours of 1000-1600 Monday-Friday, exceptions must be expected in some situations.
- All members must communicate their absence if it surpasses one hour during working time
- A meeting summary should be written for each formal meeting.
- All work hours should be tracked using Jira.
- Specific work tasks for each member will be distributed continuously through Jira.
- Should it happen that a member consistently breaks with routines and rules, the remaining members will hold a meeting with the project supervisor, to determine the consequences of these actions.
- If a team member is unable to comply with the rules, this must be informed in advance so that alternative solutions can be found. (e.g. a member is sick before a meeting)

H Planning, Follow up and Reporting

H.1 Development Process

H.1.1 Project Characteristics

This project will be the largest and most complex development task the group have taken on so far. This is something we have to consider when planning how we are going to conduct the project. It is of utmost importance that the process is well organized from start to finish, to ensure that things progress in a smooth manner. There will be many moving parts and a broad range of different things happening at the same time. There are several stakeholders involved, which have different expectations at different times. The key thing to draw from these characteristics is that the project has to be structured.

Some of the stakeholders representing the client do not have technical backgrounds in terms of software development. It is therefore essential to keep a close dialogue with them to avoid any miscommunication. This further strengthens the need for a structured development process.

The project naturally facilitates a variety of different programming languages and frameworks. This is due to the tasks at hand. There will be tasks such as parsing of different data formats, which will be conducted at non-scheduled time periods. An example of this is geological data. We do not know when new data will be presented, but we have to process it when it is released. This means that the processes for conversion, etc. needs to be run again. When creating several different microservices, an agile way to work is beneficial.

Due to the reasons mentioned above, the project will have to be conducted in a agile, but structured way. This is because there are several sub systems that needs to be implemented, which benefits from an agile methodology, and several different stakeholders that want to monitor the progress of the project that benefits from a good overall structure.

H.1.2 Software Development Model

The project facilitates an agile methodology. We need to create several different microservices, with varying languages and frameworks. This facilitates an incremental, rapid development cycle. Each microservice will be developed as an individual software project. This means that each microservice will be planned, designed, implemented and tested in its own cycle.

We had two different Software Development Models in mind when considering which one to use: Kanban and Scrum. Kanban consists of visualizing all the tasks of the project into a board. There is a maximum amount of tasks at any given time, and each team member is free to pick any task they see fit. Kanban is very flexible, and could be a good solution for this project, given that we are a small team, and we are communicating frequently. The problem with Kanban is that is lacks some of the structure Scrum provides. Kanban could very easily get out of hand, leading to a messy development environment.

A Scrum project is split into sprints. During each sprint, several Scrum events take place. Sprint review, sprint planning, daily stand-ups, etc. These meetings provide the project with structure. By scheduling several preplanned meetings in advance, it becomes much easier to actually implement them.

By comparing these two models, we concluded that Scrum provides us with the structure required for a project of this scale. We do however like the board from Kanban. We therefore plan to implement the board alongside the structure of Scrum.

H.1.3 Plan For Usage of Model

We will use a mostly standard Scrum development model, because that worked well in previous projects. We will organize the project into 2-week sprints, with two daily stand-up meetings each week, totaling to four stand-up meetings each sprint. At the beginning of each sprint, we will meet with Glex. During this meeting we will first do a sprint review, where we present the progress and results made in the previous sprint. After that we will transition the meeting into a sprint planning meeting, this way we will ensure that the project are progressing in a direction that both the client and ourselves are satisfied with. Finally, we will conduct a sprint retrospective, where we go over how the sprint went. The sprint retrospective will be conducted in the daily standup at the beginning of new sprints.

Issues will primarily be added at the sprint planning meeting, but issues that arise during development can also be brought up for discussion during the daily stand-up meetings.

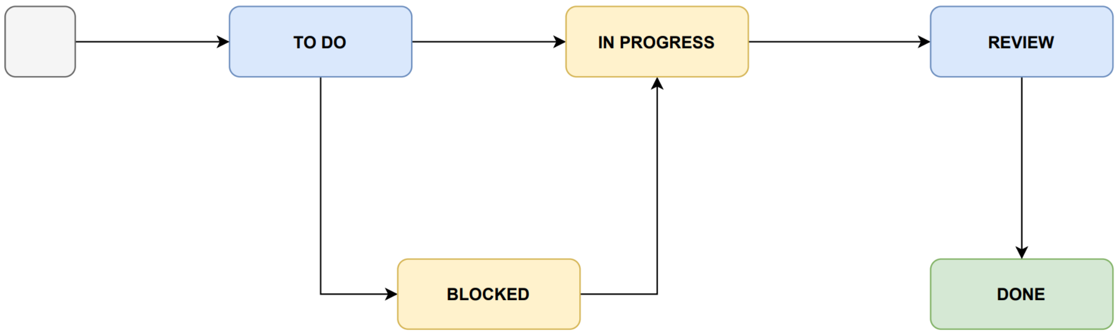
The issues will primarily be assigned during the sprint planning meeting, but issues can also be assigned during sprint stand-ups. Each team member is free to dispose the tasks they have been assigned freely. Meaning that it is each member's responsibility to ensure that each task is completed in the allocated time. All issues must pass through a review phase, before finally being marked as done. The review can either be conducted manually, or ideally through passing of automatic tests.

When a team member picks up an issue, they first transition the issue from "Todo" into "In progress". If they then discover that there are other issues that block their progress, the member can transition the issue into a "Blocked" state. When the issue is complete, the issue will be transitioned into "For review", where the other group members will review the issue before we can finally transition the issue into being "Done".

Sprint review will be conducted consecutively during each meeting with Glex. This ensures rapid feedback from the product owners and helps us as developers to develop a product the client is satisfied with.

Backlog refinement will be a "ongoing" process. We will do necessary backlog refinements in the daily standup meetings. Individual members are also allowed to refine issue throughout a sprint. New issues will be added, and old ones will be refined, by splitting up large issues, and combining small issues.

A sprint retrospective meeting will occur during the daily stand-up at the mondays in new sprints. At this meeting all team members will reflect on questions such as: "What went well?", "What could be improved?", "What should we commit to doing in the next sprint?"


TO DO
IN PROGRESS
REVIEW
BLOCKED
DONE

H.2 Plan for Status Meetings and Decision-Making

The tentative plan for meetings is as follows:

- Daily Scrum every Monday and Thursday, 10:00.
- Status meetings with supervisor Wednesday's every second week, 14:00.
- Status meetings with client Tuesday's every second week, 16:00.

I Organization of Quality Assurance

I.1 Documentation, Standards, Configuration Management

I.1.1 Documentation

In this project we will be exploring new technologies for developing the visualization parts of the digital twin. It's therefore vital that all the functionality and REST APIs are well documented. This allows Glex or any other entity, to understand, re-use parts, or continue development of the project. With this in mind, we have created some rules in order to keep the documentation organized:

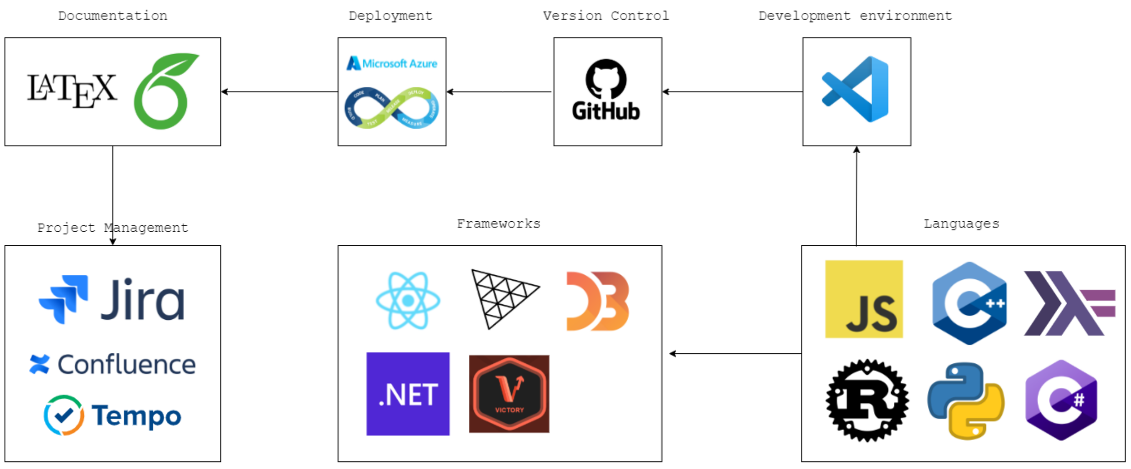
- Each git commit should be atomic, with a descriptive message explaining what and why the commit was made.
- Comments, doc-comments, and tests will serve as the main source of documentation for the source code.
- Each supervisor, Glex, and sprint meeting will be documented with participants, subjects and a short summary.
- READMEs for frontend, backend and additional services, will be kept up to date. This will include installation, build and deployment instructions.
- We will supply an OpenAPI specification for the main REST API of the project.

I.1.2 Standards

We will follow best practices regarding code and commenting standards of the respective technologies that we work with. This is to ensure the overall quality of the project, and to make it simpler for other group members to continue working on each others code. This will also aid us in the general documentation of the project.

I.1.3 Configuration Management

- Overleaf - We will use Overleaf as the main document writing tool for our Bachelor thesis. This will also be used for the project plan.
- Jira - This will be our main project management tool. Here we will create and assign issues, and keep track of them and their progress. This will also include tools for documentation and time tracking.
- Confluence - We will use Confluence for documenting our different meetings. These will be created in Jira and connected to their respective issue.
- Tempo - For time tracking we will use Tempo. This will be done inside of Jira. Here, we will be able to track time individually and connect these trackers to issues, be it tasks or meetings.
- GitHub - Our project code will be hosted on GitHub. We will use this for version control.
- Microservices - For the backend and API development of our project, we will use microservices. This allows us to use several different technologies. Not fully decided yet. Could be subject to change according to the requirements of the project.
- react-three-fiber - For visualization in frontend, we will use react-three-fiber. This is a framework that includes ThreeJS in the React component workflow. ThreeJS is a library built on WebGL, which allows for graphical rendering in the browser.
- Azure - For the cloud services in this project, we will use Azure.
- Visual Studio Code - The IDE of our choice. Has several plugins for integration with Jira, Github, etc.



I.2 Plan for Inspection and Testing

Having a test-driven development is something we have tried to utilize in previous projects, which is why we decided to make it a priority for this project. Our plan for testing is to identify the functionality needed to satisfy the requirements of the project, and write tests from the very beginning. This will be subject to change as the development process moves forward keeping in mind that changes can occur. In the section below, we will go through our plan for testing in this project, as well as an overview over the different frameworks we have available in the different technologies.

I.2.1 Testing

Our initial plan for the backend development is microservices. Here, we will implement independent testing of the individual services. This will entail a combination of mocking, stubbing, API validation testing, and others. A more specific overview of what we end up using will be provided in the final bachelor thesis report.

We will test the web app by using a combination of testing libraries for testing general JavaScript, React components, and ThreeJS based component. Jest will be the test runner and main testing library, supplemented by DOM Testing Library and React Testing Library for testing React components, and Fiber's own test renderer for testing ThreeJS based components. This will facilitate writing unit tests for all the units our web app consists of.

For the individual back-end services, we will use the appropriate testing framework, either built-in or third-party, for each of the languages and frameworks we choose to use. These will also facilitate writing unit tests, as well as writing integration and functional tests for using these services together.

Our general plan is to start with unit tests. A unit test is a programming testing method by which individual unit of source code are tested ([66]). This way we can test functionality independently, and not having to worry about breaking anything else.

Later we will move forward to integration tests. An integration test is when software modules are combined and tested as a group ([67]). And functional tests where we can test multiple modules together to see if they give us the result we expect. An example for this would be the generation of geometry from point cloud data. This requires multiple smaller modules to work together.

Technology	Framework
JavaScript/React	Jest
C++	doctest
Haskell	HUnit
Rust	Built-in
Python	Unittest
C#/.NET	xUnit

I.3 Risk Analysis

Table 5: Risk standard

Likelihood/ Consequence	**Minimal**	**Minor**	**Moderate**	**Significant**	**Critical**
highly likely	🟨	🟥	🟥	🟥	🟥
likely	🟨	🟨	🟥	🟥	🟥
probable	🟩	🟨	🟨	🟥	🟥
unlikely	🟩	🟩	🟨	🟨	🟥
highly unlikely	🟩	🟩	🟩	🟨	🟨

