Voxel Assembler

3D Web configurator based on CAD-Data

Weiterbildung OST, MAS Software Engineering

Submitted by
Mauro Hefti
Uitikonerstrasse 33
8902 Urdorf

Roman Schweri
Wagenhauserstrasse 32
8260 Stein am Rhein

Tobias Wiesendanger
Rosenweg 1b
5040 Schöftland

Supervisor
Manuel Bauer

Degree program
MAS-SE 2023-25

1 Preface

This thesis is the result of our joint efforts as part of the Master of Advanced Studies at OST. Over several months, we, Tobias Wiesendanger, Roman Schweri, and Mauro Hefti, have worked collaboratively to research, design, and develop the solution presented in this document.

We would like to thank our supervisor, Manuel Bauer, for his valuable guidance and support throughout this project, as well as our co-supervisor, Tobias Büchel, for his insightful input on our frontend during the mid-review. We would also like to thank OST for providing us with the knowledge and foundation that prepared us for this master thesis.

2 Table of Contents

3 List of Abbreviations Used

Abbreviation	Full Name	Description
APS	Autodesk Platform Service	3rd Party Service
SVF	Simple Vector Format	Viewer format used
Step		Standard for the exchange of product data
.iam	Inventor Assembly File	
.ipt	Inventor Part File	
.idw	Inventor Drawing File	
API	Application Programming Interface	
BOM	Bill of Materials	
MPC	Model Parameter Constraints	
Viewable		A format that can be handled by the Autodesk viewer (SVF)
ADR	Architectural Decision Record	
CAD	Computer Aided Design	
EF Core	Entity Framework Core	
GUID	Globally Unique Identifier	
URN	Uniform Resource Name	
CI/CD	Continuous Integration / Continuous Deployment	
MuM	Mensch und Maschine	
SPA	Single Page Application	
ECS	Elastic Container Services	
IaC	Infrastructure as Code	
Inventor	CAD-System	Software from Autodesk

4 Introduction

4.1 Relevance of the Topic and Motivation

Modern customers expect personalized products rather than one-size-fits-all solutions Manufacturing sector faces increasing pressure for configurable products.

Time-consuming manual configuration processes and costly quote preparation for products that may not sell are the norm. Engineering resources are tied up in repetitive customization work.

Traditional CAD-based workflows are inefficient for customer-facing configuration and there is a need for web-based, accessible tools that non-technical users can operate.

4.2 Mensch und Maschine

Mensch und Maschine (MuM) is a leading Autodesk reseller and solution provider that serves thousands of customers across various industries including mechanical engineering, manufacturing, construction, and architecture. As an Autodesk Platinum Partner, MuM not only distributes CAD software but also develops custom solutions and provides consulting services to help companies optimize their design and engineering workflows.
For MuM, a web-based configuration solution represents both a business opportunity and a way to add value to their customer relationships, but we want to make it clear that we did not develop this web application for them. We had the opportunity to use resources like GitLab and such and for that we wanted to thank them.

4.3 Role Description

All the team members were highly involved in the planning phase.

Roman Schweri mainly worked in the frontend. He was responsible for building the user interface and providing a good user experience. In addition, he coordinated with Mauro and Tobias to integrate missing or fix buggy API endpoints. He contributed to architectural decisions and was also involved in frontend testing, design refinements and the overall usability of the app.

Mauro Hefti did a lot of the initial setup and later worked mainly in the backend. Most of the deployment and containerization was done by him.

Tobias Wiesendanger which provided the initial idea to this master thesis was focused on a lot of organisational task and preparation before starting. During the project he was focused on the backend and presenting architectural decisions to the team.

4.4 Problem Description and Thematic Delimitation

Currently customizing products according to the customer needs, takes a lot of time and does not guarantee selling it. Too much unpaid time is used in a lot of industries.

Because of the complexity and time-consuming work many industries don’t even provide this service to begin with.

4.5 Objectives

A web-based configurator should be created to no longer need specialized people (CAD-Designer) and specialized expensive software.
The customer should be able to configure his needs by himself or maybe with a sales representative without knowing how to operate cad software or even needing to have such software.

The company should be able to go from planning to manufacturing in the shortest time possible, with the least amount of manpower involved.
By not doing the planning manually no money is lost in that step, when in the end the customer does not buy the product.

4.6 Structure of the Paper

This thesis is structured to provide a comprehensive overview of the VoxelAssembler project, from initial planning through implementation and evaluation. First, some thoughts were put into explaining the main idea and how we want to achieve this, and then the more technical details are explained.

4.7 High level overview

This is a high-level overview of what will be created and what the target of this master thesis is. The topic is highly complex and will be further explained step by step throughout this document.

4.7.1 Step 1 CAD-Environment

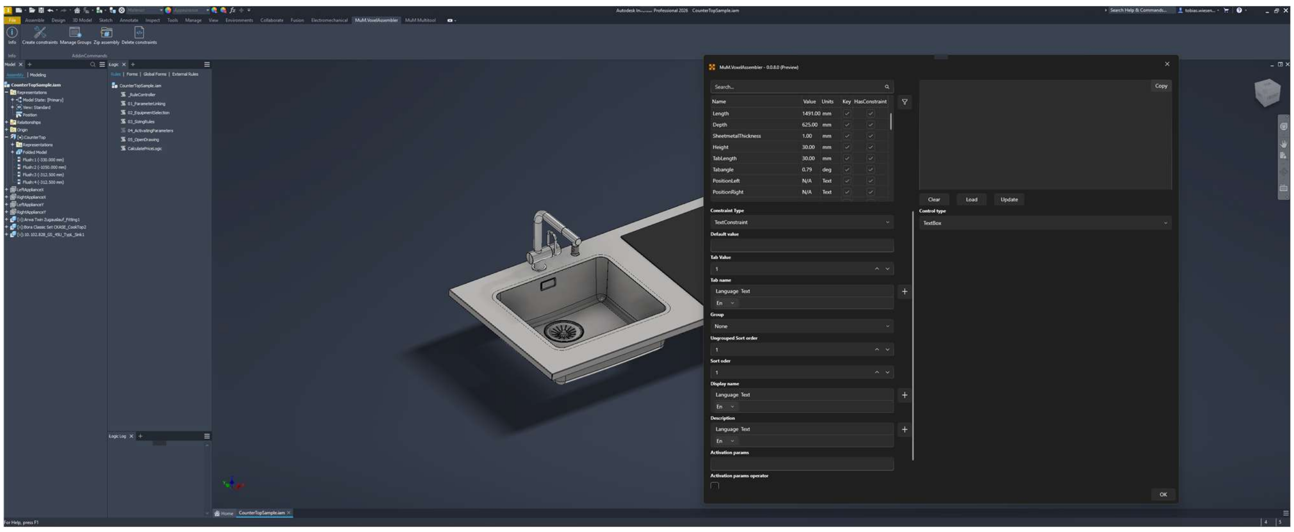
Customer prepares Product in the CAD Software called Autodesk Inventor. It is possible to use parametric modeling and logic by leveraging a technology called iLogic. (simple programming Language)

When finished the product is configurable locally, but a specialized user and a license for Autodesk Inventor is needed.

4.7.2 Step 2 MuM.VoxelAssembler

The next step for the cad designer is to describe how the configuration should look like in the frontend. There is a descriptive language called mpc (model parameter constraints) that allows to define what control should be used for the parameter and what constraints does it have. (min, max, interval etc.) Those are all defined in this descriptive language.

To make this user friendly an Inventor-Addin was created to allow doing this visually. The Addin is shown below working inside of Autodesk Inventor.



Name	Value	Units	Key	HasConstraint
Length	1491.00	mm	✓	✓
Depth	625.00	mm	✓	✓
SheetmetalThickness	1.00	mm	✓	✓
Height	30.00	mm	✓	✓
TabLength	30.00	mm	✓	✓
Tabangle	0.79	deg	✓	✓
PositionLeft	N/A	Text	✓	✓
PositionRight	N/A	Text	✓	✓

Constraint Type: TextConstraint
Default value:
Tab Value: 1
Tab name: Language Text, En
Group: None
Ungrouped Sort order: 1
Sort oder: 1
Display name: Language Text, En
Description: Language Text, En
Activation params:
Activation params operator:
Control type: TextBox

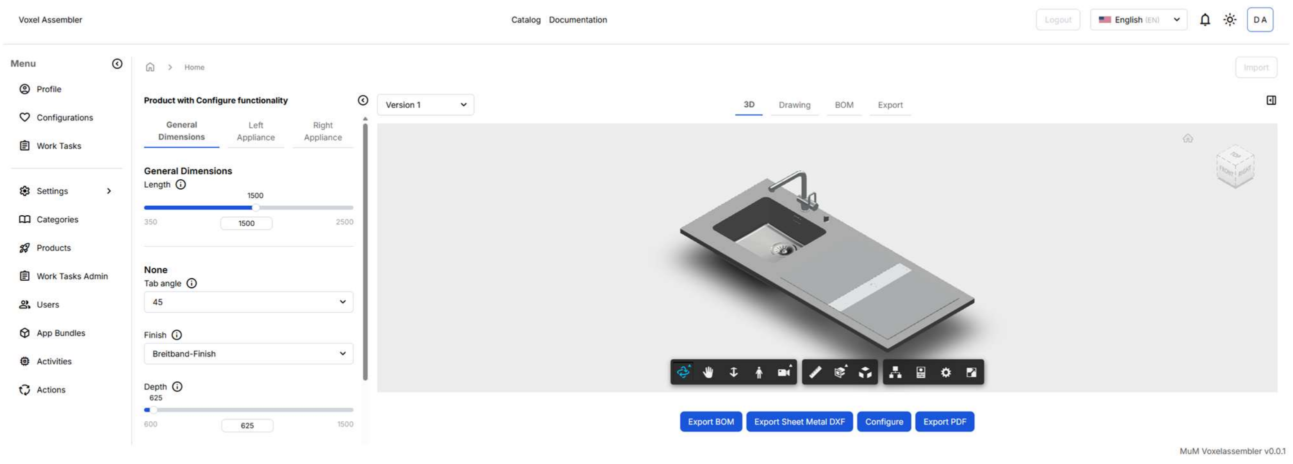
4.7.3 Step 3 App-VoxelAssembler

The frontend then allows to upload this CAD-Data by creating a new product version for a product.

The model parameter constraints are then interpreted and allow to build the frontend. On this product version the end-user is now able to configure his product by using the created controls and then firing an action on it.

What actions are available is also highly customizable. The product comes with some default, so called AppBundles. Each of those AppBundle has its own functionality. The most important one does apply those parameter changes to the model (see next step). More details are explained later in this documentation.

Below shown is the web interface for App-VoxelAssembler that allows a user to change parameter values and use actions on it like exporting a pdf.



Product with Configure functionality

Version 1

General Dimensions

Length: 1500 (350 – 2500)

None

Tab angle: 45

Finish: Breitband-Finish

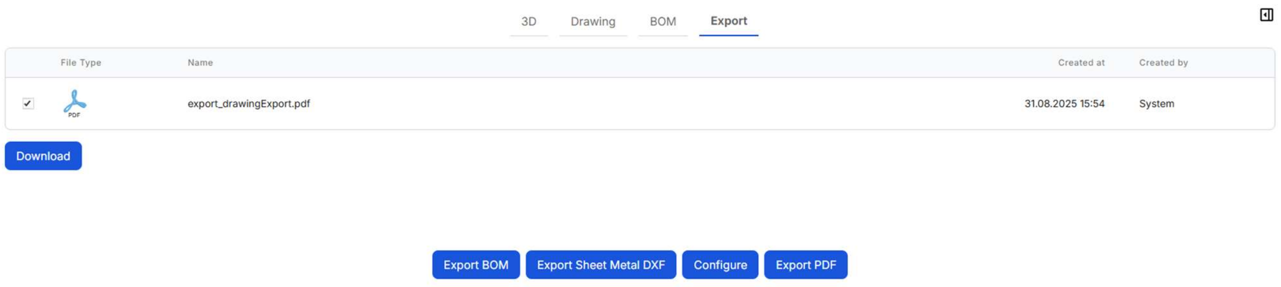
Depth: 625 (600 – 1500)

MuM Voxelassembler v0.0.1

4.7.4 Step 4 App-VoxelAssembler

Output can be used to start manufacturing. The output could be a technical drawing, CAD-Data or whatever the action created.

Below shows one of the possible exports, which is a pdf file from the drawing.



	File Type	Name	Created at	Created by
✓	PDF	export_drawingExport.pdf	31.08.2025 15:54	System

5 Project Planning

5.1 Project Approach Model

We knew what we need to build. Mainly because Tobias Wiesendanger works in that industry and talks daily to customers that could use this. Step one was starting off with a kick-off where all team members were brought up to speed.

We decided on lots of methodologies that day. For good communication we want to do a simple weekly meeting where each team member presents what he has done that week and what he is planning on doing.

Each meeting is documented. All meeting notes can be found in the attachments under **"\04_Project_Management\02_Meeting_Notes"**.