Table 6: Risks

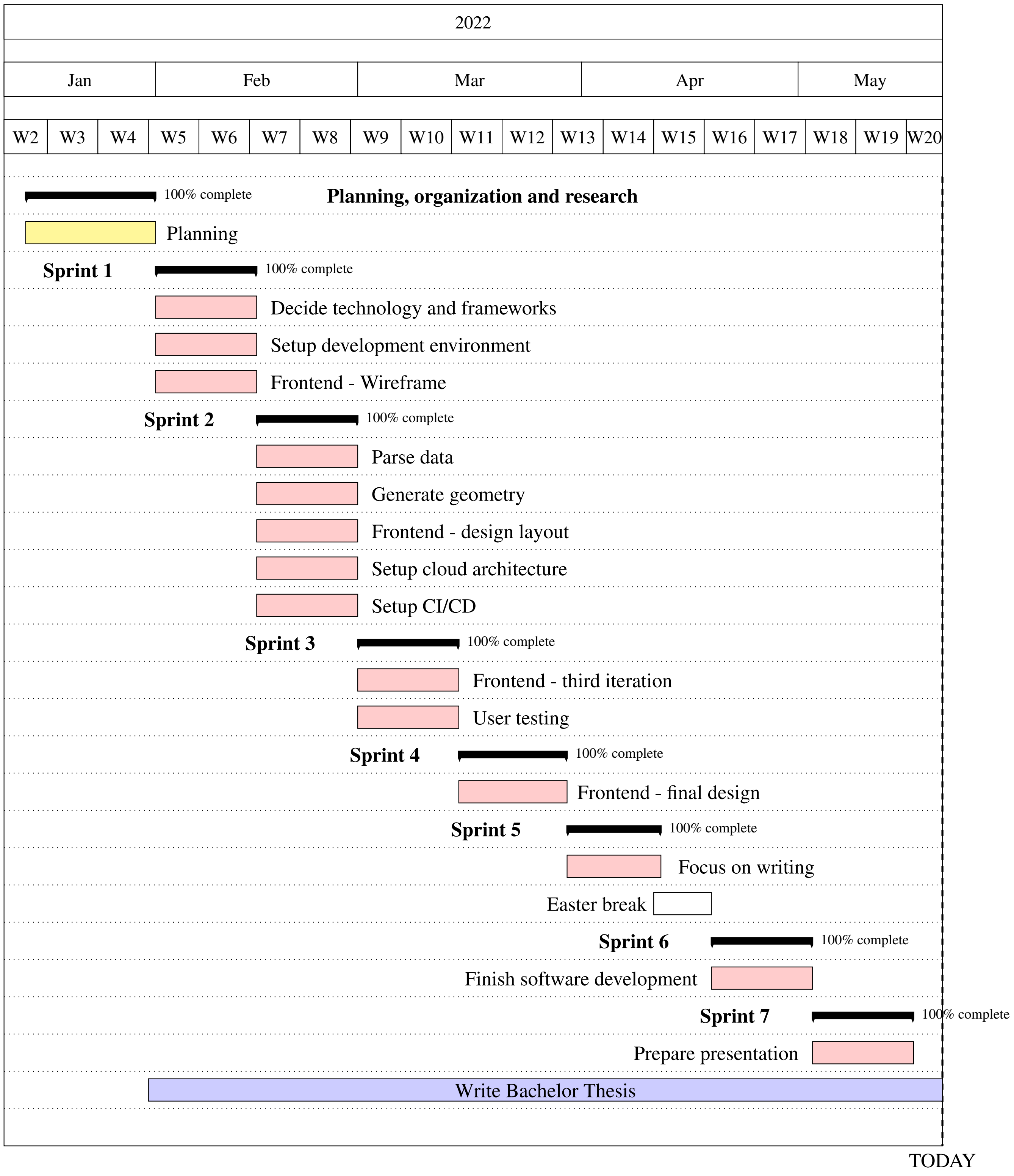
Risk	**Description**	**Likelihood/Consequence**
1	Loss of a group member (COVID, sickness, etc.)	Highly Unlikely/Significant
2	Not enough or not relevant data for digital twin.	Unlikely/Significant
3	Wrong choice of technologies, e.g., web not suitable for heavy visualization.	Unlikely/Significant
4	Product not finished by deadline.	Unlikely/Significant
5	Product does not satisfy customer requirements and expectations. Due to miscommunication, incorrect prioritization, etc.	Unlikely/Significant
6	Exceed the budget on cloud services.	Likely/Moderate
7	Unable to perform user testing because of COVID-19 and the limited demographic for the service.	Likely/Significant
8	Loss of documentation and/or source code.	Highly Unlikely/Significant
9	Insufficient test coverage/weak tests.	Unlikely/Moderate

Table 7: Risk mitigation

Priority	**Risk**	**Mitigation**
Medium	1	The group will carry on, potentially with a smaller scope negotiated with the client. All progress and code written will be documented with tests and comments, allowing their work to be continued by other team-mates
Low	2	Work closely with Glex throughout the project to obtain the most relevant data for the project.
Low	3	Adopt the project scope to something that can be achieved despite the constraints.
Low	4	Divide the project into smaller parts that can be completed iteratively. Make it so that the project is not all or nothing, but rather that we will always have something to show.
Low	5	Work closely with Glex and communicate rapidly throughout the whole project.
High	6	Set up a budget and alerts if the forecast cost exceeds the budget.
High	7	Try, to our best ability, to conduct the user tests through an online platform. This can entail having frequent meetings online with potentials user testers. This can be achieved by having a development server up and running.
Medium	8	Use Overleaf as our main document writing tool for the bachelor thesis. We will also have our source code in a Github repository and commit frequently. There will also be local storage of the project files.
Medium	9	Document the project in other ways, such as in the bachelor thesis. This will ensure good documentation regardless of test performance

J Plan of Action

J.1 GANTT diagram



J.2 Milestones and Crash Courses

J.3 Milestones

1. 31.01.22 - Finish Project Plan
2. 14.02.22 - Finish sprint 1
3. 28.02.22 - Finish sprint 2
4. 14.03.22 - Finish sprint 3
5. 28.03.22 - Finish sprint 4
6. 11.04.22 - Finish sprint 5
7. 11.04.22 - Split up focus between development and writing
8. 11.04.22 - 18.04.22 - Easter break
9. 02.05.22 - Finish sprint 6
10. 03.05.22 - Switch main focus to writing
11. 16.05.22 - Finish sprint 7
12. 20.05.22 - Deliver thesis
13. 10.06.22 - Project presentation

J.4 Crash Courses

During the project period, we will have three crash courses. Our first crash course was held 11.01.22 in Teams. Where, we got an overview of what could benefit us when planning the project, having contact with our client, and writing our project plan.

Our second crash course will be about writing the bachelor thesis. This will be held sometime between 1 - 10 of March. And our last crash course will be about the bachelor presentation. This will be held sometime between 23 - 27 of May.

B Project Agreement (Prosjektavtale)

NTNU

Norges teknisk-naturvitenskapelige universitet

Fastsatt av prorektor for utdanning 10.12.2020

STANDARDAVTALE

om utføring av studentoppgave i samarbeid med ekstern virksomhet

Avtalen er ufravikelig for studentoppgaver (heretter oppgave) ved NTNU som utføres i samarbeid med ekstern virksomhet.

Forklaring av begrep

Opphavsrett
Er den rett som den som skaper et åndsverk har til å fremstille eksemplar av åndsverket og gjøre det tilgjengelig for allmennheten. Et åndsverk kan være et litterært, vitenskapelig eller kunstnerisk verk. En studentoppgave vil være et åndsverk.

Eiendomsrett til resultater
Betyr at den som eier resultatene bestemmer over disse. Utgangspunktet er at studenten eier resultatene fra sitt studentarbeid. Studenten kan også overføre eiendomsretten til den eksterne virksomheten.

Bruksrett til resultater
Den som eier resultatene kan gi andre en rett til å bruke resultatene, f.eks. at studenten gir NTNU og den eksterne virksomheten rett til å bruke resultatene fra studentoppgaven i deres virksomhet.

Prosjektbakgrunn
Det partene i avtalen har med seg inn i prosjektet, dvs. som vedkommende eier eller har rettigheter til fra før og som brukes i det videre arbeidet med studentoppgaven. Dette kan også være materiale som tredjepersoner (som ikke er part i avtalen) har rettigheter til.

Utsatt offentliggjøring
Betyr at oppgaven ikke blir tilgjengelig for allmennheten før etter en viss tid, f.eks. før etter tre år. Da vil det kun være veileder ved NTNU, sensorene og den eksterne virksomheten som har tilgang til studentarbeidet de tre første årene etter at studentarbeidet er innlevert.

1. Avtaleparter

Norges teknisk-naturvitenskapelige universitet (NTNU) Institutt: NTNU Gjøvik
Veileder ved NTNU**: Tom Røise** e-post og tlf**.** **tom.roise@ntnu.no, 97139769**
Ekstern virksomhet: **Glex AS** Ekstern virksomhet sin kontaktperson, e-post og tlf.: **Jørgen Engen Napstad** **jorgen@glex.no, 93406526**
Student: Salvador Bascunan
Student: Sebastian Lindtvedt
Student: Dennis Kristiansen

Partene har ansvar for å klarere eventuelle immaterielle rettigheter som studenten, NTNU, den eksterne eller tredjeperson (som ikke er part i avtalen) har til prosjektbakgrunn før bruk i forbindelse med utførelse av oppgaven. Eierskap til prosjektbakgrunn skal fremgå av eget vedlegg til avtalen der dette kan ha betydning for utførelse av oppgaven.

2. Utførelse av oppgave

Studenten skal utføre: (sett kryss)

Masteroppgave	
Bacheloroppgave	**x**
Prosjektoppgave	
Annen oppgave	

Startdato: 11.01.22
Sluttdato: 20.05.22

Oppgavens arbeidstittel er: Smeaheia Digital Twin

Ansvarlig veileder ved NTNU har det overordnede faglige ansvaret for utforming og godkjenning av prosjektbeskrivelse og studentens læring.

3. Ekstern virksomhet sine plikter

Ekstern virksomhet skal stille med en kontaktperson som har nødvendig faglig kompetanse til å gi studenten tilstrekkelig veiledning i samarbeid med veileder ved NTNU. Ekstern kontaktperson fremgår i punkt 1.

Formålet med oppgaven er studentarbeid. Oppgaven utføres som ledd i studiet. Studenten skal ikke motta lønn eller lignende godtgjørelse fra den eksterne for studentarbeidet. Utgifter knyttet til gjennomføring av oppgaven skal dekkes av den eksterne. Aktuelle utgifter kan for eksempel være reiser, materialer for bygging av prototyp, innkjøp av prøver, tester på lab, kjemikalier. Studenten skal klarere dekning av utgifter med ekstern virksomhet på forhånd.

Ekstern virksomhet skal dekke følgende utgifter til utførelse av oppgaven: • **Bruk av skytjenester (Microsoft Azure)** • **Lisenser til eventuelle nødvendige rammeverk / verktøy** • **Reisekostnader ved eventuell fysisk workshop i Oslo eller Bergen**

Dekning av utgifter til annet enn det som er oppført her avgjøres av den eksterne underveis i arbeidet.

4. Studentens rettigheter

Studenten har opphavsrett til oppgaven[1]. Alle resultater av oppgaven, skapt av studenten alene gjennom arbeidet med oppgaven, eies av studenten med de begrensninger som følger av punkt 5, 6 og 7 nedenfor. Eiendomsretten til resultatene overføres til ekstern virksomhet hvis punkt 5 b er avkrysset eller for tilfelle som i punkt 6 (overføring ved patenterbare oppfinnelser).

I henhold til lov om opphavsrett til åndsverk beholder alltid studenten de ideelle rettigheter til eget åndsverk, dvs. retten til navngivelse og vern mot krenkende bruk.

Studenten har rett til å inngå egen avtale med NTNU om publisering av sin oppgave i NTNUs institusjonelle arkiv på Internett (NTNU Open). Studenten har også rett til å publisere oppgaven eller deler av den i andre sammenhenger dersom det ikke i denne avtalen er avtalt begrensninger i adgangen til å publisere, jf. punkt 8.

5. Den eksterne virksomheten sine rettigheter

Der oppgaven bygger på, eller videreutvikler materiale og/eller metoder (prosjektbakgrunn) som eies av den eksterne, eies prosjektbakgrunnen fortsatt av den eksterne. Hvis studenten skal utnytte resultater som inkluderer den eksterne sin prosjektbakgrunn, forutsetter dette at det er inngått egen avtale om dette mellom studenten og den eksterne virksomheten.

Alternativ a) (sett kryss) Hovedregel

	Ekstern virksomhet skal ha bruksrett til resultatene av oppgaven

Dette innebærer at ekstern virksomhet skal ha rett til å benytte resultatene av oppgaven i egen virksomhet. Retten er ikke-eksklusiv.

[1] Jf. Lov om opphavsrett til åndsverk mv. av 15.06.2018 § 1

Alternativ b) (sett kryss) Unntak

x	Ekstern virksomhet skal ha eiendomsretten til resultatene av oppgaven og studentens bidrag i ekstern virksomhet sitt prosjekt

Begrunnelse for at ekstern virksomhet har behov for å få overført eiendomsrett til resultatene: **Sluttresultatet av oppgaven vil ligge veldig tett på bedriftens kjernevirksomhet, strategi og videre kommersiell satsning, og selskapet ønsker derfor å stå fritt til å videreutvikle og potensielt kommersialisere en løsning basert på resultatet.**

6. Godtgjøring ved patenterbare oppfinnelser

Dersom studenten i forbindelse med utførelsen av oppgaven har nådd frem til en patenterbar oppfinnelse, enten alene eller sammen med andre, kan den eksterne kreve retten til oppfinnelsen overført til seg. Dette forutsetter at utnyttelsen av oppfinnelsen faller inn under den eksterne sitt virksomhetsområde. I så fall har studenten krav på rimelig godtgjøring. Godtgjøringen skal fastsettes i samsvar med arbeidstakeroppfinnelsesloven § 7. Fristbestemmelsene i § 7 gis tilsvarende anvendelse.

7. NTNU sine rettigheter

De innleverte filer av oppgaven med vedlegg, som er nødvendig for sensur og arkivering ved NTNU, tilhører NTNU. NTNU får en vederlagsfri bruksrett til resultatene av oppgaven, inkludert vedlegg til denne, og kan benytte dette til undervisnings- og forskningsformål med de eventuelle begrensninger som fremgår i punkt 8.

8. Utsatt offentliggjøring

Hovedregelen er at studentoppgaver skal være offentlige.

Sett kryss

x	Oppgaven skal være offentlig

I særlige tilfeller kan partene bli enige om at hele eller deler av oppgaven skal være undergitt utsatt offentliggjøring i maksimalt tre år. Hvis oppgaven unntas fra offentliggjøring, vil den kun være tilgjengelig for student, ekstern virksomhet og veileder i denne perioden. Sensurkomiteen vil ha tilgang til oppgaven i forbindelse med sensur. Student, veileder og sensorer har taushetsplikt om innhold som er unntatt offentliggjøring.

Oppgaven skal være underlagt utsatt offentliggjøring i (sett kryss hvis dette er aktuelt):

Sett kryss Sett dato

x	ett år	**01.06.23**
	to år	
	tre år	

Behovet for utsatt offentliggjøring er begrunnet ut fra følgende: **Offentliggjøring av sluttresultat vil potensielt medføre at selskapets strategi og forretningsmodell blir synliggjort tidligere enn ønsket.**

Dersom partene, etter at oppgaven er ferdig, blir enig om at det ikke er behov for utsatt offentliggjøring, kan dette endres. I så fall skal dette avtales skriftlig.