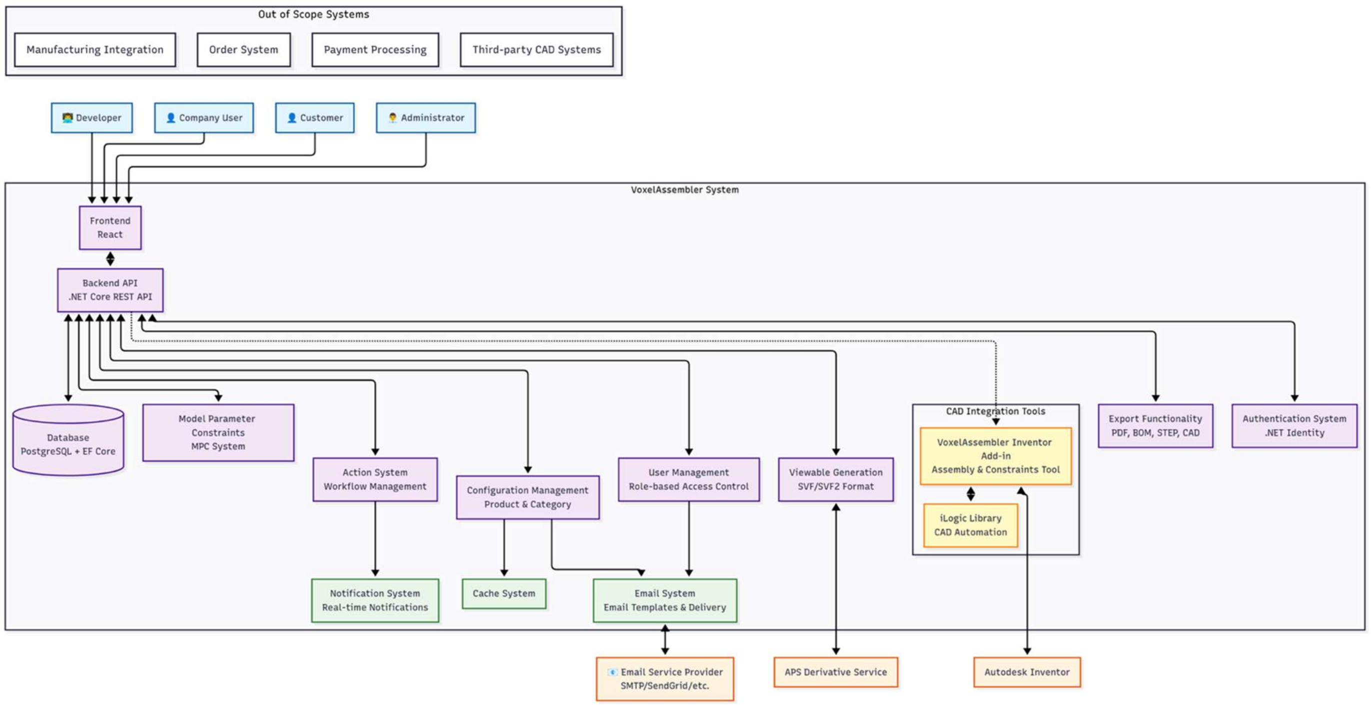
For planning we decided on using GitLab with Issues. Which lead to us creating milestones and assigning issues to each one. By having a weekly we had continuous feedback for all members.

At the end of each milestone a testing session was conducted for validation. Those were recorded and can be viewed in the attachments under **"\03_Quality_Assurance\01_Testing_Sessions"**.

5.2 System Delimitation

5.2.1 Context Diagram

This context diagram illustrates the VoxelAssembler system architecture and its interactions with external actors and services.



5.2.2 System Context / Scope

Inside System Scope:

Frontend Web Application - React-based configurator interface

Backend API - .NET Core REST API

Database - PostgreSQL with Entity Framework Core

Authentication System - .NET Identity endpoints

Configuration Management - Product and category management

Model Parameter Constraints (MPC) system

User Management - Role-based access control (CAD-Designer, End User, Administrator)

Viewable Generation - SVF/SVF2 format creation via APS Derivative Service

Action System - Workflow management for configurations

Export Functionality - PDF, BOM, STEP, native CAD formats

VoxelAssembler Inventor Add-in - Separate tool for preparing assemblies and constraints

iLogic Library - External library for CAD automation

Outside System Scope:

Order system to order configured products

Payment Processing - Not included in current scope

Manufacturing Integration - External systems for production

Third-party CAD Systems - Only Inventor is supported

5.3 Project Planning and Execution

5.3.1 Schedule, Milestones, Iterations

We planned with milestones to get a schedule. At the end of each Milestone a testing meeting was planned.

Preparation & Planning

Prepare everything needed to officially start the master thesis.

Phase 1

This milestone was focused on setting the foundations. User management, Authentication, Categories, Products and lots of settings were implemented in this Milestone. The Milestone Started March 31 and was finished on 31 May just in time.

Phase 2

In phase 2 the main functionally evolving around the configuration functionality was implemented. For this lots of things in the backend had to be implemented, like app bundles, actions, version management, activities and configuration management. Whatever was finished in the backend was then implemented in the frontend. Due to time a lot of Phase 3 planned features were already handled in this milestone. Many tasks were intermingled and had to be implemented together. Phase 2 was planned from June 1 to July 31 and was finished with about a week delay.

Phase 3

Phase 3 is focused on finishing all the must have features. Some more extended functionality for the configurator was implemented. Things like showing a BOM and a drawing were implemented now. In the frontend some smaller but visually appealing things were created like the possibility to change the design by choosing colors. This milestone was running from August 1 to August 31 and completed in time.

Small Task

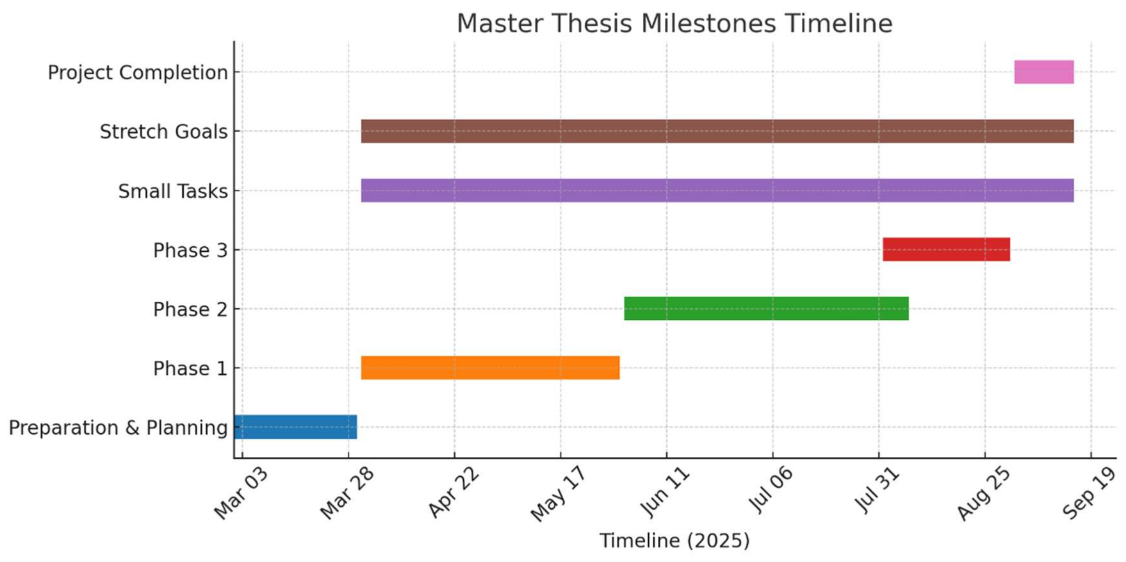
This milestone was constantly changed with small task that we had to keep track of but weren't that time consuming. Mostly things found from the testing session were added to this. There is no fixed end date for this milestone.

Stretch Goals

A lot of stretch goals were planned from the beginning, and even more were defined while developing. To keep track of them this milestone was used. This one also had due to its nature no end date and will stay open at the end if not all stretch goals were implemented. Which is highly likely.

Project completion

This milestone was used to keep track of the finishing tasks that need to be completed for handing in the master thesis. Like this documentation. This will end with the completion date of the master thesis which is 15.09.2025.



5.3.2 Time tracking

For time tracking issues were used. Time spent was constantly logged on a specific issue. To write a time report and keep track, a small tool was written that outputs a markdown file with time added together for each team member. All reports can be found here: "**\04_Project_Management\01_Time_Reports**".

At the start we did not estimate time for an issue, which we later changed for most issues. This can be used to get a velocity for future tasks.

The final report looks like this:

User	**Total Time**
Tobias Wiesendanger	762h
Mauro Hefti	397h
Roman Schweri	386h

5.4 Prioritization of Tasks and Requirements

During planning, we categorized all requirements into mandatory and optional features. This ensured a clear focus on delivering a working Minimum Viable Product (MVP) while leaving room for additional capabilities if time allowed.

5.4.1 Mandatory

The must have features mentioned in section 6 where clearly the mandatory Todo's. Still, we sometimes decided to fully build a feature before moving on with a clear commitment to finish the project.

5.4.2 Optional

Optional requirements were defined as "nice-to-have" features that extend usability, performance, or integration but were not critical for the MVP. They were placed in a lower priority and only addressed if milestone progress allowed. In the end we ended up making most of the optional features and even added more as the project progressed.

5.5 Risk Management

5.5.1 Risk Analysis

ID	Risk Description	Impact	Likelihood
R1	Scope too large for a three-student team	High	Medium
R2	Third-party APIs may change or break	Medium	Low
R3	A team member may drop out or become unavailable	Medium	Low
R4	Different customers may require different configurators (no one-size-fits-all solution)	Medium	High

5.5.2 Risk Mitigation Strategies

5.5.2.1 R1 – Scope too large for a three-student team

We have addressed this risk from the start by preparing thoroughly and defining a clear MVP (Minimum Viable Product) scope. Weekly meetings and milestone planning allow us to track progress effectively. If needed, we can reduce non-critical features to stay within time and resource constraints.

5.5.2.2 R2 – Third-party APIs may change or break

We rely on established APIs that have been in use for several years. Some of these APIs provide official client libraries maintained by their vendors, reducing the risk of sudden incompatibilities. Automated testing will help us identify potential issues early, enabling quick responses. Autodesk also communicates API changes in advance, allowing us to prepare accordingly.

5.5.2.3 R3 – A team member may drop out or become unavailable

We foster a collaborative team environment with shared responsibilities and regular knowledge-sharing sessions. Code reviews ensure that all members remain familiar with different parts of the project, minimizing the impact if one member becomes unavailable.

5.5.2.4 R4 – Different customers may require different configurators

We are aware that customers can have varying needs and requirements. To address this, we are designing a modular and flexible architecture with support for plugins or configuration options. This will allow us to adapt the configurator for different customer demands without developing entirely separate solutions.

5.6 Autonomy and Responsibility

We worked independently and took responsibility for our tasks. Whenever we encountered questions or challenges, we first discussed them among ourselves to find the best possible solution. For idea generation or to get an additional perspective, we also made use of AI tools such as ChatGPT. For more fundamental questions that affected the entire project, we consulted our expert, Manuel Bauer. In addition, for topics related to Autodesk, we were also in contact with Autodesk support.

5.7 Project Management Tools Used

At the beginning of this thesis, we discussed and evaluated many project management and collaboration tools. We required tools for version control, task tracking, communication, meetings and documentation. Since we were a small team working remotely, the chosen tools needed to be easy to use, easily accessible and integrate well with each other. That's why we selected the following tools.

5.7.1 Version Control, Issue Tracking and Time Planning

For source code management we relied on Git in combination with GitLab. Git provides version control, so every team member can work independently while keeping our source code synchronized. GitLab was used as the central collaboration platform because it offers more than just version control. We used GitLab Issues as our task management tool to track tasks, bugs and to plan our milestones. For time planning we also used GitLab, since we could track the time directly on the issues, which provided us a great overview of the time we used for each task.

5.7.2 Communication and Meetings

Since we are not in the same room every day, we had to find a way to communicate as efficient as possible. For short exchanges we used WhatsApp, which allowed us to clarify minor questions and call spontaneous meetings. For our weekly meeting we created a Discord channel. This provided us with the necessary tools such as voice & video chat as well as screen sharing so we could review code together and keep the meetings as efficient as possible. Meetings with our supervisor were held using Microsoft Teams, since it's the standardized tool for professional communication and was integrated into the infrastructure of OST.

5.7.3 Documentation

For all the documentations we wrote, except this one, such as all our architectural decision records (**"\02_Additional_Product_Documentation\05_Architectural_Decision_Records\"**), meeting notes, or other relevant documentation, we used Markdown. All three of us know markdown and use it regularly in their daily work. Markdown allowed us to write documentation and notes in a simple and lightweight format, while still being able to generate well-structured and easy to read documents. Another advantage of Markdown is, that it can be easily versioned using Git.

6 Requirement Specification

6.1 Functional Requirements

6.1.1 Must-Have Features

Authentication & Authorization

User Roles: All Users

Allow the user to register, login or reset their password to gain access to the software. Assign user roles to specific users. Optionally, we allow the user to restrict the domains that can be used for authentication by creating a domain whitelist. This feature is listed in the optional features section.

APS Authentication

User Roles: Administrator

APS credentials are needed for most of the services used in the software. Allow the user to set their APS client id, secret and scope. The APS client secret will never be visible in the frontend for security reasons. If the user wants to update the settings, he needs to paste his APS client secret to be able to save changes.

Category & Product Management

User Roles: Administrator, Developer

Allow the user to create, update and delete categories & products. Also allow the users to create versions of products.

Add App Bundles

User Roles: Administrator, Developer

Allow the user to upload app bundles to mirror actions that can be executed on the Autodesk Inventor instance.

Action system

User Roles: Administrator, Developer

Allow the user to create actions that run specified app bundles. Allow the user to assign an action to a specific product version and to specific user roles.

Default Actions

User Roles: All Users

Provide default actions so the user does not have to create the most common actions by himself. Default actions are:

- Export PDF
- Export Step
- Export BOM
- Export Native
- Calculate Cost (Pricing see below)
- Update Parameters
- Extract Parameters

Initialize Buckets

User Roles: Administrator, Developers

Allow the user to initialize the needed buckets. Buckets are Amazon S3 buckets that contain the files (CAD-Data, export formats etc.) that can be downloaded by the end user. Files can be deleted from a bucket. Also, a bucket manager is available to check what buckets exist and what objects are added to them.

Model Parameter Constraints

User Roles: Administrator, Developer

Provide an Inventor AddIn to easily describe parameters that will be visible in the configurator frontend. Allows to package a CAD-Data zip ready for upload to App-VoxelAssembler.

Configurator

User Roles: All Users

Allow the user to select a product and start the 3D configurator to this product. In here the user shall be able to run predefined actions like configuring the product or getting the BOM of his configuration. The user can set input values based on the MPC and run the actions. The user can also view the 3D model, the 2D drawing, the BOM and the exportable files if existing.

Configuration Management

User Roles: All Users

Allow the user to save, delete and share his configurations.

Account Settings

User Roles: All Users

Allow the user to change his account details and change his login credentials.

Cache Settings

User Roles: Administrator

Provide a cache system that allows to load configurations much faster. The cache notices if a configuration was already made (1:1) and returns the cached data immediately. This helps a lot for actions that take a few minutes.

Change Language

User Roles: All Users

Allow the user to change the language. Predefined languages are German and English.

User Documentation

User Roles: All Users

Provide a customer documentation for administrators and users to describe the usage of the software. Add preparation steps for how to constrain a 3D-CAD Model for VoxelAssembler. Document good practice.

iLogic Library

User Roles: Administrator, Developer

Provide a library that helps in placing models in 3D-Space.

Pricing

User Roles: All Users

For pricing many different solutions are needed. This will vary a lot depending on the customer. Prices could come from many different systems. As a base a simple solution should be implemented.

6.1.2 Optional Features

Toggle Categories & Products Visibility

User Roles: Administrator, Developer

Allow the user to control what categories and products are visible in the catalog.

3D Viewer Fullscreen

User Roles: All Users

Allow the user to display the 3D viewer in full screen mode to explore the product better.

Design Settings
User Roles: Administrator
Allow the user to set their own corporate identity including logo, company name and company colors.

Anonymous Users
Allow anonymous users to use the configurator with limited functionality.

Domain Whitelist
User Roles: Administrator
Allow the user to create a whitelist with domains that are allowed to register.

Clear Cache
User Roles: Administrator
Allow the user to clear the cached data.

Create Actions from App Bundles
Allow combining multiple app bundles to one action.

3D Viewer Extensions
User Roles: Administrator, Developer
Allow the user to enable & disable extensions of the 3D viewer for specific user roles.

Allow running instance
Keep Autodesk Inventor Instance running to provide a better performance to the user. Allow the user to toggle this and set the time how long the instance should be running in the UI.

Manage Bucket Items
User Roles: Administrator
Allow the user to display the content of a bucket as well as to delete the content or the bucket itself. Allow to check what buckets exist for used credentials.

Add native data to vault
Provide functionality to download native data and to write it to Autodesk Vault.

Configuration sidebar
Provide a collapsible sidebar to display configuration parameter. Add tabs to group the parameters.

Cloud hosting

Host the application on AWS or Azure.

Work Tasks

User Roles: All Users

Provide a system where the user can see his configurations that are currently running, succeeded, failed, pending or with any other status. Allow the user to load the configuration from there. Allow to search for certain states.

Work Tasks Administrator

User Roles: Administrator

Provide a system where the admin can see all the configurations that are currently running, succeeded, failed, pending or with any other status. Allow the Admin to load the configuration from there. Allow to search for certain states.

Load Configuration

User Roles: All Users

Allow the user to load or import a configuration by a configuration string.

3D Viewer Default Perspective

User Roles: Administrator

Allow the user to set the default perspective of the 3D model. This needs to be possible for each product version.

User Management

User Roles: Administrator

Provide an overview of all the registered users and their roles. Allow to delete a user, change its role, resend a confirmation link or directly confirm a user account.

Zip Assembly for VoxelAssembler

Provide functionality to prepare an assembly to upload it in the UI.

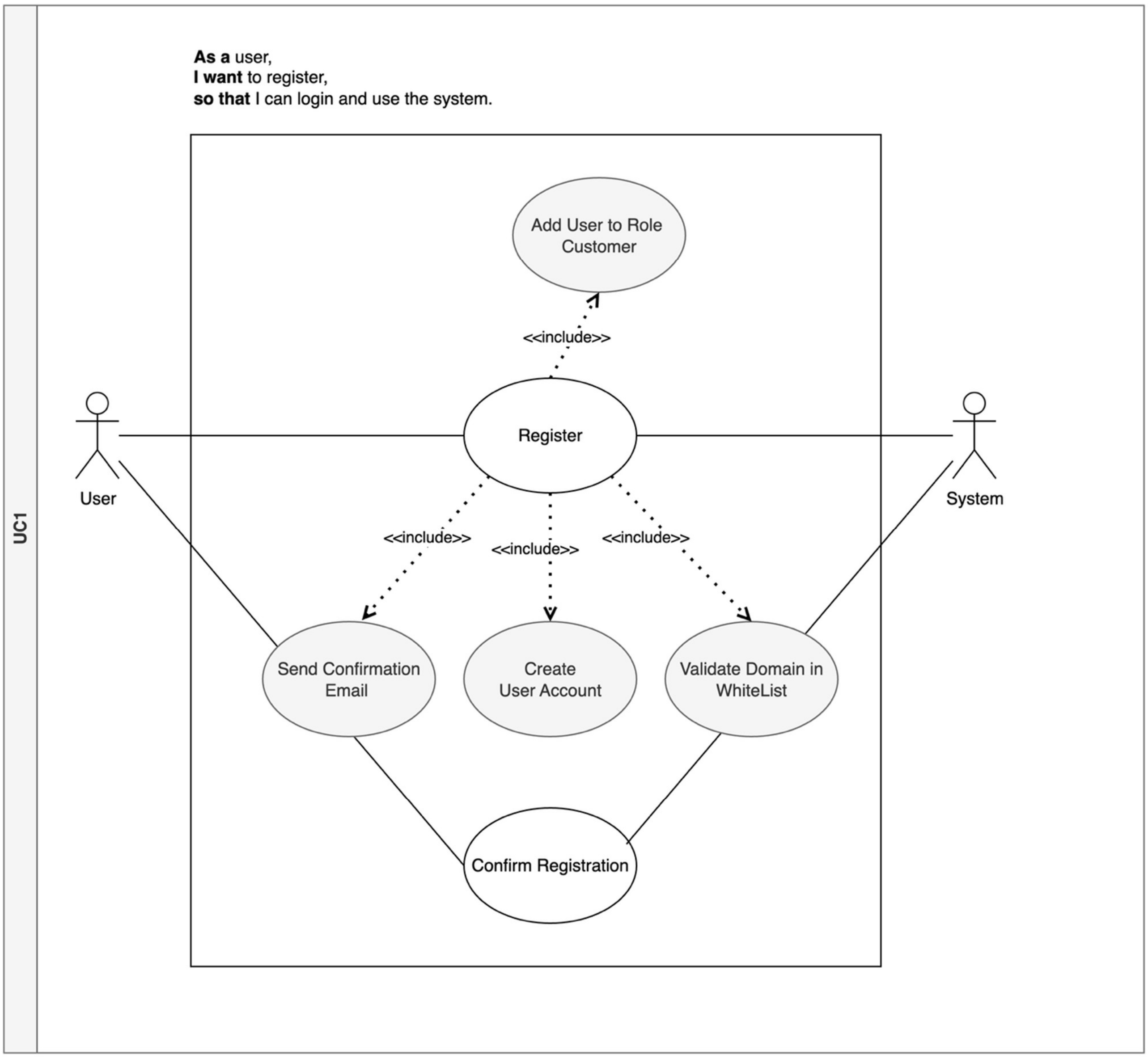
Inventor Add-In with CI/CD

Provide an easy solution to build a new version that creates an installer for customers.

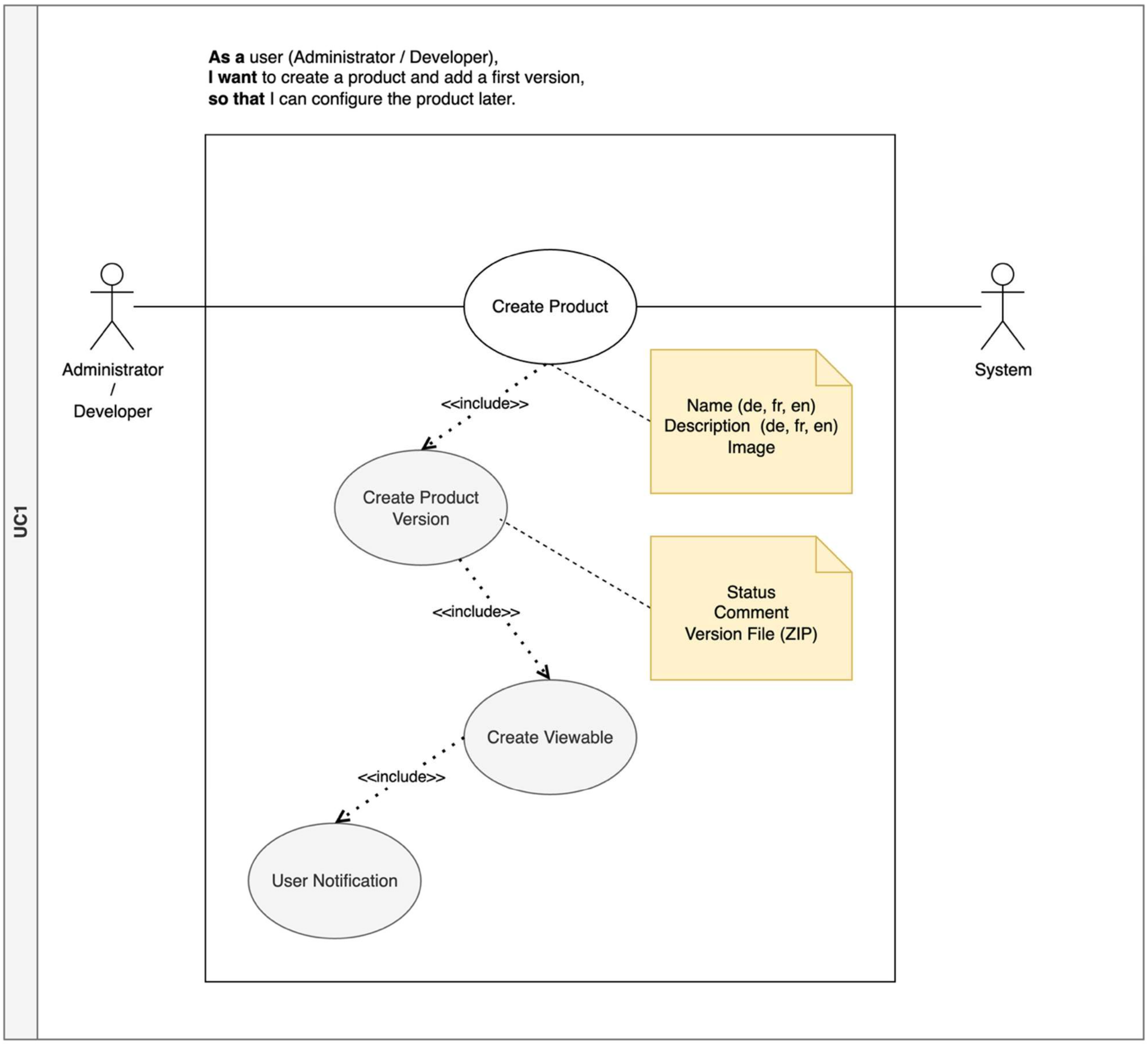
6.1.3 Use Cases / User Stories

The main documentation only covers the most relevant and central use case for the domain. Additional use cases related to the same domain are provided in the attachments (**“\02_Additional_Product_Documentation\01_Use_Cases”**). For our use case diagrams, we decided to represent them using UML notation. In addition, we followed the pattern of user stories to describe the use cases from the user’s perspective as clearly as possible.