Vedlegg til oppgaven kan unntas ut over tre år etter forespørsel fra ekstern virksomhet. NTNU (ved instituttet) og student skal godta dette hvis den eksterne har saklig grunn for å be om at et eller flere vedlegg unntas. Ekstern virksomhet må sende forespørsel før oppgaven leveres.

De delene av oppgaven som ikke er undergitt utsatt offentliggjøring, kan publiseres i NTNUs institusjonelle arkiv, jf. punkt 4, siste avsnitt. Selv om oppgaven er undergitt utsatt offentliggjøring, skal ekstern virksomhet legge til rette for at studenten kan benytte hele eller deler av oppgaven i forbindelse med jobbsøknader samt videreføring i et master- eller doktorgradsarbeid.

9. Generelt

Denne avtalen skal ha gyldighet foran andre avtaler som er eller blir opprettet mellom to av partene som er nevnt ovenfor. Dersom student og ekstern virksomhet skal inngå avtale om konfidensialitet om det som studenten får kjennskap til i eller gjennom den eksterne virksomheten, kan NTNUs standardmal for konfidensialitetsavtale benyttes.

Den eksterne sin egen konfidensialitetsavtale, eventuell konfidensialitetsavtale den eksterne har inngått i samarbeidprosjekter, kan også brukes forutsatt at den ikke inneholder punkter i motstrid med denne avtalen (om rettigheter, offentliggjøring mm). Dersom det likevel viser seg at det er motstrid, skal NTNUs standardavtale om utføring av studentoppgave gå foran. Eventuell avtale om konfidensialitet skal vedlegges denne avtalen.

Eventuell uenighet som følge av denne avtalen skal søkes løst ved forhandlinger. Hvis dette ikke fører frem, er partene enige om at tvisten avgjøres ved voldgift i henhold til norsk lov. Tvisten avgjøres av sorenskriveren ved Sør-Trøndelag tingrett eller den han/hun oppnevner.

Denne avtale er signert i fire eksemplarer hvor partene skal ha hvert sitt eksemplar. Avtalen er gyldig når den er underskrevet av NTNU v/instituttleder.

Signaturer:

Instituttleder: Dato:
Veileder ved NTNU: Dato:
Ekstern virksomhet: Jørgen Engen Napstad [illegible] Dato: 27.01.2022
Student: Salvador Bascunan [illegible] Dato: 31.01.22
Student: Dennis Kristiansen [illegible]
Student: Sebastian Lindtvedt [illegible] Dato 07.02.2022

C Task Description (Oppgavebeskrivelse)

Skybasert digital tvilling for integrerte dynamiske systemer innenfor havrom

Oppdragsgiver:

Oppdragsgiver: Glex AS (Glex)
Kontaktperson: Jørgen Engen Napstad, CEO
Adresse: Skur 25, Møhlenpriskaien 8, 5006 Bergen
Telefon: +47 93 40 65 26
Epost: jorgen@glex.no
Hjemmeside: www.glex.no

Glex AS

Glex er et softwareselskap med kontorer i Bergen og Oslo, og består i dag av et team med bakgrunn i geologi, data management, digitalisering og spillutvikling.

Glex har siden 2017 utviklet produktet Glex Energy; Et samarbeids- og visualiseringsverktøy med interaktive presentasjonsmuligheter, benyttet av flere norske selskaper innenfor olje og gass.

Selskaper kombinerer sky- og spillteknologi (Unity) for å lage nyskapende og kompleks 2D og 3D visualisering.

Oppgaven:

Studentene vil få i oppgave å utvikle en skybasert digital tvilling av et (hypotetisk) scenario innenfor et konkret use-case, som for eksempel karbonlagring (CCS), fornybar energi, havbunnsmineraler, akvakultur eller et annet relevant energi/havrom-system.

Oppgaven legger til rette for at flere studentene jobber sammen i team med de ulike deloppgavene. Det er hensiktsmessig å definere oppgaven i fire hoveddeler: Datahåndtering, cloud, API og front-end. Den digitale tvillingen vil hente data fra forskjellig kilder og databaser, prosessere dataene i flere microservices, mellomlagre data i databaser, gjøre dataene tilgjengelige via et API, og til sist visualisere dataene gjennom en nettbasert 2D & 3D front-end løsning. Se nærmere beskrivelse av disse fire hoveddelene nedenfor.

Oppgaven vil være krevende og kompleks, da dette i utgangspunktet er en stor case med mange dynamiske delsystemer, og derfor er det viktig at del-systemene defineres og de forskjellige utviklings-oppgavene begrenses. Det er mulighet for at oppgavene også kan videreføres med økende grad av kompleksitet, og mulighet for oppskalering og videreutvikling på mastergrads-nivå.

Dersom man må begrense omfang er det mulig å benytte Glex sin eksisterende teknologi for å forenkle prosessen/oppgaven. Hvis studentene ønsker å grave seg dypere ned i visse deler av oppgaven og for eksempel velger å fokusere på å utvikle unike skalerbare front-end løsninger kan utvikling av et nytt API droppes, og Glex' eksisterende API kan benyttes.

Gjennomføring av oppgaven vil gi studentene innføring i et variert og relevant fagområde, med teknologi som er i bruk hos ledende softwareselskaper i dag.

Datahåndtering:
Dataene som skal visualiseres vil inkludere relativt tunge statiske datasett (geologi og geofysikk), GIS data, historiske meta-data, media (bilder og video) og sanntidsdata. Vi vil i så stor grad som mulig bruke reelle datasett fra industrielle aktører, men vi vil også vurdere å bruke «mock» data dersom dette er hensiktsmessig for noen av kildene.

Dataene vil samles og/eller genereres i skybaserte tjenester. Studentene vil så måtte velge mellom forskjellige database-løsninger (sql, no-sql, file/blob storage) for mellomlagring av dataene.

Cloud:
Hele systemet vil utvikles for, og kjøres i Microsoft Azure. Dette er skyløsningen Glex bruker i dag, og har god kontroll på. Vi vil anbefale å bruke en rekke tjenester for å sette studentene i gang, men vil holde det åpent for at studentene kan eksperimentere og velge disse selv.

Data-håndtering og prosessering vil foregå i en rekke micro-services. Disse tjenestene må utvikles, deployes, tunes og vurderes (sikkerhet og kostnad). Frittstående tjenester kan utvikles i flere språk (Eksempelvis Azure FunctionApps, som kan utvikles i C#, Java, Javascript, Python, Go, m.m. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview)

API:
All data som flyter mellom back-end og front-end skal bevege seg gjennom et API. Studentene kan enten utvikle dette selv, eller de kan velge å koble seg på selskapets eksisterende REST API.

Front-end:
Dataene skal presenteres til sluttbruker gjennom en skreddersydd og web-basert front-end løsning. Studentene kan selv velge mellom to innfallsvinkler/teknologivalg:

1. **Unity:** Studentene kan bruke spillutviklingsverktøyet Unity til å utvikle applikasjonen (Glex AS utvikler i dag sitt produkt i Unity, og har stor kompetanse innenfor dette). Dersom man velger denne retningen vil programmeringsspråk være C#, og studentene vil kunne lene seg på eksisterende teknologi og verktøy som allerede er bygget opp internt i Glex.
2. **Javascript og Three.js:** Studentene kan bruke ett av de populære JS rammeverkene (React, Angular, Vue, etc.) i kombinasjon med teknologien Three.js (https://threejs.org/). Går man for denne innfallsvinkelen vil Glex h mindre kunnskap å bidra med, men studentene vil bruke teknologi som er bredere brukt i industrien.

Front-end løsningen vil visualisere dataene i både 2D og 3D, og skal simulere en operativ modell brukt av f.eks. et energiselskap eller et industrielt service-selskap.

Prosjektstyring og dokumentasjon:
Vi vil vektlegge at studentene hele veien dokumentere sine løsninger, og vil legge til rette for at studentene kan dokumentere løsningen gjennom verktøy som er hyppig brukt i industrien i dag (Swagger, MkDocs, Confluence, etc.)

Vi ønsker at studentene skal bruke prosjektstyringsverktøyet Jira for å holde oversikt over prosjektet.

Vi vil gjennom hele prosjektet holde en løpende dialog om og veilede studentene med teknologivalg (kodespråk, hvilke tjenester de bruker i Azure, tredjeparts rammeverker som blir implementert, osv).

D Meeting Minutes

2022-01-10 Internal meeting

Date

10 Jan 2022

Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Goals

- Getting started after the break
- Think about the specific case and scope for the project

Discussion topics

Subject	Notes
Specific case	• Talked about visualizing a windmill park • We will talk to Glex more about this
Scope	• We want to be responsible and not take on a task bigger then what can be achieved.

Action items

✓ Specific case - We will keep an open mind in our meeting with Glex

Summary

We talked about what the bachelor assignment could entail. This was mostly towards what specific case we would be tackling. We agreed to keep an open mind and discuss this further after the meeting with Glex. We also agreed to do further research when it comes to the project setup, version control, time tracking and the rest of the documentation. We also discussed establishing a lot of the tools as soon as possible, in order to not have to go back and re-do things later.

2022-01-11 Client meeting

📅Date

11 Jan 2022

👥Participants

Group members

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Glex

- Jørgen Engen Napstad
- Brit Thyberg

🥅Goals

- Discuss the bachelor assignment
    - Scope
    - Specific case
    - Technologies
- Communication channels
- Meeting frequency
- Workshop

🗣Discussion topics

Subject	**Notes**
Bachelor assignment	• What is the our specific case going to be about ◦ Carbon ◦ Equinor • How are we going to scope the project • What should we look into as a pre-requisite to the project • How is the process of the project development going to look like
Communication Channels	• Group members use Discord for communication now • Glex uses Slack for chat and sharing of content, Teams for video calls
Meeting frequency	• Suggestion for weekly meetings, Tuesdays at 16.00
Workshop	• Needed to look into the opportunity of having offices in Oslo

✅Action items

- ✓ Established Slack channel for communication
- ✓ Look further into the specific case we could tackle as our bachelor assignment
- ✓ Set weekly meetings
- ✓ Closed the discussion regarding workshop

⤴Summary

Good first meeting for introductions. The group members will continue the discussion of our understanding of the project and communicate with Glex through Slack.

2022-01-11 Internal meeting

📅Date

11 Jan 2022

👥Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

🥅Goals

- Setup different tools
    - Jira
    - Overleaf
    - Time tracking
- Recap meeting with Glex
    - Specific case
    - Scope

🗣Discussion topics

Subjects	Notes
Specific case/Scope	• Still unclear as to what we are going to create a digital twin of

✅Action items

✓ Establish a clear idea of what we think can be made

📝Summary

We established some setup tools and started working with them and getting to know them. We have started creating some issues and looked into the time tracking. The idea here is to make it as seamless as we can, so it does not disrupt the workflow. We also discussed the meeting with Glex and decided to talk about this with our supervisor tomorrow. There is some confusion from our side about what we are going to create a digital twin of.

2022-01-12 Internal meeting

📅Date

12 Jan 2022

👥Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

🥅Goals

- Establish a couple of specific cases to show Glex
- Further work on the tools that will be used during development

🗣Discussion topics

Subjects	Notes
Specific case	• Further discussion on what we could present to Glex
Tools	• Trying to make the workflow seamless • Some issues with time tracking • Some issues with meeting documents

✅Action items

- ✓ Create a short text with image and sent to Glex
- ✓ Established more of the workflow in Jira

📝Summary

We drafted a short text for Glex and posted it on the Slack channel. We suggested a new meeting as soon as possible to make it easier to explain our thoughts clearly. We also continued working on the tools of the project, and got to know Jira better. We established more of the workflow regarding time tracking and meeting documentation.

2022-01-12 Supervisor meeting

Date

12 Jan 2022

Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- Tom Røise

Goals

- Introduction for the semester
- Summary of the meeting with Glex
- General guidance

Discussion topics

Subject	Notes
Meeting with Glex	• Positive meeting • Explained some confusion regarding what the specific case for the project is • Established some practical things, like communication channels etc.
General guidance	• How we should approach getting a clearer picture of what our specific case for the project is • What we should focus on otherwise moving forward

Action items

- [x] Clear idea on how to communicate some of the confusion with Glex

Summary

We received some good guidance on how to communicate our confusions with Glex. We also talked about our ambitions for this project and how we have started to plan and setup the different tools. We will focus a bit more on meeting logs and time tracking at the beginning to establish a good foundation.

2022-01-13 Glex meeting

Date

13 Jan 2022

Participants

Group members

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Glex

- Jørgen Engen Napstad
- Brit Thyberg

Goals

- Specific case for digital twin
- Combination between static and dynamic/live data
- Technologies
- Specification sheet frontend

Discussion topics

Subjects	Notes
Specific case	• CO2 storage well • Dataset • Set new meeting with data manager • Test technologies • Specification sheet for front end, what is required

Action items

✓ More details about the specific case digital twin of a CO2 storage well

Summary

Glex described more on the entire value-chain. We agreed to go more into detail regarding the CO2 storage well. We will begin writing our project plan and keep in touch with Glex for more information. From Glex's side, they agreed to create a specification sheet for the frontend that needs to be developed. This will make it easier for us to specify what the requirements are in the project plan.

2022-01-18 Glex meeting

📅Date

18 Jan 2022

👥Participants

Group members

- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- @ Salvador Bascunan

Glex:

- Jørgen Engen Napstad
- Brit Thyberg

🥅Goals

- Figure out details related to project plan document
- What does Glex want to achieve from this project?
- What is the overall goal of the software to be created?
- Who is going to use this system?
- What is the use case of the system?
- What do you want us to create?
- What sensors will be available?
- We need to know one specific case now, but it can change later. This is primarily a formal thing.
- Clear up expectations when it comes to the project case. What our role is, and what their role is.

🗣Discussion topics

Subject	Notes
Use case	• Who - Companies that do CCS - Equinor/Northern lights ◦ Evaluate real-time the workflow on CCS ◦ Log data - to be able to go back ◦ Multiple use cases - Investors/Geologists
What we are creating	• Future visualization for CCS - This is something that everyone will have to take into account sooner or later

✅Action items

☐

📝 Summary

Got more insight on what our specific project case was going to be. And who would possibly use the software.

2022-01-19 Supervisor meeting

📅Date

19 Jan 2022

👥Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

🥅Goals

- Feedback on previous project
- Bachelor project

🗣Discussion topics

Subject	**Notes**
Feedback	• Good idea on the project • Good implementation within timeframe • Good in the technology aspect • Good sections in the report - implementation and architecture • Could improve - security aspects • Could improve - user-friendly design - usability • Could improve - kravspek, brukergrensesnitt (a bit weak) • Could improve - how non-technical people would understand the project
Bachelor	• Specific case can be a bit open, even after the project planning • It’s not a problem that our project case changes a bit ◦ Important to keep it interesting for us
Other	• Don’t try to do everything - pick our battles

📝Summary

We got a good overview over the feedback given to us on our previous. This allows us to have some clear pointers regarding what we can improve on in this project. We also cleared some confusion and worry regarding our project case. We can write in the project plan that our case is not decided yet. We also got some tips on what to focus on in our project.

2022-01-25 Glex meeting

📅Date

25 Jan 2022

👥Participants

Group members

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Glex

- Brit Thyberg
- Jørgen Engen Napstad
- Patrick Sullivan

🥅Goals

- Clarify visualization in 3D
- More specifics

🗣Discussion topics

Subject	Notes
3D Visualization alongside well	• Important to Glex • Must further research the technical aspects around this, is this possible to implement?

📝 Summary

We agreed to further investigate technologies for frontend, and how some of the visualization can be implemented.

2022-01-26 Supervisor meeting

Date

26 Jan 2022

Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- Tom Røise

Goals

- Feedback on project plan

Discussion topics

Subject	Notes
Prosjektplan	• Klargjør om det er digital tvilling eller visualisering i henhold til forventningene • litt mer om Glex som et firma, mange kunder, hva slags kunder • spesifisere hele pipelinen i result goals • læringsmål? • om hva “skal ikke” inngå - delimitation • mer i case description - En eventuell visualisering av hele pipelinen • mer dybdebegrunnelse bak valgene - project characteristics • mer dybde og forklaring bak valg - plan for usage of model • noe rundt statuser i prosess-løpene? ◦ Epic? - f eks ha user testing etter 3 sprinter ◦ større statussjekk ◦ leveranse/testslipp e.l. i løpet av våren • bruker tester? • kunne vært flere risikoer ◦ forretning ◦ teknologi ◦ prosjekt

Summary

Got useful feedback on Project Plan.

2022-01-31 Internal meeting

📅 Date

31 Jan 2022

👥 Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

🥅 Goal

- Fill backlog
- Estimate story points
- Assign issues

🗣 Discussion topics

Subjects	**Notes**
Discuss story points	• Implement story points as a conceptual scale, measuring only the relation between different issues and not its scale in time. The scale will be a WIP as the project work continues.

✅ Action items

- [ ]

📝 Summary

Agreed to implement user stories.

2022-02-01 Client meeting

Date

01 Feb 2022

Participants

- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- @ Salvador Bascunan

Glex:

- Jørgen Engen Napstad
- Brit Thyberg
- Patrick Sullivan

Goals

- Discuss project agreement

Discussion topics

Subject	Notes
Project agreement	• Discuss further with supervisor

Summary

Agreed to discuss the project agreement further with supervisor.

2022-02-02 Supervisor meeting

Date

02 Feb 2022

Participants

- @ Sebastian Lindtvedt
- @ Salvador Bascunan
- @ Dennis Kristiansen
- Tom Røise

Goals

- Further discuss project agreement

Discussion topics

Presenter	Notes
Project Plan feedback	• As a whole - Still creates the expectation of more dynamic - The project plan has been approved ◦ dynamic vs static • Sources - Wikipedia is ok here, but be careful overusing it • Domain - precise • Domain - presis use of references - italic ◦ otherwise good • goals - effect - some could be quantitative ◦ bit difficult to specify exact ◦ should have some quantitative, despite not having numbers • goals - learning - practice is not a goal, but through this you can gain experience. consider merging • delimitation - is it created in a flexible manner where the object changes..? - are we developing for a special field. to specify if it can be used to something other than specified • requirement 3 - good • dont need that many sub-levels • characteristics - good • dev model - good case discussion, but reference to the point before. final report should contain references to specific sources • usage of model - combine SRM and SPM? • Sprint review meeting? sprint retrospective? • Figure out about logos • testing - clear detailt for introverted testing ◦ what about user testing ◦ what about system testing ◦ to find errors, but also to increase usability and understanding • risks - find more descriptive mitigations • gantt - mostly good - no other activities except easter, outside of school • milestones - can include something about user-testing, that is a milestone milestone
Gjenstående	• dev model - good case discussion, but reference to the point before. final report should contain references to specific sources • Figure out about logos • testing - clear detailt for introverted testing ◦ what about user testing ◦ what about system testing ◦ to find errors, but also to increase usability and understanding • milestones - can include something about user-testing, that is a milestone milestone • At numbers to risks in the top table

Action items

- [ ]

📝 Summary

Good feedback session.

2022-02-08 Client meeting

Date

08 Feb 2022

Participants

Group members

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Glex

- Jørgen Engen Napstad
- Brit Thyberg
- Patrick Sullivan

Goals

Discussion topics

Subject	**Notes**
Access to blob storage	• Added. confirmed with Azure Portal.
Meeting frequency	• Agreed to hold the communication open on Slack. • Agreed to have meeting every two weeks

Summary

Agreed on some practical subjects.

2022-02-14 Internal meeting

Date

14 Feb 2022

Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Goals

- Visualization of different parts (fault sticks, well logs, horizon, surfaces)
- Sprint planning
- Create user stories to different parts of the project

Discussion topics

Subject	Notes
General	• We discussed how the different parts of the digital twin could be visualized. • We still have some problems understanding how all of this would work together.
Horizons	• What do XLINE and INLINE mean? • We decided to ask Glex more about this to be able to write a user story
Fault sticks	• We understand enough to create a user story
Well logs	• We understand enough to create a user story
Surfaces	• Are the column and rows going to be used for anything, going to ask Glex

Summary

Productive meeting where we discussed some parts of the project which are still a bit unclear. How we are going to deal with scaling, rotation and translation of the object in the scene is still a bit unclear. We decided to ask Glex more about this in the Slack channel, and see if it makes things clearer. We also have a meeting tomorrow which could clarify more.

2022-02-15 Client meeting

📅 Date

15 Feb 2022

👥 Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Glex

- Jørgen Engen Napstad
- Brit Thyberg
- Patrick Sullivan

🥅 Goals

- Clarify some confusion around some areas of the project

🗣 Discussion topics

Subject	Notes
What are the Shape files going to be used for?	• 2D data to map • Will possibly not be implemented
What is lithostratigraphy? What does the data in the excel sheet mean? And how to visualize it?	• It will be visualized as start/end on well trajectory • Divided in Lithos, a certain trait
How do we visualize chronostratigraphy? Arrows and text?	• Same as lithostratigraphy, show timeperiod with start/end
Well trajectory surveys: Is the intention that receive the “raw trajectory survey data” and then calculate well-path?	• Calculate from raw data
Is it correct to say that the data from excel sheet always comes in an excel sheet, not in formal/official formats that are machine readable?	• The excel file can be converted to a machine readable format
How does positioning work in the scene? Will the models values be used the way they are, or are we supposed to adjust them manually?	• Positions come from UTM, all x y are inside a “box”
Feedback on Figma	• Fault sticks - correct visualization • Well logs - color palette

📝 Summary

Good a lot of answers to questions and doubts. Clarified some areas of the project.

2022-02-16 Supervisor meeting

📅 Date

16 Feb 2022

👥 Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- Tom Røise

🥅 Goals

- Feedback on Bachelor Thesis

🗣 Discussion topics

Subject	Notes
Feedback	• Introduction: POC eller functioning/extensible software - creates expectations ◦ if we create a digital twin for this dataset only ◦ or if we create a program that facilitates creating digital twin from dataset • Domain: Sources to Norwegian Government announcement • Product: Missing: Figure/Sketch/World + digital twin/a little bit about geologic-showing against what their vision is for digital twin/ short intro about static and dynamic data/ or intro about “storage” smeaheia figure • Project goals: Stakeholders • Use case: What else? discussion around methodology and technical decisions? operational requirements, issues/kanban? ◦ create some drafts and discuss with client maybe. • Development plan: maybe merge 3 and 5 ◦ risk analysis? • General: Need more about digital twins, like an intro with sources • Technical design: Sources ◦ Why have we structure it this way ◦ How did we figure out the different parts ◦ sources • Data processing: Discuss the different alternatives ◦ static and dynamic, how we thought about dividing this • REST API: is REST REST? or can be implemented as another type of “REST”? • GUI: and inspiration sources? - show different iterations. Traditional geologic software/ our sketches/ final solution. HCI / GUI patterns to show dynamic data? • Geometry Generation: Build credibility around how we found tools ◦ code snippets and explanation ◦ no so much text ◦ show tables • Deployment: can include diagrams and sketch from supplier • Testing: Include more types of testing • Discussion: Talk about sustainability and the groups contribution to the world • Use the “emnebeskrivelse” in discussion

✅ Action items

📝 Summary

Good feedback session on our plan for the bachelor thesis.

2022-02-18 Client meeting

📅 Date

18 Feb 2022

👥 Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Glex:

- Patrick Sullivan

🥅 Goals

- Further clarification on subjects in project

🗣 Discussion topics

Subject	Notes
Explain measured depth vs. TVDSS vs. TVDKB and calculations and conversions.	• See figure for terms • Conversion via minimum curvature method • Useful to be able to toggle between them

✅ Action items

- [ ]

📝 Summary

Short meeting with Patrick. Helped us clarify some doubts about certain formulas.

2022-03-01 Client meeting

Date

01 Mar 2022

Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Glex

- Jørgen Engen Napstad
- Brit Thyberg

Goals

- Show off our progress so far and get some feedback.
- Show off our Cloud Architecture and get some feedback.
- Obtain permissions for Azure.

Discussion topics

Subject	**Notes**
What we presented	• How the fault sticks processing is going • How the different file types in well logs are handled • How the surfaces look, showed a small demo
Feedback	• Try to have a minimal prototype of the program to present next meeting • Good progress so far

Action items

- [ ] Check out the limits imposed on Azure Functions and Azure Web Jobs to make sure they will work for our purposes.

Summary

Good feedback session with regards to our progress so far.

2022-03-09 Supervisor meeting

📆 Date

09 Mar 2022

👥 Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- Tom Røise

🥅 Goals

- General update

🗣 Discussion topics

Subject	**Notes**
	• Nothing in particular to note.

✅ Action items

☐

📝 Summary

Updated supervisor with current progress.

2022-03-15 Client meeting

Date

15 Mar 2022

Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen

Glex

- Jørgen Engen Napstad
- Brit Thyberg

Goals

Discussion topics

Subject	Notes

Action items

- [ ]

Decisions

2022-03-23 Supervisor meeting

📅 Date

23 Mar 2022

👥 Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- Tom Røise

🥅 Goals

- Feedback on MVP
- General update

🗣 Discussion topics

Subject	Notes
Feedback MVP	• Looks good!
Well-logs 2D graphs	• Make sure we get more feedback from Glex
Other well-logs data	• Make a few different data visualization methods, do not spend time visualizing tonnes of data, focus on the different visualization methods.
How do we handle data, missing data, incorrect data?	• Write about this in the report. • Make sure we apply a consistent method for handling data.
Code quality?	• We have linting and CI on web-app and REST API, what about the rest? • How do we lint and check code quality for Python projects?

✅ Action items

📝 Summary

Good feedback session on MVP. Got some pointer on what to write about also.

2022-03-29 Client meeting

Date

29 Mar 2022

Participants

- @ Salvador Bascunan
- @ Dennis Kristiansen
- @ Sebastian Lindtvedt

Glex

- Jørgen Engen Napstad

Goals

- Go through web-app and get feedback

Discussion topics

Subject	Notes
UX/UI	• Well names on top of wells ◦ To differenciate • From Feasibility we will use seabed and top sognefjord fm, remove gassnova • Toggle surfaces on/off • Group faults - within the 3D cube • Be able to pick faults and see them at the same time • Group toggle functionality • Insert a map/image of the area somewhere in the UI
Navigasjon	• Add controls • Preset camera positions
Content	• Fix well-trajectory

Action items

Summary

Good feedback session with Glex.

2022-03-30 Supervisor meeting

Date

30 Mar 2022

Participants

- @ Salvador Bascunan
- @ Dennis Kristiansen
- @ Sebastian Lindtvedt
- Tom Røise

Goals

- Feedback session

Discussion topics

Subject	**Notes**
Feedback	• Explain how our solution can work with other dataset • Should talk about security in general ◦ How can someone “mess” with the service ◦ How can this be handled ◦ We must explain how and why we thought this through • Have a draft ready by 19.04

Action items

Summary

Feedback session with deadline on having a draft ready for review.