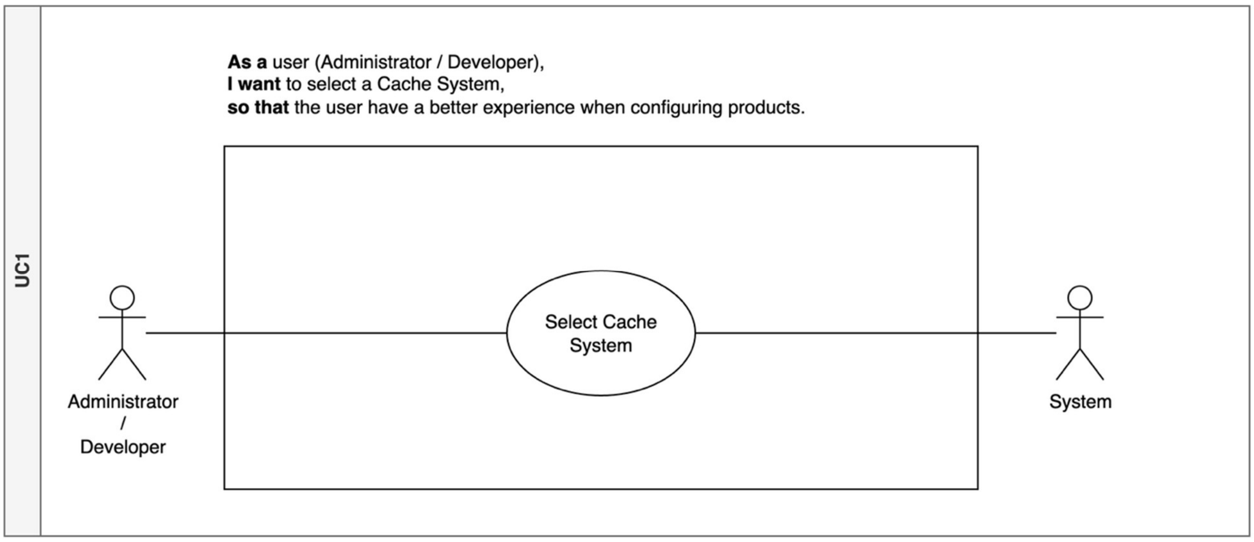
6.1.3.1 User Management



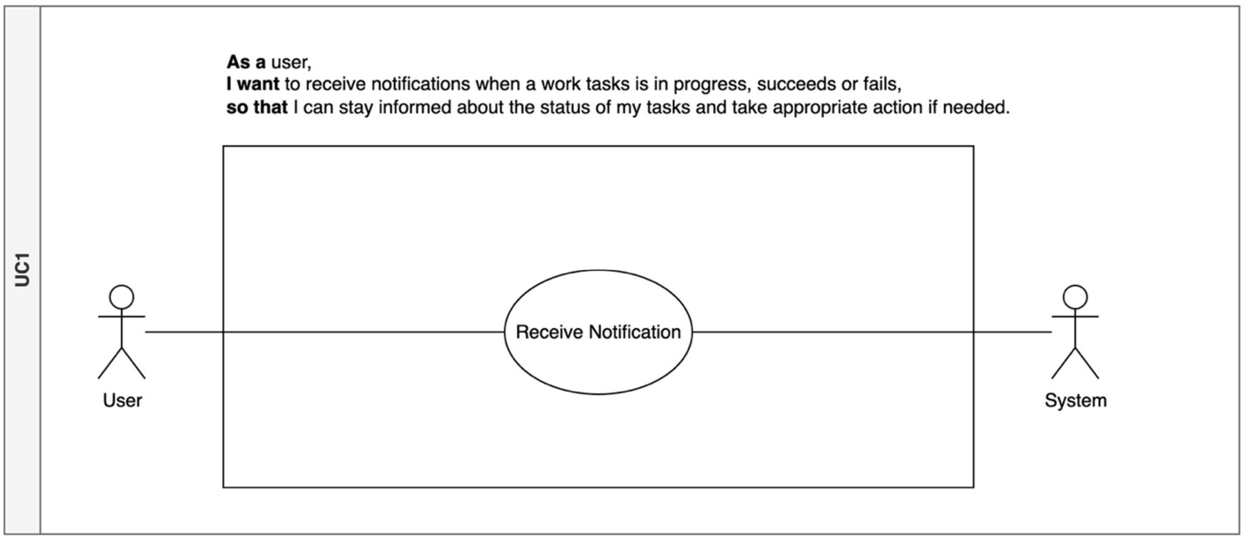
6.1.3.2 Product & Category System



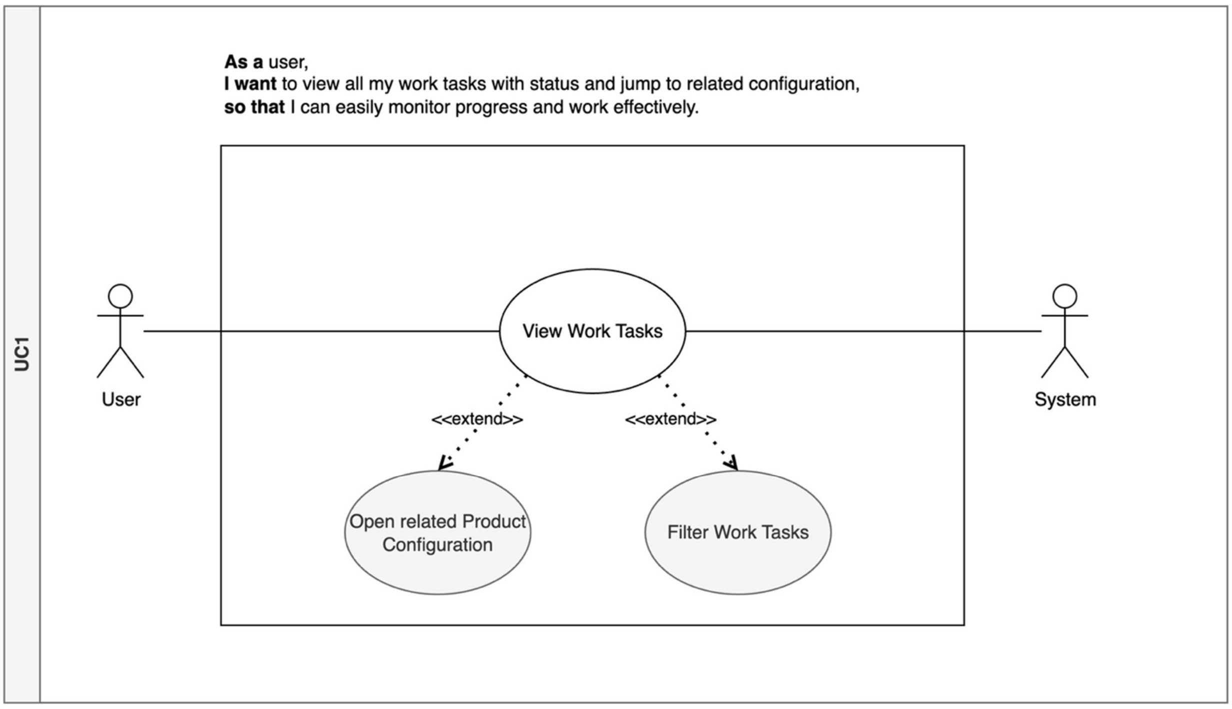
6.1.3.3 Cache System


UC1
As a user (Administrator / Developer),
I want to select a Cache System,
so that the user have a better experience when configuring products.
Administrator / Developer
Select Cache System
System

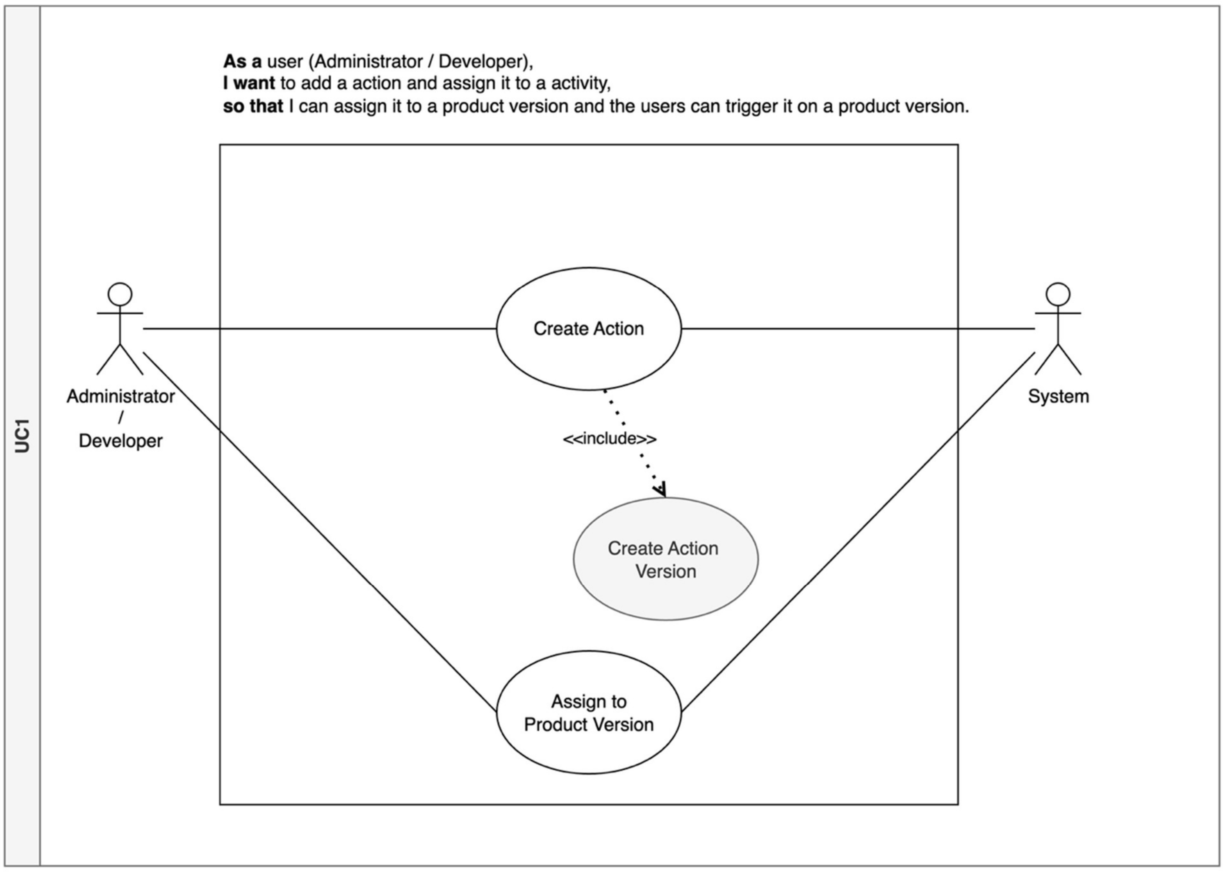
6.1.3.4 Notification System


As a user,
I want to receive notifications when a work tasks is in progress, succeeds or fails,
so that I can stay informed about the status of my tasks and take appropriate action if needed.

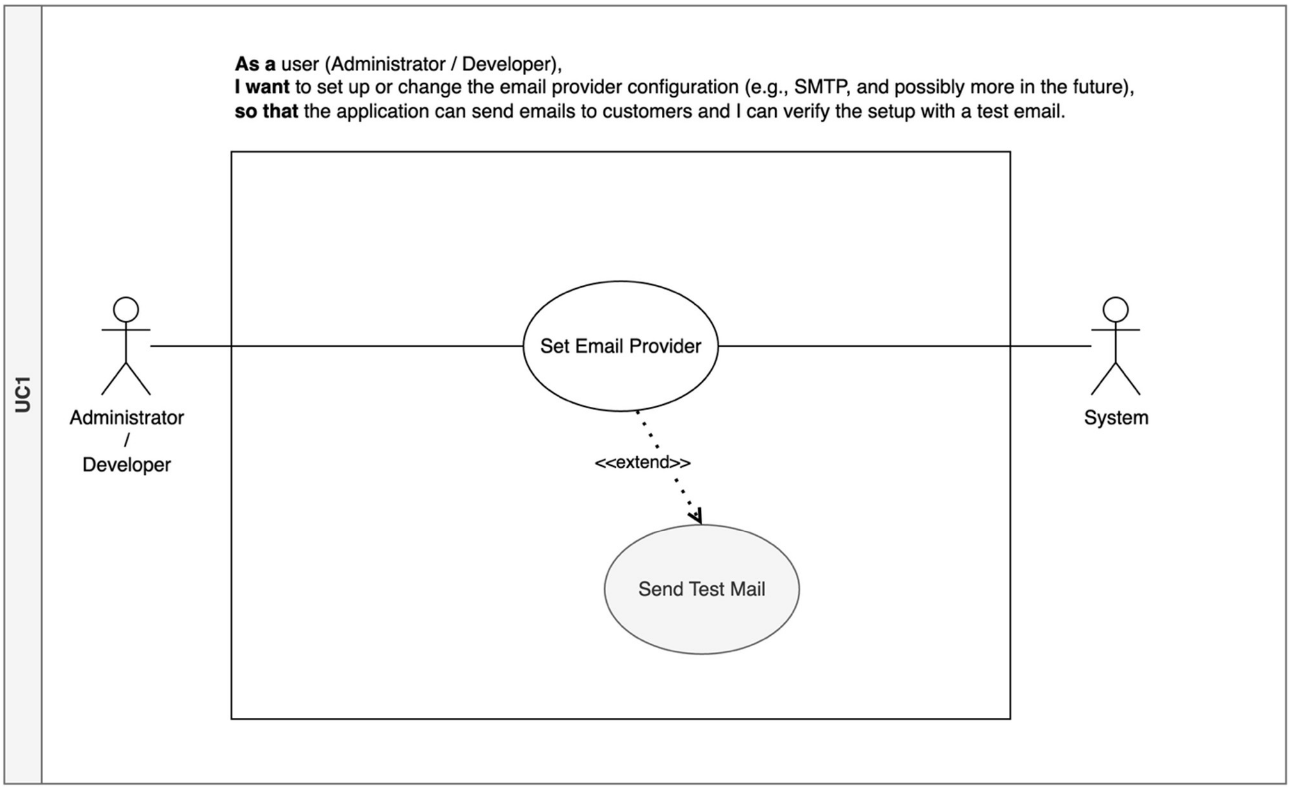
6.1.3.5 Work Tasks


As a user,
I want to view all my work tasks with status and jump to related configuration,
so that I can easily monitor progress and work effectively.

6.1.3.6 Action System

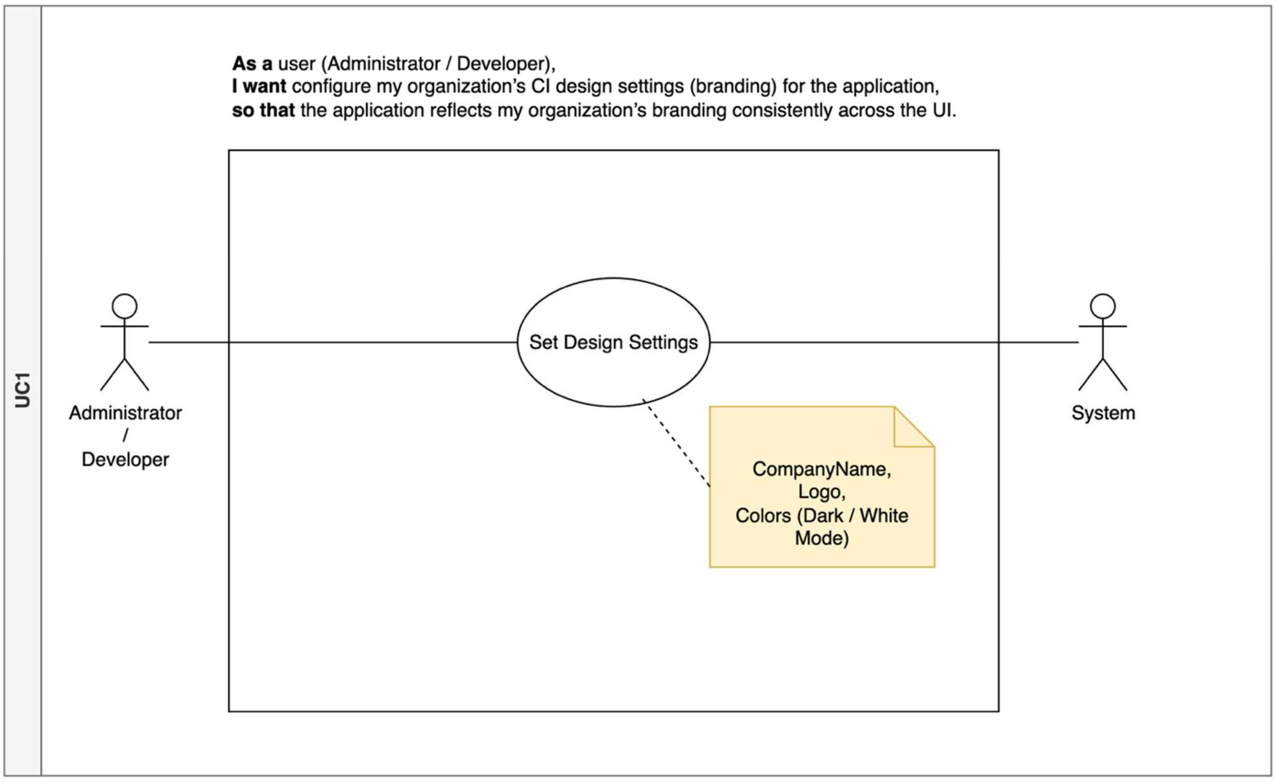

UC1
As a user (Administrator / Developer),
I want to add a action and assign it to a activity,
so that I can assign it to a product version and the users can trigger it on a product version.
Administrator / Developer
System
Create Action
<<include>>
Create Action Version
Assign to Product Version