2022-04-21 Supervisor meeting

Date

21 Apr 2022

Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- Tom Røise

Goals

- Feedback session

Discussion topics

Subject	Notes
Table of contents	Trim down to two levels only, three is unnecessary
Glossary	More words need to be included
Introduction	How has the definition of a digital twin affected our work
General	Remember to bring forward our idea to create a more general pipeline for digital twins. Time perspective, past tense Comment on balance between static and real-time data Quotes, include who came with a certain claim, not just a number. Short explanation on who made the claim. Don’t simplify our own work. Don’t use words like “just”, “simple”, “relative” etc. Makes it seem like we have done a small job. Show before and after an optimalisation Not much on why the whole pipeline can be used in other geographical areas, this concerns all data types. Why we can just add a surface, faults etc.. from other datasets. Explain the usefulness of our service to the environment. How our solution can contribute to CO2 storage.
Target audience	Only for demo/test, or to a real audience? just for smeaheia or in general?
Delimitations	What is the consequence of just focusing on “Storage”, is this solution created with the intention of integrating it with other parts of the CCS What data will be updated live, which are not. Bit unclear now
Group organization	All team members are developers primarily, its not an extra job.
Use case	Is use-case right for us? do we need it? Can have just the diagram and issue board Remember that it can be difficult to understand the service. Show all functionality through use-cases, diagrams etc. A sketch of the service as a whole

Development plan	Our use of scrum, not in general Collaboration with Client Roles, meeting, assigning tasks, sprint length, focus, adjustments Gantt, can be useful with a before and after
Technical design	Should know more about the domain before the technical aspect begins. Describe a bit about the geology and CO2 storage before we go into our solution. Justify the solution.. Maybe also what the different elements are, like surfaces, well logs, faults etc..
System architecture	Sources, more content.
Project organization	“And so on”……….. Explain in more detail why this is a good structure for our project
Data processing	Introduce the reader to what surface data is, what a geologist uses it for Sources, discuss alternatives Why we ended up with the value 1000 Nothing is random, or stumble on, show that everything has intention and meaning “x is the value between them”, explain this better
Development process	Discuss Scrum roles, meetings etc..
GUI	Show earlier iterations, improvements etc..
Implementation	Where is it explained that this will run in a browser We didn’t stumble upon good framworks, explain how and why we use them When talking about algorithms, explain why this algorithm is good for our use-case. Show data that supports this. Show sources and code examples/snippets. What defines a good results? why is our result the best?
Well and well-logs	Figure and sketch is needed What is the level of quality of the data, and how do we deal with errors in the data Too simple explanation on why we use JSON, explain more. Code snippets, important Source to polynomial regression, why do we use this? Reference to requirement specification on why we calculate certain values, TVDSS etc. Before and after an optimalization, etc
Real time	Remember real time data…
Fault sticks	Show what faults are, figure etc..
Deployment	Why we use different technologies. What is it about our case that made us use Azure Functions. Not useful to talk about technologies if we dont explain what they do for us.
Testing	Talk about why we test everything. Performance test User tests, we made an early prototype. we got feedback etc. Advantages with the approach we used. Quick feedback etc
Discussion	Bring forward/focus on why its relevant today. we created a service that helps with CO2 capture etc. Bring forward sustainability and usefulness for the environment.

✅ Action items

📝 Summary

Extensive and productive feedback session on bachelor thesis.

2022-04-26 Client meeting

📅 Date

26 Apr 2022

👥 Participants

- @ Sebastian Lindtvedt
- @ Salvador Bascunan
- Jørgen Engen Napstad

🥅 Goals

- Showcase progress and discuss the last phase of the project with client

🗣 Discussion topics

Subject	**Notes**
Discussion	• Focus is bachelor thesis now • Maybe try and finish some of the last issues in the web-app

✅ Action items

⤴ Summary

Short meeting updating our client to the progress.

2022-04-27 Supervisor meeting

📅 Date

27 Apr 2022

👥 Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- Tom Røise

🥅 Goals

- Thesis feedback

🗣 Discussion topics

Subjects	Notes
Questions	• How can we formulate that were presented with a project proposal to create a specific digital twin, and we adjusted this to create a more general solution? • What adjustments would need to be made in order for another dataset to be used here • Text build-up ◦ Fault stick part is organized to read
Design vs discussion	Talk about possible alternatives at the start, discuss the consequence of those choices in the discussion
Surfaces Geometry	• Explain why we chose glTF format on the file ◦ We tried other format - include this • Create a figure/sketch - explain the steps on generating • URL to Open3D - sources in general • Sources behind algorithms, definitions and explain why these algorithms where considered • Define the criteria's for what makes an algorithm good for our purpose. compare them. maybe a table? • source to pivot image
Faults	• Images of faults, difficult to know what we are talking about • Sequence sketch as illustration • Difficult to see how each part is connected and works together • Image say very little, set it in a perspective so it’s easier to understand what is going on

✅ Action items

⤴ Summary

Productive feedback meeting.

2022-05-04 Supervisor meeting

Date

04 May 2022

Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- Tom Røise

Goals

- Thesis feedback

Discussion topics

Subject	Notes
Questions	• How do we explain the whole blob trigger functionality? • It is ok to have cosmosDB in the service, or should it be outside • How can we include “Data Manager” actor • Can we have blob storage in the service? • Functional and non-functional security requirements
Requirements	• More focus on what, not how • Explain how the requirements have evolved, right now get the impression of waterfall model. Careful stating that these are the final requirements • don’t use “would” • Say something about end-user earlier • Maybe include a sketch of the requirements • Don’t have anything regarding the solution in the requirements • include the use-case diagram earlier • figures on litho and chrono if possible • consider project goals in chapter 1 • characterize geologist more, what he/she knows etc. define the stereotype • Use use stories instead of high-level/low-level use cases • Think measurability
Development Plan	• Plan or process? • Just structure? maybe dynamic and easy to change • Bring forward that we experiment, iterate, etc.. • Don’t use supervisor name • Move technology choices to technical design • Gantt, split up to iterations
Deployment	• Architecture in deployment? Needs to be properly explained. deployment is more a transition to operation in an automated way. Be concrete about what we are talking about here. • Security, should this be in deployment? • Security on technical design • Our implementation in Azure, explain it this way • Long code snippets

Action items

Summary

Good feedback session on bachelor thesis.

2022-05-12 Supervisor meeting

Date

12 May 2022

Participants

- @ Salvador Bascunan
- @ Sebastian Lindtvedt
- @ Dennis Kristiansen
- Tom Røise

Goals

- Thesis feedback

Discussion topics

Subject	Notes
Title	• Find a name that reflects the project as a whole
Requirements	• Show surfaces, faults etc.. earlier, so the reader can understand what the report is about • Explain what the different elements are in this context • Show what it entails when a requirement is met
Use case	• Don't call it Azure ◦ Data management system • Say more about actors • Connect use-cases to issues
Technical design	• Overall image is a bit confusing • Talk about that there is not talk about how things were implemented before the implementation part
Figures and tables	• Name figures and tables, and add explanations
General	• Talk about how much code we have, and how it distributes
Discussion	• Result goals

Action items

Summary

Final session with supervisor. Productive feedback on bachelor thesis.

E Sprint Retrospectives

Sprint 1 Retrospective

📋 Overview

Reflect on past work and identify opportunities for improvement by following the instructions for the Retrospective Play.

Date	14.02.2022
Team	Bachelor Project
Participants	@ Sebastian Lindtvedt @ Dennis Kristiansen @ Salvador Bascunan

💭 Retrospective

ℹ Add your Start doing, Stop doing, and Keep doing items to the table below. We'll use these to talk about how we can improve our process going forward.

What worked well?	**What could be improved?**	**What should we commit to doing in the next sprint?**
• The two weekly standup meetings	• Reduce the amount of time spent on meetings, reduce the frequency	• Prepare more for the meetings with Glex and the supervisor
• Sprint length worked well	• The issues are vague and tends to lasts a long time	• More specific issues, spend a little more time planning issues at the beginning of each sprint

✅ Action items

Sprint 2 Retrospective

📋 Overview

Date	28.02.2022
Team	Bachelor Project
Participants	@ Sebastian Lindtvedt @ Dennis Kristiansen @ Salvador Bascunan

💭 Retrospective

Add your Start doing, Stop doing, and Keep doing items to the table below. We'll use these to talk about how we can improve our process going forward.

What worked well?	**What could be improved?**	**What should we commit to doing in the next sprint?**
	Time wasted on pending access to Azure	Be more clear in our messages with the client that access is crucial in terms of time
TDD in general is starting to feel better	We can create issues based on the different test cases	Use smart commits linked with specific issues, such as a specific test cases

✅ Action items

Sprint 3 Retrospective

Overview

Date	14.03.2022
Team	Bachelor Project
Participants	@ Sebastian Lindtvedt @ Dennis Kristiansen @ Salvador Bascunan

Retrospective

Add your Start doing, Stop doing, and Keep doing items to the table below. We'll use these to talk about how we can improve our process going forward.

What worked well?	**What could be improved?**	**What should we commit to doing in the next sprint?**
Integration of all the elements went well enough	Communication around who does what.	Define issues and assign them. Then stick to that.
The meetings went well when they occurred	Improve communication within the group in regard to planned meetings.	Notify earlier when scheduled meetings can’t be attended.

✅ Action items

Sprint 4 Retrospective

📋 Overview

Reflect on past work and identify opportunities for improvement by following the instructions for the Retrospective Play.

Date	28.03.2022
Team	Digital Twin
Participants	@ Dennis Kristiansen @ Sebastian Lindtvedt @ Salvador Bascunan

💭 Retrospective

Add your Start doing, Stop doing, and Keep doing items to the table below. We'll use these to talk about how we can improve our process going forward.

Start doing	**Stop doing**	**Keep doing**
• Nothing particular	• Nothing particular	• Nothing particular

✅ Action items

Sprint retrospective 5

📋 Overview

Reflect on past work and identify opportunities for improvement by following the instructions for the Retrospective Play.

Date	19.04.2022
Team	Digital Twin
Participants	@ Dennis Kristiansen @ Sebastian Lindtvedt @ Salvador Bascunan

💭 Retrospective

Add your Start doing, Stop doing, and Keep doing items to the table below. We'll use these to talk about how we can improve our process going forward.

Start doing	**Stop doing**	**Keep doing**
• Nothing particular	• Nothing particular	• Nothing particular

✅ Action items

- [ ]

F Timesheets

Issue / User	Logged
DIG-1 - Write project plan	**15.65**
Dennis Kristiansen	3.75
Salvador Bascunan	6.00
Sebastian Lindtvedt	5.90
DIG-2 - Setup collaboration tools	**4.38**
Dennis Kristiansen	3.00
Sebastian Lindtvedt	1.38
DIG-3 - Test vscode jira integration	**0.50**
Dennis Kristiansen	0.50
DIG-5 - Draft of project idea	**5.00**
Dennis Kristiansen	1.00
Salvador Bascunan	1.00
Sebastian Lindtvedt	3.00
DIG-6 - Research CCS and Northern Lights	**17.48**
Dennis Kristiansen	14.00
Salvador Bascunan	1.70
Sebastian Lindtvedt	1.78
DIG-7 - Research and Learn MS Azure	**9.75**
Dennis Kristiansen	9.75
DIG-8 - Internal meeting 10.01.22	**0.50**
Dennis Kristiansen	0.50
DIG-9 - Internal meeting 11.01.22	**3.00**
Salvador Bascunan	1.50
Sebastian Lindtvedt	1.50
DIG-10 - Supervisor meeting 12.01.22	**1.00**
Salvador Bascunan	0.50
Sebastian Lindtvedt	0.50
DIG-11 - Internal meeting 12.01.2022	**7.50**
Dennis Kristiansen	2.50
Salvador Bascunan	2.50
Sebastian Lindtvedt	2.50
DIG-12 - Glex meeting 11.01.22	**2.00**

Issue / User	Logged
Salvador Bascunan	1.00
Sebastian Lindtvedt	1.00
DIG-14 - Explore tools related to development	**35.83**
Dennis Kristiansen	3.00
Salvador Bascunan	15.55
Sebastian Lindtvedt	17.28
DIG-15 - Glex meeting 13.01.22	**3.00**
Dennis Kristiansen	1.00
Salvador Bascunan	1.00
Sebastian Lindtvedt	1.00
DIG-16 - Internal meeting 13.01.22	**2.00**
Dennis Kristiansen	1.00
Salvador Bascunan	1.00
DIG-17 - Internal meeting 14.01.22	**4.33**
Dennis Kristiansen	1.40
Salvador Bascunan	1.50
Sebastian Lindtvedt	1.43
DIG-18 - Internal meeting 17.01.22	**5.63**
Dennis Kristiansen	1.87
Salvador Bascunan	1.88
Sebastian Lindtvedt	1.88
DIG-19 - Experiment with react-threejs-fiber for frontend	**19.58**
Dennis Kristiansen	14.67
Salvador Bascunan	2.00
Sebastian Lindtvedt	2.92
DIG-20 - Research Three.js with vanilla react	**4.03**
Sebastian Lindtvedt	4.03
DIG-21 - Glex meeting 18.01.22	**2.08**
Dennis Kristiansen	0.75
Salvador Bascunan	0.67
Sebastian Lindtvedt	0.67
DIG-22 - Internal meeting 18.01.22	**3.00**

Issue / User	Logged
Dennis Kristiansen	1.00
Salvador Bascunan	1.00
Sebastian Lindtvedt	1.00
DIG-23 - Write about Planning, Follow up and Reporting	**3.93**
Sebastian Lindtvedt	3.93
DIG-24 - Learn LaTeX and BibTex	**0.50**
Dennis Kristiansen	0.50
DIG-26 - Supervisor meeting 19.01.22	**1.50**
Dennis Kristiansen	0.50
Salvador Bascunan	0.50
Sebastian Lindtvedt	0.50
DIG-27 - 20.01.22 Internal meeting	**6.05**
Dennis Kristiansen	2.00
Salvador Bascunan	2.05
Sebastian Lindtvedt	2.00
DIG-28 - Research ASP.NET Web APIs	**6.62**
Dennis Kristiansen	4.50
Salvador Bascunan	1.00
Sebastian Lindtvedt	1.12
DIG-29 - 22.01.22 Internal meeting	**2.00**
Salvador Bascunan	1.00
Sebastian Lindtvedt	1.00
DIG-30 - 24.01.22 Internal meeting	**10.08**
Dennis Kristiansen	2.50
Salvador Bascunan	3.60
Sebastian Lindtvedt	3.98
DIG-31 - 25.01.22 Glex meeting	**2.25**
Dennis Kristiansen	0.75
Salvador Bascunan	0.75
Sebastian Lindtvedt	0.75
DIG-32 - 26.01.22 Supervisor meeting	**1.50**
Dennis Kristiansen	0.50

Issue / User	Logged
Salvador Bascunan	0.50
Sebastian Lindtvedt	0.50
DIG-33 - Finish - Background	**0.50**
Salvador Bascunan	0.50
DIG-34 - Finish - Domain	**1.50**
Dennis Kristiansen	1.50
DIG-35 - Finish - Project Goals	**0.27**
Sebastian Lindtvedt	0.27
DIG-37 - Finish - Case Description	**0.75**
Dennis Kristiansen	0.75
DIG-38 - Finish - Planning, Follow up and Reporting	**0.62**
Dennis Kristiansen	0.33
Sebastian Lindtvedt	0.28
DIG-39 - Finish - Organization of Quality Assurance	**1.00**
Salvador Bascunan	1.00
DIG-41 - Setup Jira, sprints and SCRUM workflow	**1.00**
Dennis Kristiansen	1.00
DIG-46 - Finish - Configuration management	**0.35**
Sebastian Lindtvedt	0.35
DIG-47 - Internal meeting 27.01.2022	**4.07**
Dennis Kristiansen	1.37
Salvador Bascunan	1.35
Sebastian Lindtvedt	1.35
DIG-48 - 28.01.22 Internal meeting	**10.45**
Dennis Kristiansen	3.45
Salvador Bascunan	3.50
Sebastian Lindtvedt	3.50
DIG-49 - Setup test framework for the front-end	**4.00**
Dennis Kristiansen	4.00
DIG-50 - 31.01.22 Internal meeting	**9.17**
Dennis Kristiansen	3.17
Salvador Bascunan	2.95

Issue / User	Logged
Sebastian Lindtvedt	3.05
DIG-51 - Establish development environments	**0.65**
Sebastian Lindtvedt	0.65
DIG-52 - Setup .NET environment	**3.07**
Sebastian Lindtvedt	3.07
DIG-53 - Test Python for 3D geometry	**24.42**
Salvador Bascunan	24.42
DIG-54 - Research technologies	**9.02**
Dennis Kristiansen	4.75
Salvador Bascunan	0.75
Sebastian Lindtvedt	3.52
DIG-56 - Setup cloud architecture	**11.27**
Dennis Kristiansen	9.17
Sebastian Lindtvedt	2.10
DIG-57 - Setup CI/CD	**5.33**
Dennis Kristiansen	4.50
Sebastian Lindtvedt	0.83
DIG-58 - Design frontend first iteration	**4.75**
Dennis Kristiansen	0.75
Sebastian Lindtvedt	4.00
DIG-59 - Link Figma to Jira	**0.25**
Dennis Kristiansen	0.25
DIG-60 - Setup shared Figma project	**0.08**
Dennis Kristiansen	0.08
DIG-61 - 01.02.2022 Client meeting	**2.40**
Dennis Kristiansen	0.77
Salvador Bascunan	0.82
Sebastian Lindtvedt	0.82
DIG-62 - 02.02.2022 Supervisor meeting	**2.47**
Dennis Kristiansen	0.83
Salvador Bascunan	0.82
Sebastian Lindtvedt	0.82

Issue / User	Logged
DIG-63 - Setup initial layout for the main screen	**1.50**
Dennis Kristiansen	1.50
DIG-64 - Explore data	**6.90**
Dennis Kristiansen	3.50
Salvador Bascunan	2.00
Sebastian Lindtvedt	1.40
DIG-65 - 03.02.2022 Internal meeting	**2.50**
Dennis Kristiansen	0.83
Salvador Bascunan	0.83
Sebastian Lindtvedt	0.83
DIG-66 - Test Python for CSV files	**3.98**
Sebastian Lindtvedt	3.98
DIG-67 - Look into GitHub Actions vs. DevOps Piplelines	**6.00**
Dennis Kristiansen	6.00
DIG-68 - Research VictoryChart	**11.54**
Dennis Kristiansen	4.17
Sebastian Lindtvedt	7.38
DIG-69 - Experiment with React hooks	**24.20**
Dennis Kristiansen	4.17
Salvador Bascunan	6.82
Sebastian Lindtvedt	13.22
DIG-70 - Internal meeting 7.02.2022	**5.75**
Dennis Kristiansen	1.92
Salvador Bascunan	1.92
Sebastian Lindtvedt	1.92
DIG-71 - As a geologist, I want to visualize well logs alongside the well-trajectory	**29.75**
Dennis Kristiansen	29.75
DIG-72 - Parse LAS files	**5.40**
Dennis Kristiansen	5.40
DIG-73 - Replace xUnit with nUnit	**0.33**
Dennis Kristiansen	0.33
DIG-74 - Setup doctesting in Python	**1.00**

Issue / User	Logged
Salvador Bascunan	1.00
DIG-78 - Explain REST technologies	**0.65**
Sebastian Lindtvedt	0.65
DIG-79 - Explain data processing technologies	**3.23**
Salvador Bascunan	3.23
DIG-81 - Write about architecture	**3.00**
Dennis Kristiansen	3.00
DIG-83 - Write about TDD	**2.00**
Dennis Kristiansen	2.00
DIG-85 - 08.02.2022 Glex meeting	**1.85**
Dennis Kristiansen	0.62
Salvador Bascunan	0.62
Sebastian Lindtvedt	0.62
DIG-86 - Internal meeting 08.02.2022	**1.68**
Dennis Kristiansen	0.67
Salvador Bascunan	0.50
Sebastian Lindtvedt	0.52
DIG-87 - Visualize well logs with 2D graphs	**8.97**
Dennis Kristiansen	8.97
DIG-88 - Write Introduction	**2.90**
Sebastian Lindtvedt	2.90
DIG-89 - 10.02.2022 Internal meeting	**1.57**
Dennis Kristiansen	0.78
Salvador Bascunan	0.78
DIG-90 - Organize Thesis	**1.28**
Sebastian Lindtvedt	1.28
DIG-91 - Visualize Fault Sticks	**5.57**
Sebastian Lindtvedt	5.57
DIG-92 - Visualize Horizons	**1.00**
Salvador Bascunan	1.00
DIG-93 - Visualize Well logs	**4.22**
Dennis Kristiansen	1.33

Issue / User	Logged
Sebastian Lindtvedt	2.88
DIG-94 - 14.02.2022 Internal meeting	**4.12**
Dennis Kristiansen	2.25
Salvador Bascunan	0.87
Sebastian Lindtvedt	1.00
DIG-95 - As a geologist, I want to visualize fault sticks and their faults	**9.87**
Sebastian Lindtvedt	9.87
DIG-96 - Parse fault sticks	**13.10**
Sebastian Lindtvedt	13.10
DIG-97 - As a geologist, I want to visualize the well-trajectories in 3D	**14.97**
Dennis Kristiansen	14.97
DIG-98 - Generate 3D well-trajectories	**13.25**
Dennis Kristiansen	13.25
DIG-99 - As a geologist, I want to visualize the surfaces	**2.50**
Dennis Kristiansen	2.50
DIG-100 - Parse CSV files	**3.33**
Dennis Kristiansen	3.33
DIG-102 - Learn more about Azure	**10.75**
Dennis Kristiansen	8.83
Salvador Bascunan	1.92
DIG-103 - Learn about Azure Functions	**16.98**
Dennis Kristiansen	11.00
Salvador Bascunan	2.87
Sebastian Lindtvedt	3.12
DIG-104 - Write some subsections in Bachelor Thesis and organize for feedback	**1.08**
Salvador Bascunan	1.08
DIG-106 - Make generic in-scene depth vs. y graph	**5.25**
Dennis Kristiansen	5.25
DIG-107 - Make in-scene graph for chronostratigraphy	**1.50**
Dennis Kristiansen	1.50
DIG-108 - Experiment with React Hooks	**14.63**
Dennis Kristiansen	3.50

Issue / User	Logged
Salvador Bascunan	7.55
Sebastian Lindtvedt	3.58
DIG-109 - 15.02.2022 Glex meeting	**3.33**
Dennis Kristiansen	1.00
Salvador Bascunan	1.17
Sebastian Lindtvedt	1.17
DIG-110 - 16.02.2022 Supervisor meeting	**2.75**
Dennis Kristiansen	0.75
Salvador Bascunan	1.00
Sebastian Lindtvedt	1.00
DIG-111 - Make in-scene graph for lithostratigraphy	**2.50**
Dennis Kristiansen	2.50
DIG-112 - Learn about routing and traffic control for Azure	**4.50**
Dennis Kristiansen	4.50
DIG-113 - Parse text files	**2.38**
Salvador Bascunan	2.38
DIG-114 - Write tests for parsing of text files	**1.77**
Salvador Bascunan	1.77
DIG-115 - Generate GLTF file from point cloud	**39.40**
Salvador Bascunan	39.40
DIG-116 - 17.02.2022 Internal meeting	**3.02**
Dennis Kristiansen	1.00
Salvador Bascunan	1.02
Sebastian Lindtvedt	1.00
DIG-117 - 18.02.2022 Glex meeting	**1.40**
Dennis Kristiansen	0.50
Salvador Bascunan	0.45
Sebastian Lindtvedt	0.45
DIG-118 - Write test for GLTF generation	**3.38**
Salvador Bascunan	3.38
DIG-119 - Add name to mesh in gltf file	**1.43**
Salvador Bascunan	1.43

Issue / User	Logged
DIG-120 - Add normals to web-app import of surfaces	**2.08**
Salvador Bascunan	2.08
DIG-121 - Add color to surfaces based on height	**1.72**
Salvador Bascunan	1.72
DIG-122 - 22.02.2022 Internal meeting	**4.50**
Dennis Kristiansen	0.33
Salvador Bascunan	1.58
Sebastian Lindtvedt	2.58
DIG-123 - Review project plan	**6.13**
Sebastian Lindtvedt	6.13
DIG-124 - Export loaded data in a common format	**14.58**
Dennis Kristiansen	14.58
DIG-125 - Make Azure Function for uploading well-logs	**28.33**
Dennis Kristiansen	28.33
DIG-126 - Learn about Pandas	**8.95**
Dennis Kristiansen	8.95
DIG-128 - Test web-app with all surfaces	**10.60**
Salvador Bascunan	10.60
DIG-129 - Add test cases for color and pygltflib	**3.75**
Salvador Bascunan	3.75
DIG-130 - Create JSON from parsed fault sticks	**2.43**
Sebastian Lindtvedt	2.43
DIG-131 - Load fault sticks on frontend	**3.08**
Sebastian Lindtvedt	3.08
DIG-132 - Create geometry from loaded fault sticks	**36.68**
Sebastian Lindtvedt	36.68
DIG-133 - Serve fault stick JSON data with cloud function	**10.85**
Sebastian Lindtvedt	10.85
DIG-134 - 24.02.2022 Internal meeting	**2.67**
Dennis Kristiansen	0.67
Salvador Bascunan	1.00
Sebastian Lindtvedt	1.00

Issue / User	Logged
DIG-135 - Create architectural drawings	**2.43**
Sebastian Lindtvedt	2.43
DIG-136 - Internal meeting 28.02.2022	**1.57**
Dennis Kristiansen	0.50
Salvador Bascunan	0.53
Sebastian Lindtvedt	0.53
DIG-137 - Document parsing of fault sticks	**6.50**
Sebastian Lindtvedt	6.50
DIG-138 - Add a color generator	**5.37**
Salvador Bascunan	5.37
DIG-139 - Client meeting 01.03.2022	**4.90**
Dennis Kristiansen	1.67
Salvador Bascunan	1.62
Sebastian Lindtvedt	1.62
DIG-140 - As a group, finalize the bachelor thesis	**22.20**
Dennis Kristiansen	2.50
Sebastian Lindtvedt	19.70
DIG-141 - Learn about lasio	**0.33**
Dennis Kristiansen	0.33
DIG-142 - 02.03.2022 Lightning course - Report writing	**4.00**
Dennis Kristiansen	2.00
Salvador Bascunan	2.00
DIG-143 - Look into and setup storybook	**5.92**
Dennis Kristiansen	5.92
DIG-144 - 03.03.2022 - Internal meeting	**1.98**
Dennis Kristiansen	1.00
Salvador Bascunan	0.50
Sebastian Lindtvedt	0.48
DIG-145 - Setup cloud function for surfaces processing	**58.32**
Salvador Bascunan	56.38
Sebastian Lindtvedt	1.93
DIG-146 - Add data processing for horizons	**11.02**

Issue / User	Logged
Salvador Bascunan	11.02
DIG-147 - Add geometry generation for horizons	**14.02**
Salvador Bascunan	14.02
DIG-148 - Add color to horizon geometry	**0.77**
Salvador Bascunan	0.77
DIG-149 - Add tests to horizons functionality	**2.32**
Salvador Bascunan	2.32
DIG-150 - Add surfaces to a web app scene	**4.70**
Salvador Bascunan	4.70
DIG-151 - Setup victory for 2d graphs	**2.00**
Dennis Kristiansen	2.00
DIG-152 - Create static-web-service for web-app	**1.33**
Dennis Kristiansen	1.33
DIG-153 - Create web-service for REST API	**2.82**
Sebastian Lindtvedt	2.82
DIG-157 - Learn Victory	**1.00**
Dennis Kristiansen	1.00
DIG-158 - Implement current wireframe in web-app	**5.20**
Dennis Kristiansen	5.20
DIG-159 - 08.03.2022 Internal meeting	**5.57**
Dennis Kristiansen	2.00
Salvador Bascunan	2.00
Sebastian Lindtvedt	1.57
DIG-160 - 09.03.22 Supervisor meeting	**3.00**
Dennis Kristiansen	1.00
Salvador Bascunan	1.00
Sebastian Lindtvedt	1.00
DIG-161 - Calculate true vertical depth	**1.75**
Dennis Kristiansen	1.75
DIG-162 - Surfaces: Write about data processing	**19.67**
Salvador Bascunan	19.67
DIG-163 - Fix color on surface mesh	**2.82**