6.1.3.7 Email System

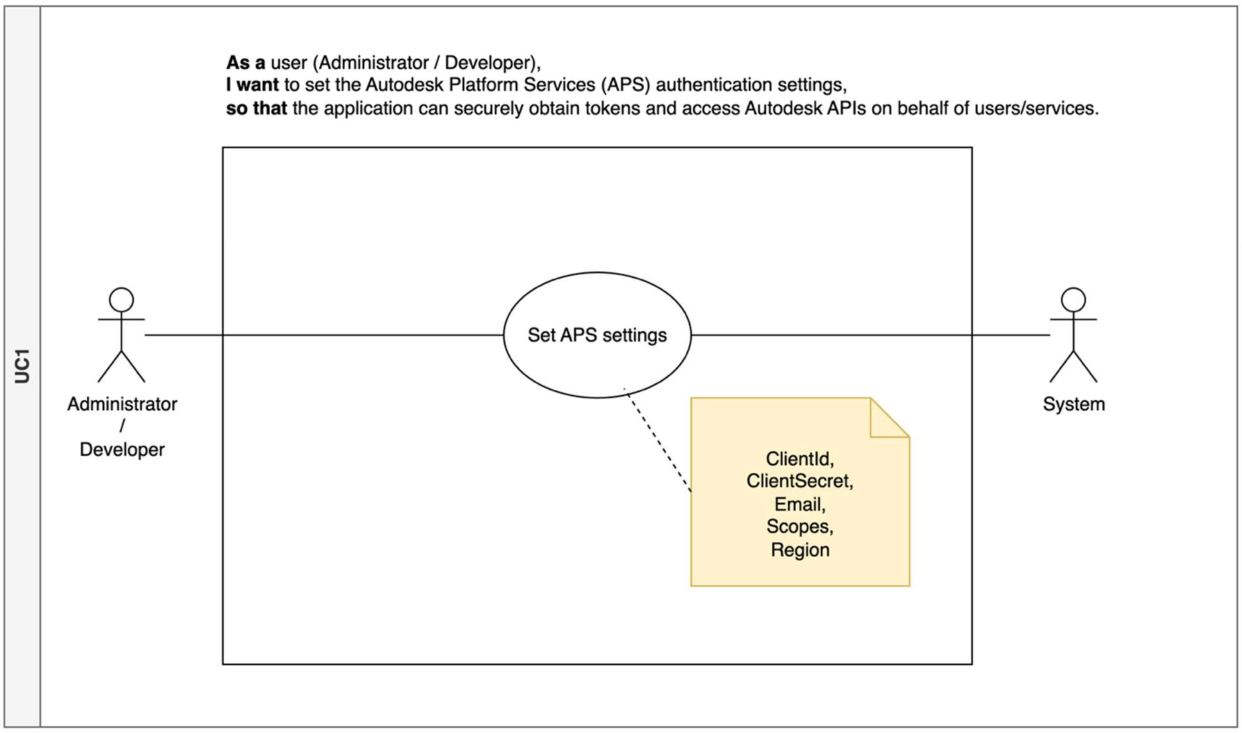

UC1
As a user (Administrator / Developer),
I want to set up or change the email provider configuration (e.g., SMTP, and possibly more in the future),
so that the application can send emails to customers and I can verify the setup with a test email.
Administrator / Developer
System
Set Email Provider
<<extend>>
Send Test Mail

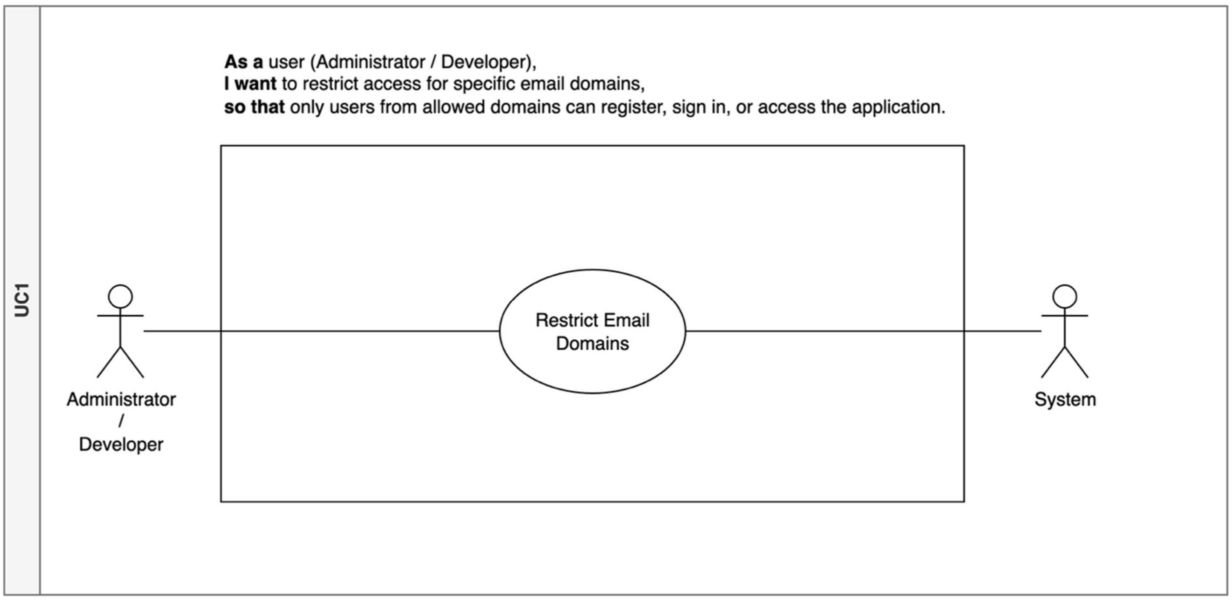
6.1.3.8 Settings

Design Settings

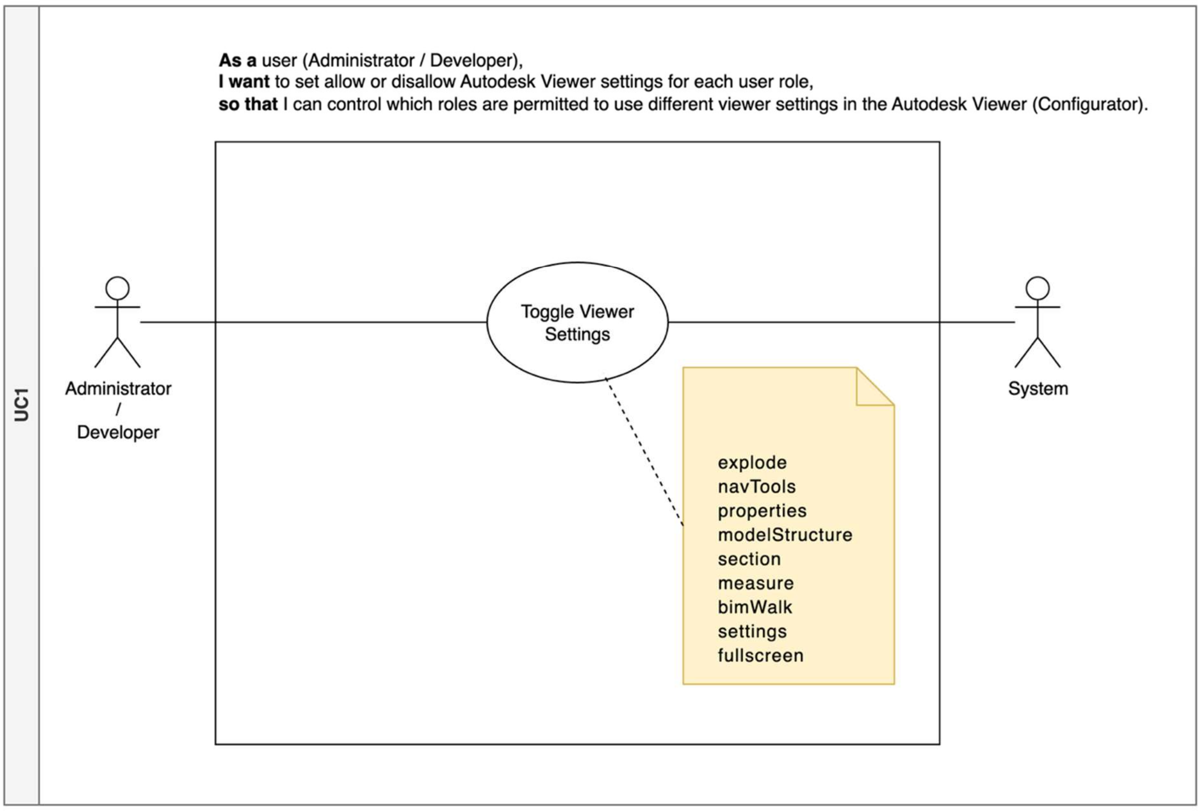

UC1
As a user (Administrator / Developer),
I want configure my organization's CI design settings (branding) for the application,
so that the application reflects my organization's branding consistently across the UI.
Administrator / Developer
Set Design Settings
System
CompanyName, Logo, Colors (Dark / White Mode)

Authentication Settings

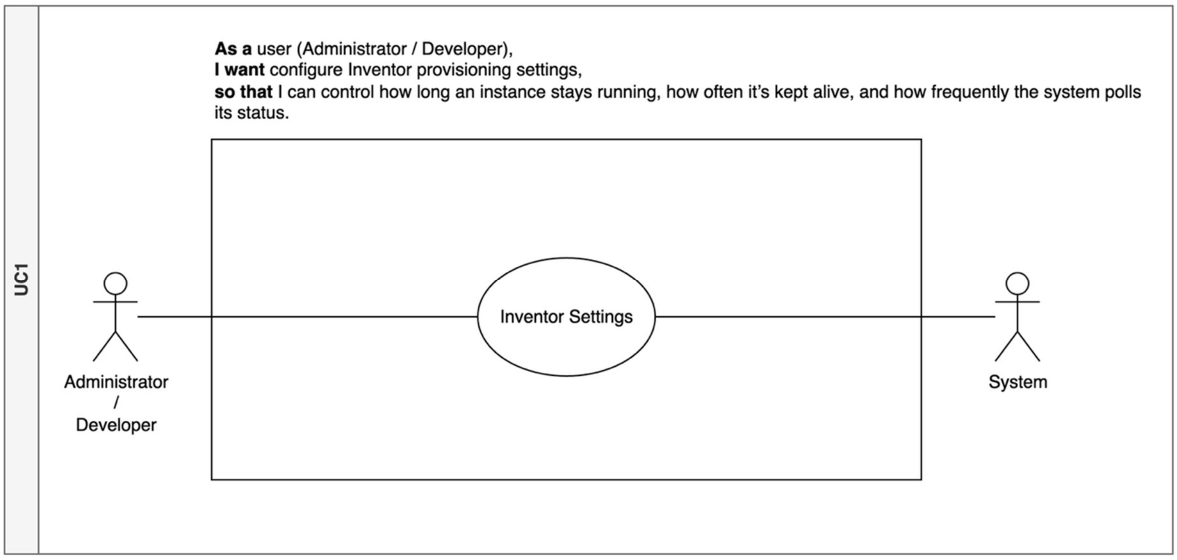

UC1
As a user (Administrator / Developer),
I want to set the Autodesk Platform Services (APS) authentication settings,
so that the application can securely obtain tokens and access Autodesk APIs on behalf of users/services.
Administrator / Developer
Set APS settings
System
ClientId, ClientSecret, Email, Scopes, Region


UC1
As a user (Administrator / Developer),
I want to restrict access for specific email domains,
so that only users from allowed domains can register, sign in, or access the application.
Restrict Email Domains
Administrator / Developer
System

Viewer Settings

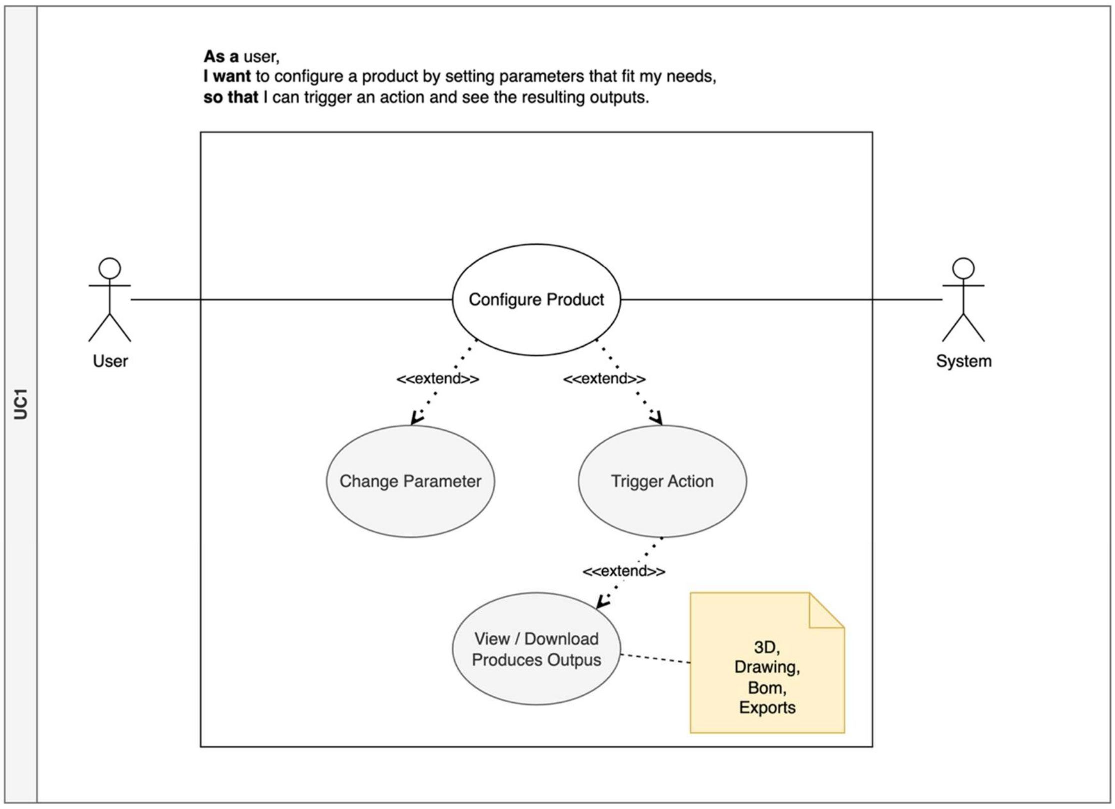

UC1
As a user (Administrator / Developer),
I want to set allow or disallow Autodesk Viewer settings for each user role,
so that I can control which roles are permitted to use different viewer settings in the Autodesk Viewer (Configurator).
Toggle Viewer Settings
explode
navTools
properties
modelStructure
section
measure
bimWalk
settings
fullscreen
Administrator / Developer
System

Inventor Settings



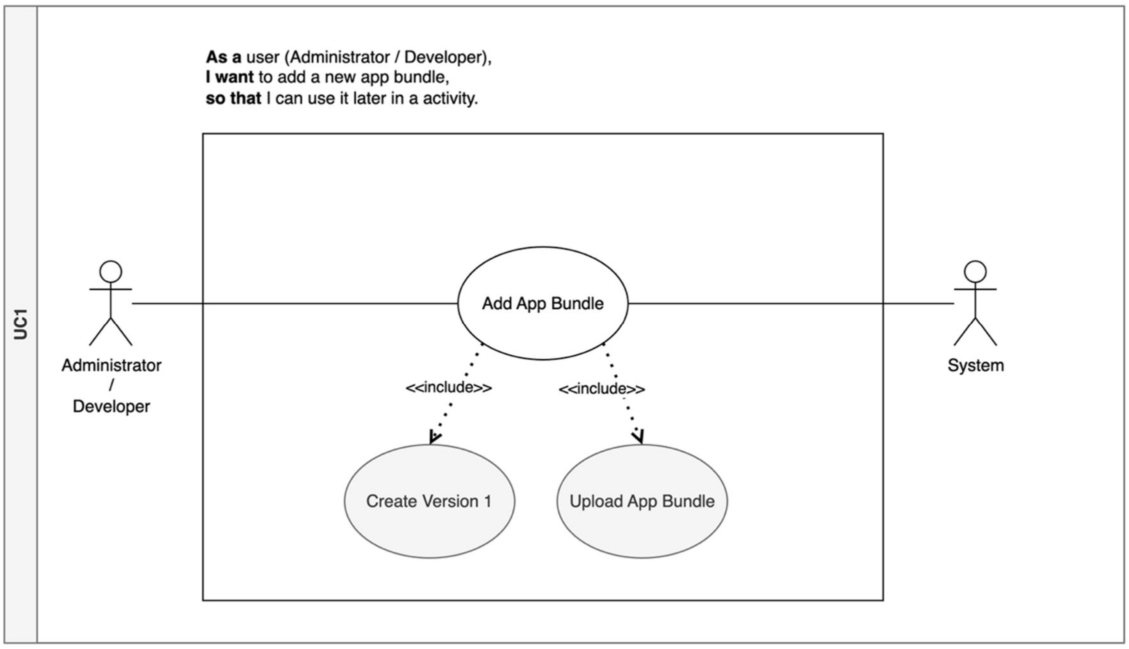
As a user (Administrator / Developer),
I want configure Inventor provisioning settings,
so that I can control how long an instance stays running, how often it's kept alive, and how frequently the system polls its status.

6.1.3.9 Product Configuration

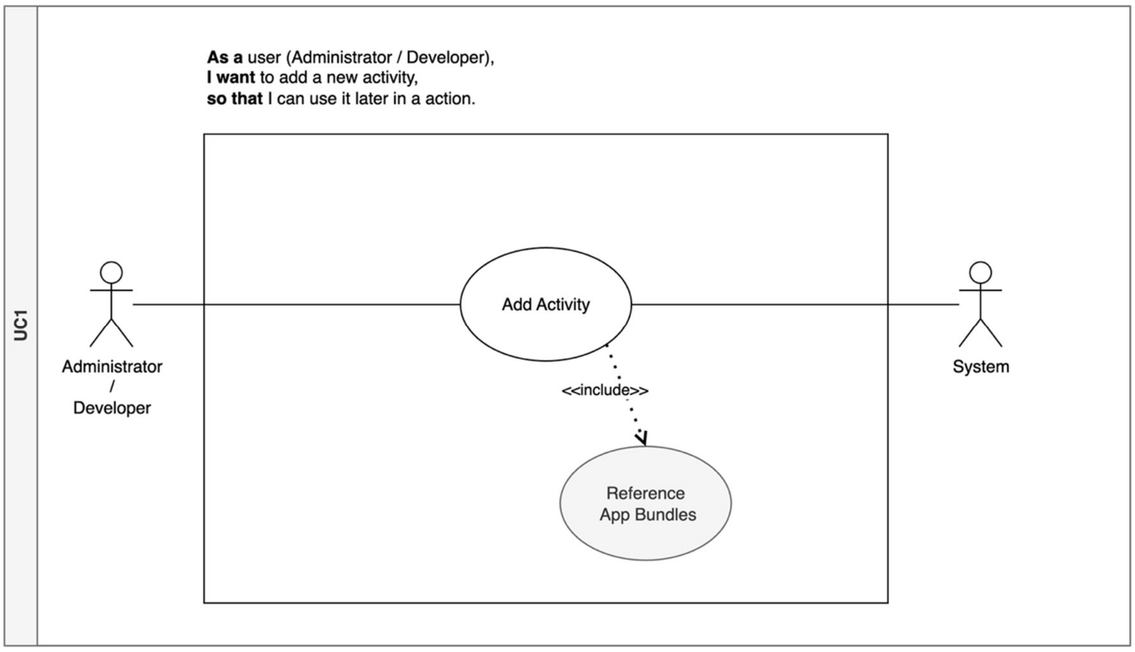

UC1
As a user,
I want to configure a product by setting parameters that fit my needs,
so that I can trigger an action and see the resulting outputs.
User
Configure Product
System
<<extend>>
<<extend>>
Change Parameter
Trigger Action
<<extend>>
View / Download Produces Outpus
3D, Drawing, Bom, Exports

6.1.3.10 Autodesk Services

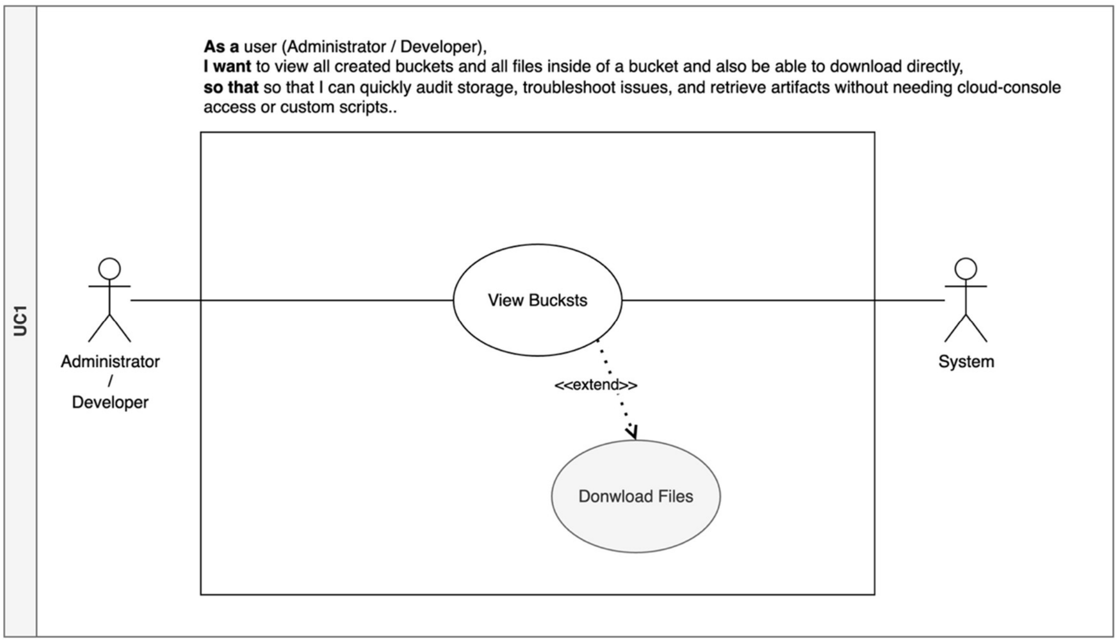
App Bundles


UC1
As a user (Administrator / Developer),
I want to add a new app bundle,
so that I can use it later in a activity.
Administrator / Developer
Add App Bundle
System
<<include>>
<<include>>
Create Version 1
Upload App Bundle

Activities


UC1
As a user (Administrator / Developer),
I want to add a new activity,
so that I can use it later in a action.
Add Activity
<<include>>
Reference App Bundles
Administrator / Developer
System

Buckets


UC1
As a user (Administrator / Developer),
I want to view all created buckets and all files inside of a bucket and also be able to download directly,
so that so that I can quickly audit storage, troubleshoot issues, and retrieve artifacts without needing cloud-console access or custom scripts..
View Bucksts
<<extend>>
Donwload Files
Administrator / Developer
System

6.1.4 User Roles

We decided on four different roles for the users. Currently some roles have almost the same possibilities but allow later for more differentiation. The following roles were implemented:

- Administrator
- Developer
- Company-User
- Customer

6.1.4.1 Administrator

There can be multiple administrators. They can manage all users and have access to all features. One administrator is by default created. The system makes sure there is always at least one administrator. It is not possible to delete the last administrator.

6.1.4.2 Developer

The developer has access to most features. A developer is someone that is managing configurations. He can upload new products and manage them. He can also create new versions and categories. Some of the features are only visible for developers. As an example, it is possible to create a new version of a product and show it only to developers until it is published for company users or customers.

6.1.4.3 Company-User

A company user is someone that is working at the company that is hosting the application. For example, salespeople could be company users. This role can be used if customers are not allowed to see each product / version.

6.1.4.4 Customer

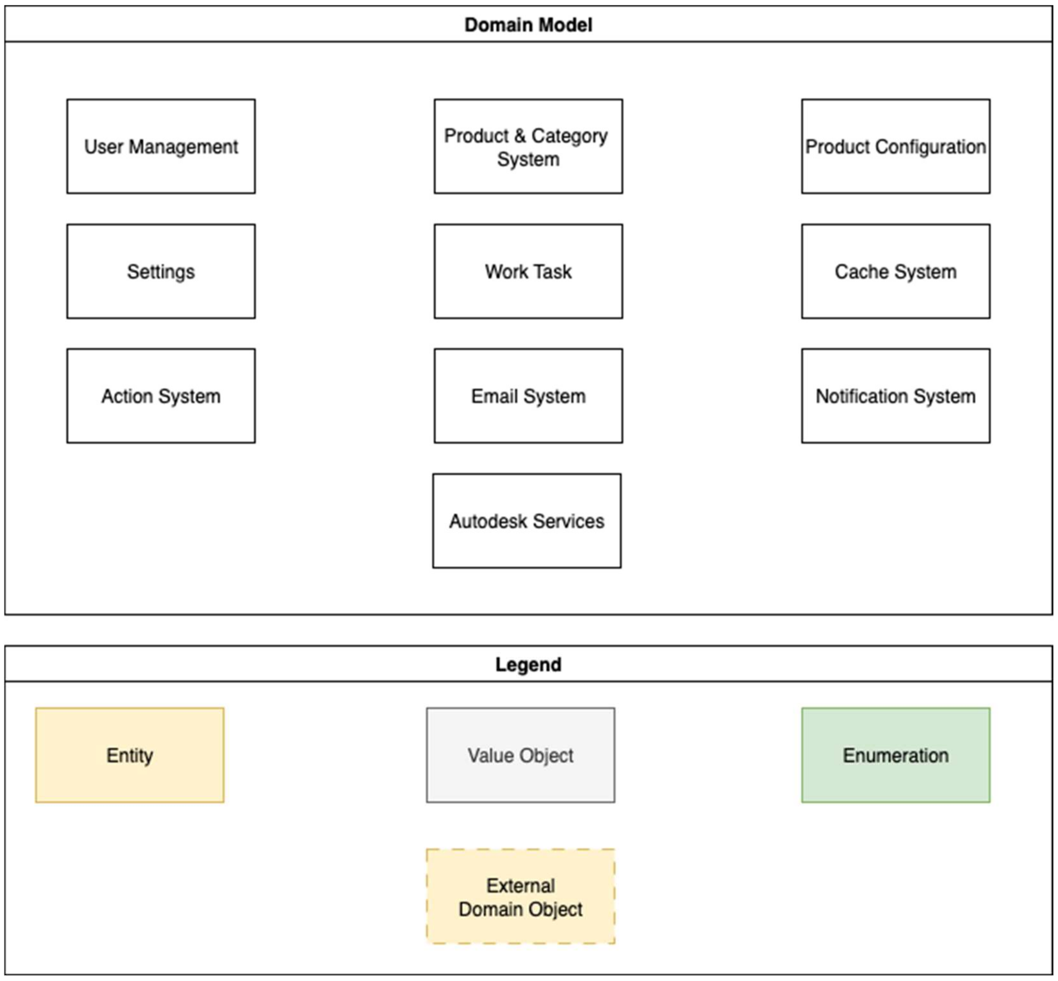
A customer is an external user from a different company than the one that is hosting the app. This role can be used if some product / versions should be only visible to company users.