Issue / User	Logged
Salvador Bascunan	2.82
DIG-164 - 10.03.2022 Internal meeting	**2.43**
Salvador Bascunan	1.22
Sebastian Lindtvedt	1.22
DIG-165 - Rework UI	**27.92**
Sebastian Lindtvedt	27.92
DIG-166 - Test processing shape files	**3.32**
Salvador Bascunan	3.32
DIG-167 - 14.03.2022 Internal meeting	**4.03**
Dennis Kristiansen	1.17
Salvador Bascunan	1.63
Sebastian Lindtvedt	1.23
DIG-168 - 15.03.2022 Client meeting	**4.13**
Dennis Kristiansen	1.50
Salvador Bascunan	1.32
Sebastian Lindtvedt	1.32
DIG-169 - Improve color on surface meshes	**7.33**
Salvador Bascunan	7.33
DIG-173 - Improve functionality to better fit cloud function	**16.68**
Salvador Bascunan	16.68
DIG-174 - Optimize web-app performance	**2.98**
Dennis Kristiansen	2.98
DIG-175 - Experiment with data fetching methods	**1.00**
Dennis Kristiansen	1.00
DIG-176 - 21.03.2022 Internal meeting	**6.80**
Dennis Kristiansen	2.27
Salvador Bascunan	2.27
Sebastian Lindtvedt	2.27
DIG-179 - Generate UV mappings for well-trajectory geometry	**1.25**
Dennis Kristiansen	1.25
DIG-180 - Setup CosmosDB	**0.93**
Sebastian Lindtvedt	0.93

Issue / User	Logged
DIG-181 - Write fault data to CosmosDB	**11.68**
Sebastian Lindtvedt	11.68
DIG-182 - Generate texture maps with chrono and litho data	**13.00**
Dennis Kristiansen	13.00
DIG-183 - Surfaces: Write about geometry generating	**26.05**
Salvador Bascunan	26.05
DIG-184 - 23.03.2022 Supervisor meeting	**5.20**
Dennis Kristiansen	2.00
Salvador Bascunan	1.58
Sebastian Lindtvedt	1.62
DIG-185 - Clean up and document FSP code	**0.62**
Sebastian Lindtvedt	0.62
DIG-186 - Deploy cloud function	**8.22**
Sebastian Lindtvedt	8.22
DIG-187 - 24.03.2022 Internal meeting	**1.80**
Dennis Kristiansen	0.67
Salvador Bascunan	0.55
Sebastian Lindtvedt	0.58
DIG-188 - Overhaul state and data handling in web-app	**5.50**
Dennis Kristiansen	5.50
DIG-189 - Create endpoint for serving fault data in API	**26.97**
Sebastian Lindtvedt	26.97
DIG-190 - 28.03.2022 Internal meeting	**5.58**
Dennis Kristiansen	1.75
Salvador Bascunan	1.92
Sebastian Lindtvedt	1.92
DIG-191 - 29.03.2022 Client meeting	**2.57**
Dennis Kristiansen	0.92
Salvador Bascunan	0.75
Sebastian Lindtvedt	0.90
DIG-192 - Add labels to wells	**3.52**
Dennis Kristiansen	1.00

Issue / User	Logged
Sebastian Lindtvedt	2.52
DIG-193 - I want to see real-time data	**0.40**
Sebastian Lindtvedt	0.40
DIG-194 - Overhaul menu for toggling faults, surfaces, and chronostratigraphy	**6.67**
Dennis Kristiansen	6.67
DIG-195 - 30.03.2022 Supervisor meeting	**3.00**
Dennis Kristiansen	1.08
Salvador Bascunan	0.83
Sebastian Lindtvedt	1.08
DIG-196 - Fix faulty cloud function	**15.05**
Sebastian Lindtvedt	15.05
DIG-197 - Write surface data to cosmosdb	**10.27**
Salvador Bascunan	10.27
DIG-198 - Create endpoint for surfaces in API	**11.42**
Salvador Bascunan	11.42
DIG-199 - Meeting 05.04.2022	**4.35**
Dennis Kristiansen	1.50
Salvador Bascunan	1.50
Sebastian Lindtvedt	1.35
DIG-200 - Read fault data from endpoint	**5.47**
Sebastian Lindtvedt	5.47
DIG-201 - Mock real time data	**12.77**
Sebastian Lindtvedt	12.77
DIG-202 - 07.04.2022 Internal meeting	**7.65**
Dennis Kristiansen	3.50
Salvador Bascunan	2.07
Sebastian Lindtvedt	2.08
DIG-203 - Deploy dt-api with mocked data	**0.82**
Sebastian Lindtvedt	0.82
DIG-204 - Consume data and visualize on frontend	**16.50**
Sebastian Lindtvedt	16.50
DIG-205 - Make API endpoints for well-logs	**10.50**

Issue / User	Logged
Dennis Kristiansen	10.50
DIG-206 - Make Azure Function for composite logs	**2.50**
Dennis Kristiansen	2.50
DIG-208 - Actually fix not-working binding expression and lookup in composition-function...	**16.33**
Dennis Kristiansen	16.33
DIG-209 - Write about faults	**20.75**
Sebastian Lindtvedt	20.75
DIG-210 - 19.04.2022 Internal meeting	**5.22**
Dennis Kristiansen	1.83
Salvador Bascunan	1.67
Sebastian Lindtvedt	1.72
DIG-213 - Setup manifest for the area	**9.50**
Dennis Kristiansen	9.50
DIG-217 - Add tests for all endpoints	**1.68**
Sebastian Lindtvedt	1.68
DIG-219 - Add tests for /faults/overview	**1.03**
Sebastian Lindtvedt	1.03
DIG-220 - Add tests for /faults/exists	**0.55**
Sebastian Lindtvedt	0.55
DIG-221 - Add tests for /realtime/flowrate/<float:ts>	**0.23**
Sebastian Lindtvedt	0.23
DIG-222 - Add tests for /surfaces	**0.82**
Sebastian Lindtvedt	0.82
DIG-223 - Add tests for /surfaces/overview	**0.17**
Sebastian Lindtvedt	0.17
DIG-224 - Add tests for /surfaces/exists	**0.13**
Sebastian Lindtvedt	0.13
DIG-225 - 21.04.2022 Supervisor meeting	**3.42**
Dennis Kristiansen	1.17
Salvador Bascunan	1.08
Sebastian Lindtvedt	1.17
DIG-226 - Fix project plan	**0.12**

Issue / User	Logged
Sebastian Lindtvedt	0.12
DIG-227 - 25.04.2022 Internal meeting	**3.45**
Dennis Kristiansen	1.15
Salvador Bascunan	1.15
Sebastian Lindtvedt	1.15
DIG-228 - Handle feedback	**1.38**
Sebastian Lindtvedt	1.38
DIG-229 - Write about Development plan/process	**26.32**
Sebastian Lindtvedt	26.32
DIG-230 - 26.04.2022 Client meeting	**3.48**
Salvador Bascunan	1.75
Sebastian Lindtvedt	1.73
DIG-231 - Write about Requirements	**31.55**
Salvador Bascunan	31.55
DIG-232 - 27.04.2022 Supervisor meeting	**2.30**
Salvador Bascunan	1.15
Sebastian Lindtvedt	1.15
DIG-233 - 28.04.2022 Internal meeting	**1.50**
Dennis Kristiansen	0.50
Salvador Bascunan	0.50
Sebastian Lindtvedt	0.50
DIG-234 - Add the other wells	**2.00**
Dennis Kristiansen	2.00
DIG-235 - Write about well-logs	**11.28**
Dennis Kristiansen	11.28
DIG-236 - Write about deployment	**8.17**
Dennis Kristiansen	8.17
DIG-238 - 02.05.2022 Internal meeting	**5.68**
Dennis Kristiansen	2.00
Salvador Bascunan	1.83
Sebastian Lindtvedt	1.85
DIG-241 - Re-organize "data processing surfaces" into implementation	**6.30**

Issue / User	Logged
Salvador Bascunan	6.30
DIG-242 - Write use-cases and create use-case diagram	**8.28**
Salvador Bascunan	8.28
DIG-243 - 03.05.2022 Internal meeting	**3.17**
Dennis Kristiansen	1.17
Salvador Bascunan	1.17
Sebastian Lindtvedt	0.83
DIG-244 - Fix well maps endpoint returning incorrect output	**0.33**
Dennis Kristiansen	0.33
DIG-245 - 04.05.2022 Supervisor meeting	**6.32**
Dennis Kristiansen	2.22
Salvador Bascunan	2.00
Sebastian Lindtvedt	2.10
DIG-246 - Write about technical design	**40.52**
Dennis Kristiansen	6.42
Salvador Bascunan	10.02
Sebastian Lindtvedt	24.08
DIG-247 - Internal meeting 05.05.2022	**1.62**
Dennis Kristiansen	0.33
Salvador Bascunan	0.50
Sebastian Lindtvedt	0.78
DIG-248 - Add FPS Panel to web-app	**0.83**
Dennis Kristiansen	0.83
DIG-249 - Write about testing	**18.33**
Dennis Kristiansen	18.33
DIG-250 - Write about Graphical User Interface	**7.18**
Salvador Bascunan	7.18
DIG-251 - 06.05.2022 Internal meeting	**3.00**
Dennis Kristiansen	1.00
Salvador Bascunan	1.00
Sebastian Lindtvedt	1.00
DIG-252 - 07.05.2022 Internal meeting	**6.05**

Issue / User	Logged
Dennis Kristiansen	2.03
Salvador Bascunan	2.00
Sebastian Lindtvedt	2.02
DIG-253 - Add more display options for realtime data.....	**1.28**
Sebastian Lindtvedt	1.28
DIG-254 - 08.05.2022 Internal meeting.....	**0.92**
Salvador Bascunan	0.42
Sebastian Lindtvedt	0.50
DIG-255 - Write discussion.....	**19.18**
Dennis Kristiansen	1.00
Salvador Bascunan	6.87
Sebastian Lindtvedt	11.32
DIG-256 - 09.05.2022 Internal meeting.....	**2.13**
Dennis Kristiansen	1.00
Salvador Bascunan	0.42
Sebastian Lindtvedt	0.72
DIG-257 - 10.05.2022 Internal meeting.....	**4.53**
Dennis Kristiansen	1.50
Salvador Bascunan	1.50
Sebastian Lindtvedt	1.53
DIG-258 - 10.05.2022 Glex meeting.....	**1.02**
Dennis Kristiansen	0.33
Salvador Bascunan	0.33
Sebastian Lindtvedt	0.35
DIG-259 - Generate test coverage reports.....	**1.00**
Dennis Kristiansen	1.00
DIG-260 - 12.05.2022 Supervisor meeting.....	**7.48**
Dennis Kristiansen	2.53
Salvador Bascunan	2.50
Sebastian Lindtvedt	2.45
DIG-261 - 13.05.2022 Internal meeting.....	**11.10**
Dennis Kristiansen	3.73

Issue / User	Logged
Salvador Bascunan	3.57
Sebastian Lindtvedt	3.80
DIG-262 - Write geology 101 section	**4.10**
Salvador Bascunan	4.10
DIG-263 - Re-format all tables, listings, and figure to a common format	**1.37**
Dennis Kristiansen	1.00
Sebastian Lindtvedt	0.37
DIG-264 - Write implementation section of REST API	**3.67**
Dennis Kristiansen	3.67
DIG-265 - Write implementation section of frontend	**3.08**
Sebastian Lindtvedt	3.08
DIG-266 - Re-write implementation section of well-logs	**4.50**
Dennis Kristiansen	4.50
DIG-267 - Write about realtime data	**3.63**
Sebastian Lindtvedt	3.63
DIG-269 - Write about Horizons implementation	**5.05**
Salvador Bascunan	5.05
DIG-270 - Write abstract	**1.00**
Salvador Bascunan	1.00
DIG-271 - Write Conclusion	**1.53**
Sebastian Lindtvedt	1.53
DIG-273 - 14.05.2022 Internal meeting	**9.25**
Dennis Kristiansen	3.25
Salvador Bascunan	3.00
Sebastian Lindtvedt	3.00
DIG-275 - Iterate through Requirements	**4.93**
Salvador Bascunan	4.93
DIG-277 - 15.05.2022 Internal meeting	**5.58**
Dennis Kristiansen	1.83
Salvador Bascunan	1.85
Sebastian Lindtvedt	1.90
DIG-278 - Read thesis and take notes	**3.75**

Issue / User	Logged
Salvador Bascunan	3.75
DIG-279 - 16.05.2022 Internal meeting	**9.48**
Dennis Kristiansen	2.50
Salvador Bascunan	3.45
Sebastian Lindtvedt	3.53
DIG-280 - Clean up py-horizons and py-surfaces branches	**5.27**
Salvador Bascunan	5.27
DIG-281 - Thesis improvements	**64.23**
Dennis Kristiansen	15.00
Salvador Bascunan	26.23
Sebastian Lindtvedt	23.00
DIG-282 - 18.05.2022 Internal meeting	**5.25**
Dennis Kristiansen	1.75
Salvador Bascunan	1.75
Sebastian Lindtvedt	1.75
DIG-283 - 19.05.2022 Internal meeting	**12.32**
Dennis Kristiansen	4.50
Salvador Bascunan	2.03
Sebastian Lindtvedt	5.78

G Project Data Notes

Project data notes

Fault Sticks

An interpretation of where the faults are in the seismic. Many fault sticks combined equals a fault

Inside the file:

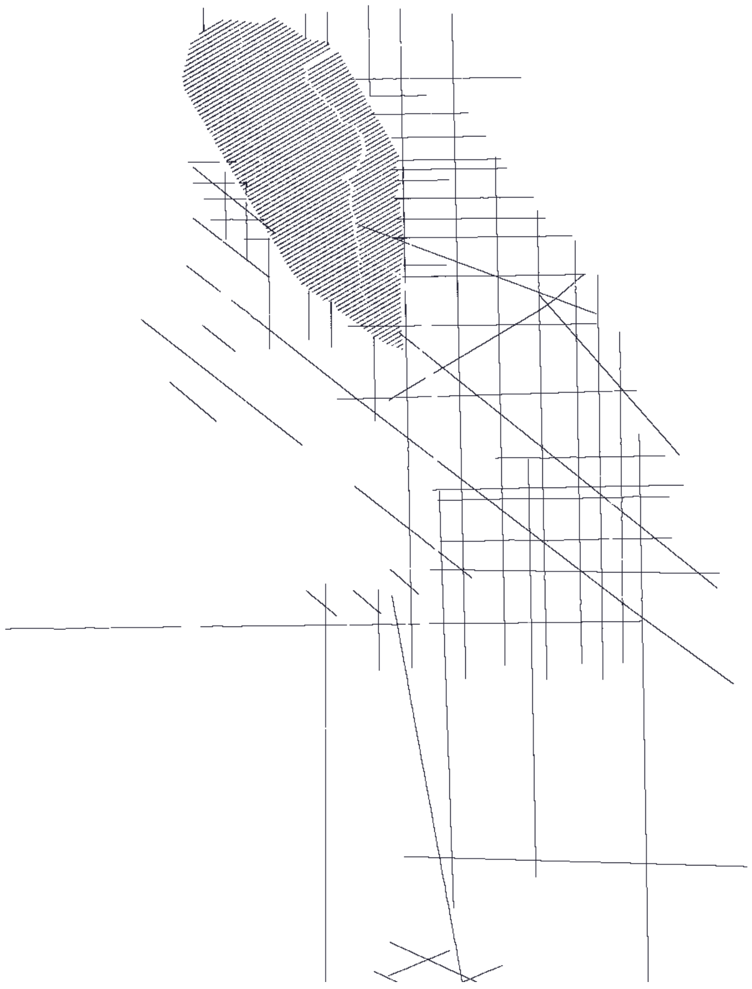
Fault name: Which fault this particular stick belongs to.