6.2 Non-Functional Requirements

Aspect	Requirement (Summary)	Target / Acceptance Criteria
Usability	Consistent UI components; role-based visibility; localization; intuitive navigation	< 3 clicks from login to first configurable product; language switch updates UI and messages; Two available languages
Availability	Background jobs for long tasks; resilient real-time updates	System needs to stay responsive while heavy load, scalability must be possible from small to large customers
Robustness	Input validation; isolation of features (vertical slices); retry strategy for background jobs, good logging	Invalid requests return standardized error response; no unhandled exceptions reach client
Security	Cookie-based auth; role-based authorization; least privilege (separate read/write paths); sanitized errors	Unauthorized -> 401/403; no sensitive data in error payloads
Logging & Observability	Structured logs for jobs; minimal noise on success; detailed failure traces	Each job traceable end-to-end; failed export diagnosable from logs alone
Maintainability & Extensibility	Vertical slice architecture; pluggable actions/exports	Add new action or export by adding one slice + single registration point
Quality Assurance	Unit + integration tests; code reviews; automated pipeline checks	Critical domain services ≥ 60% line coverage; each write endpoint has at least one integration test
Localization	Backend + frontend localization; invariant codes with localized messages	Language change reflects in UI and validation without logout
Scalability (Future-Oriented)	Logical separation read/write; background workload isolation	Horizontal scaling possible without redesign
Configuration Management	Versioned products and parameters; immutable history for released versions	Past configurations fully reproducible

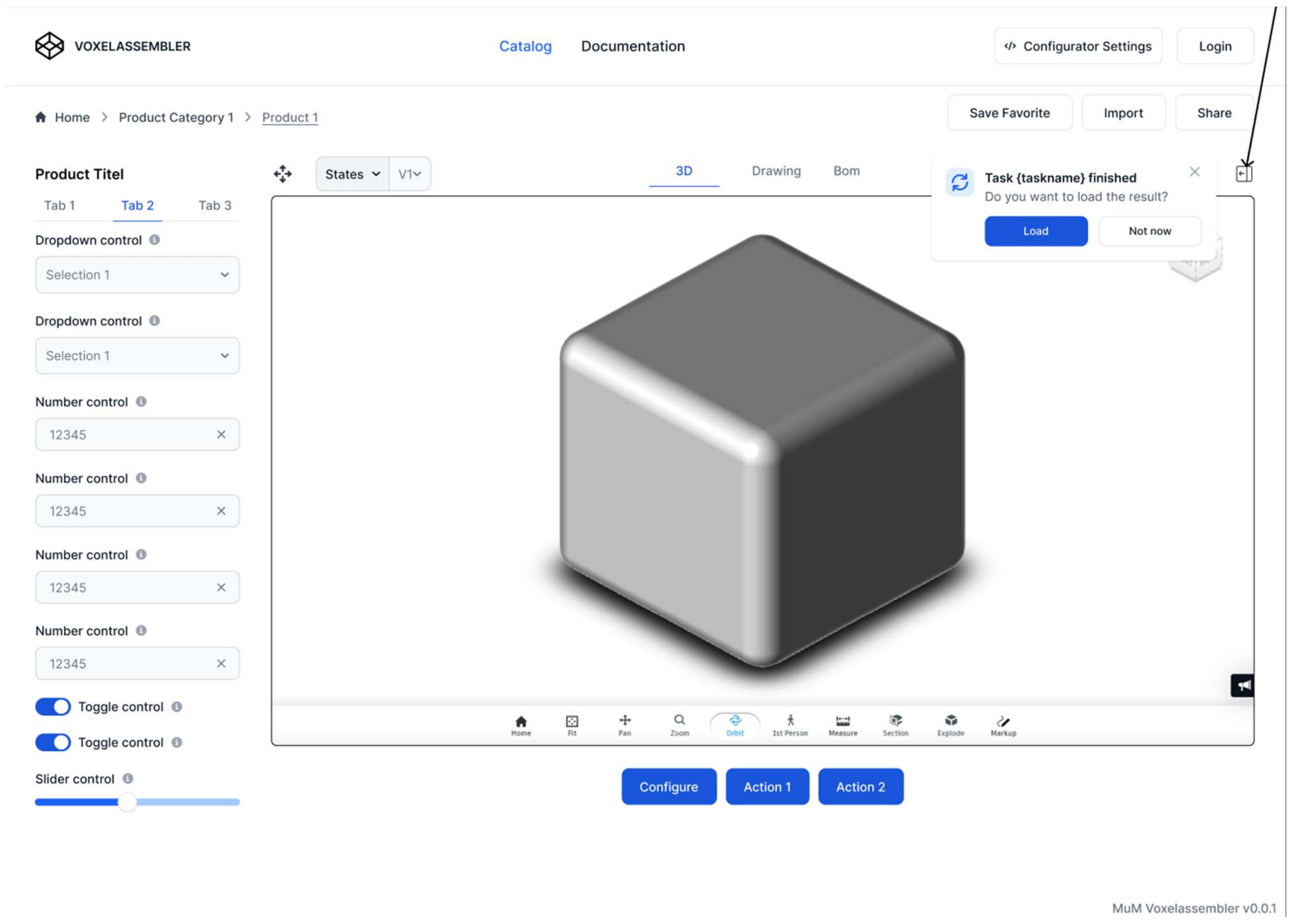
7 Domain Model

Instead of designing one large domain model, we decided to divide our domain into several sub-domains. This approach makes the model easier to understand and manage. The following figure shows all domain sub-domains at a high level. In the subsequent sections, we will go into more detail about each sub-domain, explaining how the individual entities and value objects are used. When modeling the domain, we incorporated certain concepts from Domain-Driven Design (DDD). However, we did not follow the methodology strictly in every aspect but rather applied the concepts where they provided clear value. To improve readability and consistency, we also follow a color scheme in our diagrams. Entities are always yellow, value objects are always gray, and Enums are always green.
In addition, special objects that are not directly part of the domain are shown with dashed outlines. It is important to note that the domain model does not represent exact database relationships. For details about those, please refer to the ER diagram.
In the legend of the figure, the corresponding domain business rules are documented.



7.1 Designs

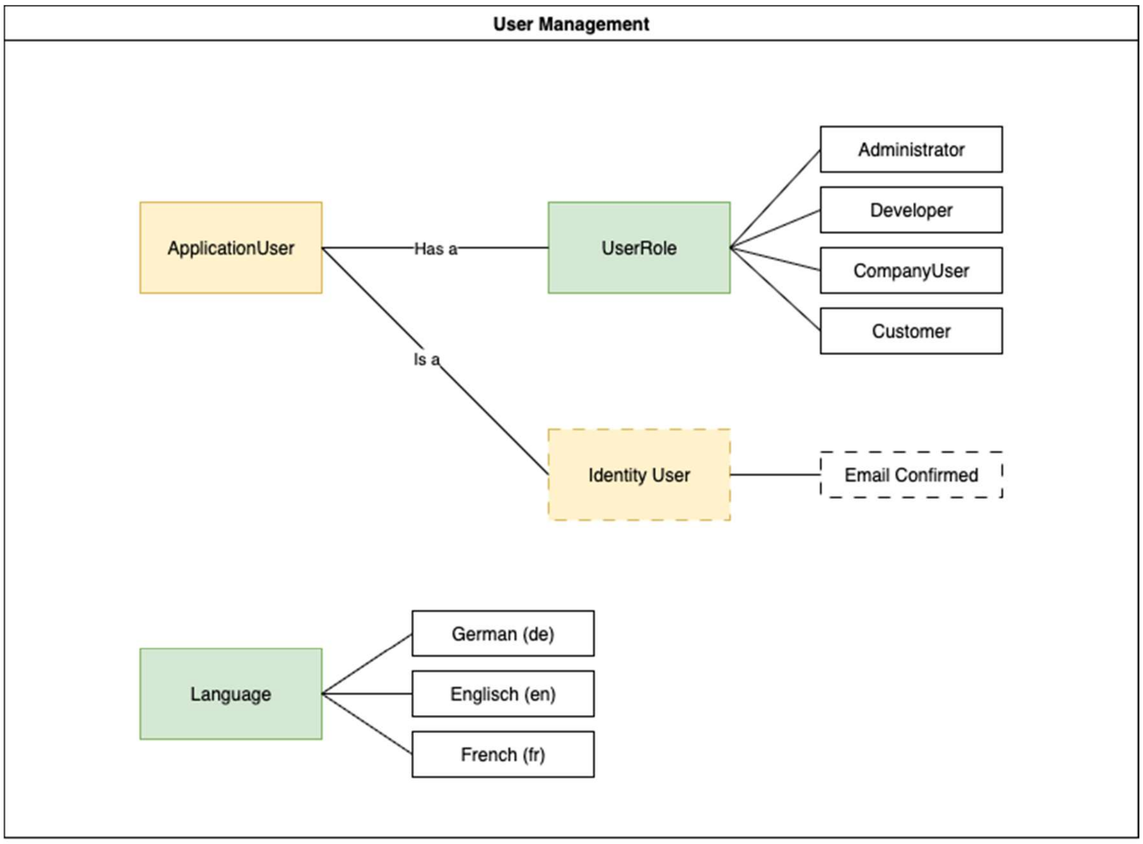
For this product a visual appealing interface is important. To address this and for easier communication what needs to be built, we designed each page beforehand. This way it was a lot of JDI (Just do it) in the frontend. All the pages can be found in **“\02_Additional_Product_Documentation\03_Design_Pages”**. Below is an example page designed in Figma.



Task {taskname} finished
Do you want to load the result?

7.2 User Management

In the User Management domain, we handle all aspects of user administration. Our ApplicationUser entity derives from the IdentityUser provided by the .NET Identity framework and extends it with additional properties. One important extension is an Enum representing our user roles. This defines the different roles a user can have within the system. In addition, this domain also includes Language, which represents the application language. It centrally manages which languages are available and can be selected in the UI. Since both UserRole and Language are closely tied to the backend domain, the backend is also responsible for providing translations for these Enums. The exact domain rules and constraints are documented in the legend of the domain diagram.



Legend

A **ApplicationUser** has a **UserRole**.

A **Administrator** / **Developer** has the highest privilege in the Application.

UserRole is an enumeration.

Language is an enumeration.

The **Language** represents the supported languages of the Application.

A **ApplicationUser** is a **IdentityUser**.

The **IdentityUser** is part of the .NET Identity Framework.

A **ApplicationUser** is only allowed to access the Application when the Email is confirmed.

7.3 Product & Category System

The Product and Category System define how products are structured and made available to users At the top level, the application exposes a Catalog. While the catalog itself is not a true domain entity, it serves as a representation of how categories and products are presented to the user. A Category can contain either subcategories or products directly, allowing for a hierarchical structure of categories. Products are not limited to a single category, instead, a Product can appear in multiple categories at the same time.

7.3.1 Product Version

Every product must have at least one Product Version. A product version belongs to a Category and carries a Version Number, which is incremented whenever a new version is created. For each product, there can be at most one version that is marked as active. In addition, every product version must always include a Viewable, which represents the product's visible or accessible form. Even if this viewable is initially only a placeholder and not yet valid, it must exist to ensure the product version is complete. The exact domain rules and constraints are documented in the legend of the domain diagram.

7.3.2 Category Version

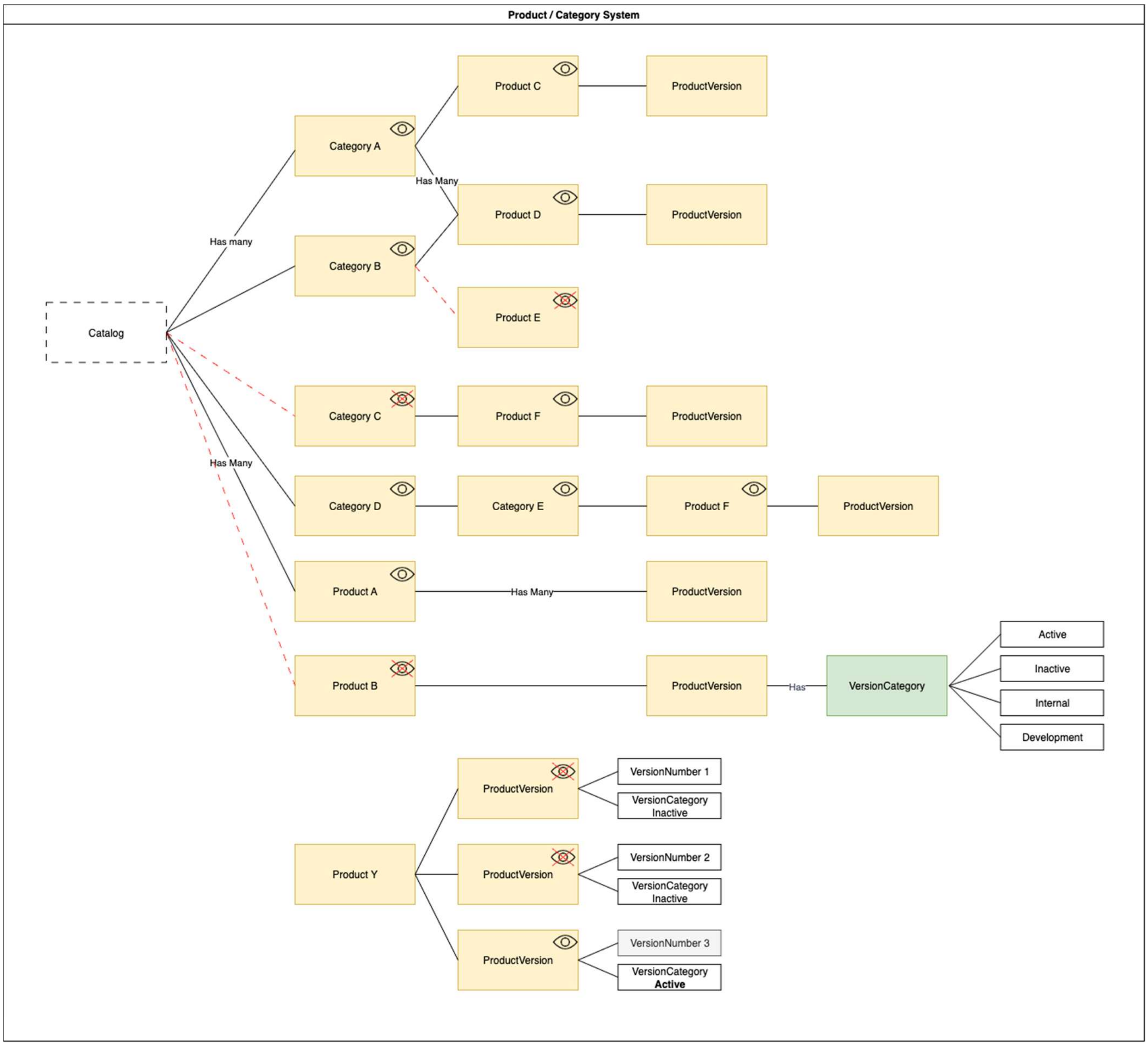
A Product Version always belongs to a Version Category. In this case the version is more of a type Version and not used for any traceability. There are four possible values:

- Active – The currently published and visible version of a product.
- Inactive – Versions that have been retired and are no longer visible in the catalog.
- Internal – Versions that are visible only to administrators. They are used by product developers to run early tests without publishing to all users.
- Development – Similar to Internal, these versions are visible only to administrators and support ongoing development and experimentation.