Stick: Which stick this point belongs to

Filename	Notes
fault_sticks_2010	
fault_sticks_2010.crsmeta.xml	
fault_sticks_GN1101_2012	
fault_sticks_GN1101_2012.crsmeta.xml	

Horizon/Key_Horizons_2010/Shape files

Filename	Notes

• Draupne Fm .dbf • Draupne Fm .prj • Draupne Fm .shp • Draupne Fm .shx	• dbf: attribute format; columnar attributes for each shape, in dBase IV format {content-type: application/octet-stream OR text/plain} • prj: projection description, using a well-known text representation of coordinate reference systems {content-type: text/plain OR application/text} • shp: shape format; the feature geometry itself {content-type: x-gis/x-shapefile} • shx: shape index format; a positional index of the feature geometry to allow seeking forwards and backwards quickly {content-type: x-gis/x-shapefile} 
Dunlin Gp .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Fenfjord Fm .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Johansen Fm .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Seabed .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Sognefjord Fm .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Statfjord Fm .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Top Shetland Group .dbf/.prj/.shp/.shx	More information in Draupne Fm notes

Horizons/Key_Horizons_2016

Filename	Notes
Top_Brent_206	• INLINE : ?? • XLINE : ?? • Last three values appear to be x y z coordinates

Top_Brent_206.crsmeta.xml	• Metadata • Contains the spatial context of the accompanied file • Unsure of how and what is going to be used from here
Top_Seabed_2016	Same as Top_Brent_206
Top_Seabed_2016.crsmeta.xml	Same as Top_Brent_206
Top_Sognefjord_2016	Same as Top_Brent_206
Top_Sognefjord_2016.crsmeta.xml	Same as Top_Brent_206
Top_Sognefjord_2016_3D_tracked	Same as Top_Brent_206
Top_Sognefjord_2016_3D_tracked.crsmeta.xml	Same as Top_Brent_206

Horizons/Key_Horizons_2016/Shape files

Filename	Notes
Seabed .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Top Brent .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Top Sognefjord 3D .dbf/.prj/.shp/.shx	More information in Draupne Fm notes

Horizons/Key_Horizons_GN1101_2012

Filename	Notes
Base Quaternary	Same as Top_Brent_206
Base Quaternary.crsmeta.xml	Same as Top_Brent_206
Fensfjord Fm	Same as Top_Brent_206
Fensfjord Fm.crsmeta.xml	Same as Top_Brent_206
Heather Fm 2	Same as Top_Brent_206
Heather Fm 2.crsmeta.xml	Same as Top_Brent_206
Krossfjord	Same as Top_Brent_206
Krossfjord.crsmeta.xml	Same as Top_Brent_206
Peak Draupne Fm	Same as Top_Brent_206
Peak Draupne Fm.crsmeta.xml	Same as Top_Brent_206
Seabed	Same as Top_Brent_206
Seabed.crsmeta.xml	Same as Top_Brent_206
Sognefjord Fm	Same as Top_Brent_206
Sognefjord Fm.crsmeta.xml	Same as Top_Brent_206
Top Dunlin Group	Same as Top_Brent_206
Top Dunlin Group.crsmeta.xml	Same as Top_Brent_206
Top Shetland Group	Same as Top_Brent_206
Top Shetland Group.crsmeta.xml	Same as Top_Brent_206

Horizons/Key_Horizons_GN1101_2012/Shape files

Filename	Notes
Base Quaternary .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Draupne Fm .dbf/.prj/.shp/.shx	More information in Draupne Fm notes

Dunlin Gp .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Fensfjord Fm .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Heather Fm 2 .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Krossfjord .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Seabed .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Sognefjord Fm .dbf/.prj/.shp/.shx	More information in Draupne Fm notes
Top Shetland Gp .dbf/.prj/.shp/.shx	More information in Draupne Fm notes

Surfaces/Surfaces_Picks_From_Feasibility_Phase

Filename	Notes
Sea bed	• Comments # • x y z column row • unsure if we need to process column and row for anything • Grid size - 669 x 519
Sea bed.crsmeta.xml	• Metadata • Contains the spatial context of the accompanied file • Unsure of how and what is going to be used from here
Top Sognefjord Fm	Same as Sea bed
Top Sognefjord Fm.crsmeta.xml	Same as Sea bed

Surfaces/Surfaces_GN1101_2012(Gassnova)

Filename	Notes
Base Quaternary	Same as Sea bed
Base Quaternary.crsmeta.xml	Same as Sea bed
Dunlin Gp	Same as Sea bed
Dunlin Gp.crsmeta.xml	Same as Sea bed
Fensfjord Fm	Same as Sea bed except: • Grid size - 224 x 235
Fensfjord Fm.crsmeta.xml	Same as Sea bed
Seabed	Same as Sea bed
Seabed.crsmeta.xml	Same as Sea bed
Top Draupne Fm	Same as Sea bed
Top Draupne Fm.crsmeta.xml	Same as Sea bed
Top Shetland Gp	Same as Sea bed
Top Shetland Gp.crsmeta.xml	Same as Sea bed
Top Sognefjord Fm	Same as Sea bed
Top Sognefjord Fm.crsmeta.xml	Same as Sea bed

Well data/Composite well logs

The files contain log data from a probe descending down a well bore hole, collecting various pieces of information, that we would like to visualize in the web app.

The LAS format (version 2) by The Canadian Well Logging Society, not to be confused with the LIDAR scan format with the exact same name. Is a format for storing this composite well log information.

In order to visualize the data with JavaScript, it would be convenient to convert it into a JSON based format, like https://jsonwelllogformat.org/ by Petroware AS. For this we can use Log I/O, which is proprietary? …and paid? Or invent our own JSON based format like what wellio.js have done (which is OSS and free). Maybe based on pandas DataFrame.to_json().

Well_Data/32-2-1

Filename	Notes
DTP_2008__993-1__32-2-1__WLC_COMPOSITE__1.LAS	

Well_Data/32-4-1 T2

Filename	Notes
NEW_4C__17-1__32-4-1_T2__COMPOSITE__1.LAS	

H Smeaheia Dataset License

SMEAHEIA DATASET

THIS IS A HUMAN-READABLE SUMMARY OF AND NOT A SUBSTITUTE FOR THE LICENSE

You are free to:

- Download and **use** the Licensed Material for non-commercial and commercial purposes
- Create, produce and reproduce **Adapt**ed Material
- **Share** the Licensed Material and/or the Adapted Material

Under the following terms:

- You may not sell the Licensed Material.
- You must give Equinor and Gassnova credit, and provide a link to these terms and conditions, as well as a copyright notice if applicable
- You may not share Adapted Material under a license that prevents recipients from complying with these terms and conditions
- You shall not use the Licensed Material in a manner that appears misleading nor present the Licensed Material in a distorted or incorrect manner.

This license is based on CC BY 4.0 license, two important changes are:

- The licensed material may not be sold
- The license covers all data in the dataset whether or not it is by law covered by copyright

SMEAHEIA DATASETS LICENSE
TERMS AND CONDITIONS FOR USE OF LICENSE TO DATA

1.1 Gassnova SF and Equinor ASA owns the Smeaheia Dataset. Gassnova and Equinor have decided to make this dataset available for you as Licensed Material, for the purpose of encouraging innovative uses of the data and knowledge development of CO2 storage technology.

1.2 By exercising the License and/or downloading the Licensed Material, you accept to be bound by these terms and conditions (**"the Terms and Conditions"**) as of that same time (**"the Effective Date"**).

2 DEFINITIONS

For the purposes of these Terms and Conditions, the following terms shall have the meaning stated below.

2.1 Adapted Material shall mean material that is derived from or based upon the Licensed Material and/or in which the Licensed Material is modified. Changes in formatting, file types and other changes that do not affect the content of the Licensed Material shall not result in the material being Adapted Material as the result will remain as Licensed Material.

2.2 **Data** shall mean qualitative or quantitative data and information, in any form or format, whether or not it is by law protected by copyright, by Sui Generis Database Rights, or neither.

2.3 **Intellectual Property Rights (IPR)** shall mean any and all intellectual property rights, such as patents, utility models, software, source codes, databases, trademarks, designs, domain names, copyrights, trade secrets or know-how.

2.4 **Licensed Material** shall mean the Data described in Exhibit 1.

2.5 **Sui Generis Database Rights** shall mean rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded.

3 LICENSE

3.1 The License

Subject to these Terms and Conditions, Gassnova and Equinor grant you a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to download and use the Licensed Material for non-commercial and commercial purposes, including to create, produce and reproduce Adapted Material **("the License")**.

3.2 General limitations

You shall not use the Licensed Material in a manner that appears misleading nor present the Licensed Material in a distorted or incorrect manner. You may not sell the Licensed Material. To the extent that your use of the Licensed Material is permitted by mandatory law, you do not have to comply with these Terms and Conditions.

3.3 Sharing Licensed Material and/or Adapted Material

You may share the Licensed Material and/or the Adapted Material, either openly or not, as long as Gassnova and Equinor are attributed. A notice that refers to these Terms and Conditions should be included, and if applicable, a copyright notice. If you share Adapted Material that may be shared under an adapter's license, the adapter's license must not prevent recipients of the Adapted Material from complying with these Terms and Conditions. Every recipient of the Licensed Material and/or Adapted Material will automatically receive an offer to exercise the License under these Terms and Conditions.

3.4 Moral rights

To the extent the Licensed Material contains copyright protected material, moral rights are not licensed under these Terms and Conditions. However, to the extent possible by law, Gassnova and Equinor waive and/or agree not to assert any such rights held by Gassnova and Equinor to the limited extent necessary to allow you to exercise the License, but not otherwise.

3.5 Sui Generis Database Rights

If you include all or a substantial portion of the Licensed Material in a database for which you have Sui Generis Database Rights, then that database (but not its individual contents) constitutes Adapted Material.

4 NO ENDORSEMENT

Nothing in these Terms and Conditions constitutes or shall be interpreted as you being, or your use of the License being, connected with, sponsored or endorsed by Gassnova or Equinor. You may not use Gassnova's or Equinor's name or trademark to support, recommend or market your use of the License or any products or services using or encompassing the License or Licensed Material.

5 LIABILITY AND INDEMNITY

5.1 No warranties or liability

The Licensed Material is provided "as is" and may contain errors or omissions. Gassnova and Equinor undertakes no liability for the risks of industrial realisation and commercial exploitation of the Licensed Material and Gassnova and Equinor shall have no liability regarding the fitness for purpose, quality, non-infringement, accuracy, or merchantability of the Licensed Material. Gassnova and Equinor provides no warranties, expressed or implied, either relating to the content or to the relevance of the Licensed Material. Gassnova and Equinor disclaim any liability for errors or defects associated with the Licensed Material to the maximum extent permitted by law.

5.2 Damages

Gassnova and Equinor are not liable for any of your damages, direct, indirect or consequential losses as a consequence of your use of the Licensed Material or infringement of third-party Intellectual Property Rights or other rights.

5.3 Indemnification

You shall indemnify Gassnova and Equinor for any and all liability for lawsuits and claims by third parties that arise as a consequence of your use of the License or infringement of third party Intellectual

Property Rights or other rights, insofar as such losses, lawsuits or claims are not a result of Gassnova's and Equinor's defect in title, gross negligence or wilful breach.

6 TERMINATION

6.1 Termination

If you fail to comply with these Terms and Conditions, then the License terminates automatically. The License may be reinstated if you cure the breach of these Terms and Conditions within 30 days of discovering the breach, or upon express reinstatement by Gassnova and Equinor.

6.2 Changes in distribution

Gassnova and Equinor may decide to offer the Licensed Material under other terms and conditions or stop distributing the Licensed Material at any time.

7 NORWEGIAN LAW AND DISPUTES

These Terms and Conditions **shall be governed by and interpreted in accordance with Norwegian law.**

The parties shall endeavour to resolve disputes concerning these Terms and Conditions through negotiation. **Disputes which are not resolved by mutual agreement within one month after negotiations were requested shall be settled by court proceedings brought before Stavanger District Court as legal venue.**

***

These Terms and Conditions are deemed accepted by you by exercising the License and/or downloading the Licensed Material.

EXHIBIT 1 – LICENSED MATERIAL

Licensed Material includes the following:

Data: Smeaheia Dataset, published via the CO2 DataShare online portal administrated by SINTEF AS (https://CO2datashare.org) as specified in the document "CO2DataShare Smeaheia Landing page" (Link to document).

The referenced release document with supporting documentation is included in the dataset.

Lindtvedt, Bascunan & Kristiansen

NTNU
Norwegian University of
Science and Technology