The visibility of product versions depends on the type of user role:

- Admin users can see products that include versions marked as Internal and Development, in addition to Active versions.
- Customer and Company users can only see Active versions.

Inactive versions are never visible in the catalog. There is also an important lifecycle rule: When an administrator adds a new Active version to a product, the previously active version is automatically downgraded to Inactive. This ensures that at most one version per product is marked as Active at any time.



Legend

A **Product** has a **Product Version**.

A **Product Version** has a **Version Category.**

A **Product** can be in one or multiple **Categories**.

A **Product** can exits as a **root Product** without a **Category**.

A **Category** can have **Products** or other **Sub Categories**.

The **Catalog** shows per default the root level of the tree. Means r**oot Categories** and **root Products** are shown.

In the **Catalog** we can traverse down to next level. So that means if the User selects **Category A**. **Category A** becomes the new root and only **Product C** and **Product D** are shown.

A **Category** and **Product** has a **Visible Flag**. Only **Visible Categories** or **Products** are shown in the **Catalog**. If a **Category** is not visible, the whole **Category** will not be shown in the **Catalog** even if there are visible **Products** inside.

For a **Category** to be shown in the **Catalog** it must contain at least one **visible Product**.

For Role **Administrator** or **Developer** also **not Visible Products** and **Categories** are Visible inside the **Catalog**.

For Role **Administrator** or **Developer** also **ProductVersions** with **VersionCategory Development** or **Internal** are visible inside the Catalog.

For Role **Customer** or **Company** only **ProductVersions** with **VersionCategory Active** are visible.

Every **Product** can only have at most one **ProductVersion** with **VersionCategory Active**.

- - - - Not Visible

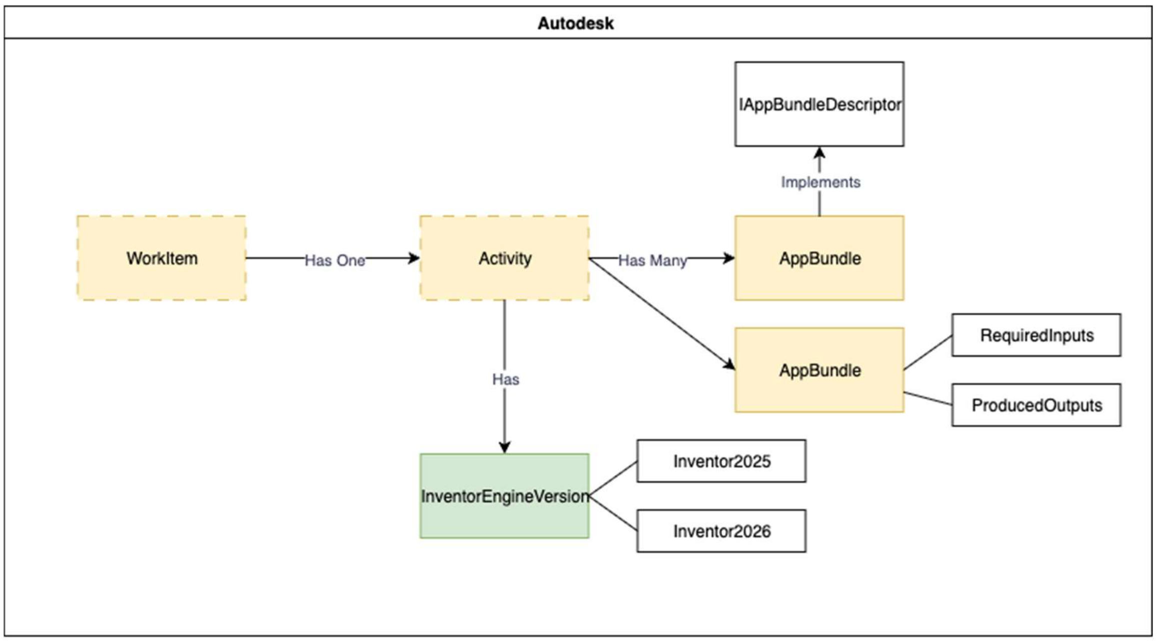
——— Visible

7.4 Autodesk Automation API / Primary entities

There are four primary entities:

- Activity
- WorkItem
- AppBundle
- Engine

From the documentation the relations between the entities are defined like the diagram below.



Legend

A **WorkItem** has one **Activity**.

A **Activity** has one or many **AppBundles**.

A **AppBundle** has one or many **RequiredInputs**.

A **AppBundle** has one or many **ProducesOutputs**.

A **Activity** has as **InventorEngineVersion**.

InventorEngineVersion is a enumeration.

Each of the entities is described below.

7.4.1 AppBundle

An AppBundle is used for running functionality on the cloud. In the case of VoxelAssembler an AppBundle is an Inventor-Addin that runs some functionality. In most cases this is a single functionality. Multiple AppBundles can be used together by combining them with an Activity.
Currently the following AppBundles are available:

- BomExtractor
- Parametrization
- PdfExporter
- PriceCalculator
- SheetMetalDxfExporter

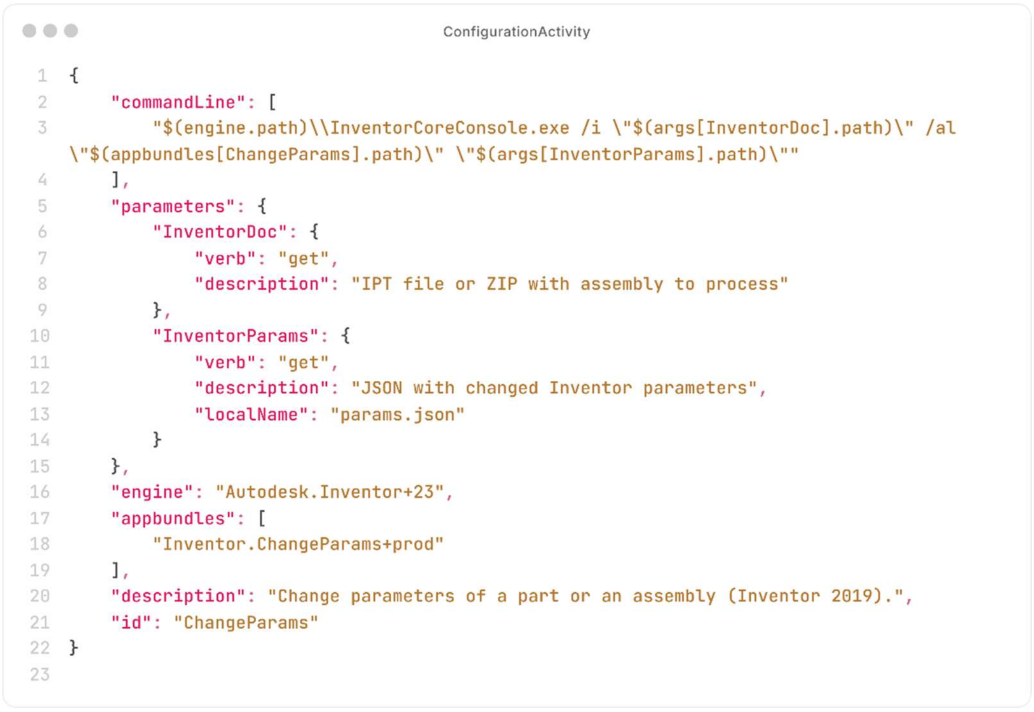
More can be easily added. Also, customer specific AppBundles are possible. VoxelAssembler provides everything needed for a customer to upload and handle new AppBundles.

AppBundles are managed by Autodesk after uploading. It is not possible to modify them once uploaded. A new version needs to be created. VoxelAssembler keeps track of each AppBundle in its Database to also keep track of the outputs each AppBundle produces. Each AppBundle implements an Interface called `IAppBundleDescriptor` which later allows to read needed In- and Output.

7.4.2 Activity

An activity is an action that can be executed within an engine. In the case of VoxelAssembler this is always Inventor. An Activity uses one or more AppBundles. Input parameters must align with the AppBundles used. There are some other parameters that can be provided like if those parameters are optional, a description, a localName, if it is a zip and others.

For example, if a configuration activity should be created, CAD-Data is needed, and the new defined parameters need to be provided. This could look something like this:



```
{
    "commandLine": [
        "$(engine.path)\\InventorCoreConsole.exe /i \"$(args[InventorDoc].path)\" /al \"$(appbundles[ChangeParams].path)\" \"$(args[InventorParams].path)\""
    ],
    "parameters": {
        "InventorDoc": {
            "verb": "get",
            "description": "IPT file or ZIP with assembly to process"
        },
        "InventorParams": {
            "verb": "get",
            "description": "JSON with changed Inventor parameters",
            "localName": "params.json"
        }
    },
    "engine": "Autodesk.Inventor+23",
    "appbundles": [
        "Inventor.ChangeParams+prod"
    ],
    "description": "Change parameters of a part or an assembly (Inventor 2019).",
    "id": "ChangeParams"
}
```

This defines that there needs to be an input document and a file that provides the parameters.

7.4.3 WorkItem

A WorkItem is a specification of the processing job for an Activity, and it is submitted to and executed by the engine. For VoxelAssembler this gets wrapped by WorkTask, which allows to add more information to it. As an example, a WorkItem must deliver an address to download the above mentioned InventorDoc. This is a S3 signed URL in our case but could be something else.
Note that a WorkItem cannot be modified after it has been created.

7.4.4 Engine

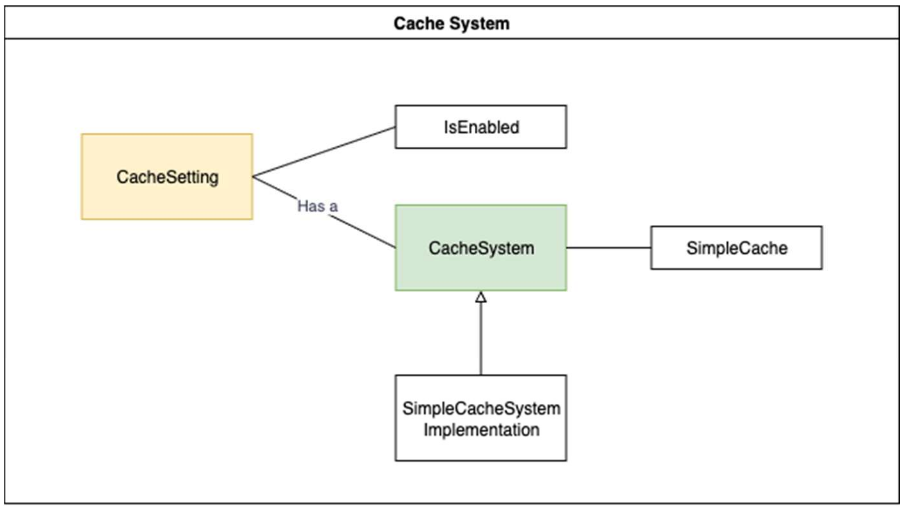
An engine executes a WorkItem. Autodesk provides multiple engines in the cloud. In our case only Inventor is used. Currently in the newest Version 2026, which can be later updated. An AppBundle must be written for a certain engine. Due to API changes problems can arise if those do not align.

7.5 Viewable

A Viewable is what is used by the viewer to show a 3D or 2D(drawing) to the user. Each product version has a default viewable that allows to display something when the product is loaded the first time. For each product configuration a new viewable is created. Depending on settings the same thing is done for a 2D-Drawing that is displayed in its own tab.

7.6 Cache System

The Cache Domain is responsible for managing caching strategies within the application. Our cache system is designed to support multiple caching strategies. Currently, only a simple cache system is implemented, which checks in the database whether a given product configuration already exists. This makes the cache persistent rather than transient like an in-memory cache. The central entity CacheSetting controls which cache system is active. For now, only the simple cache system can be selected, but additional implementations can be added in the future and activated through this setting. The Simple Cache System checks in the database whether a given Product Configuration already exists. This means the cache is persistent and not transient like an in-memory cache. The property IsEnabled controls whether the cache is active. When disabled, product configurations are always resolved directly from the source without checking the cache. To decide if something is cached the used action is considered. Depending on what the Action produces, a decision must be made if the cache has everything needed. If yes, the cache is used and if not a WorkTask is triggered.



Legend

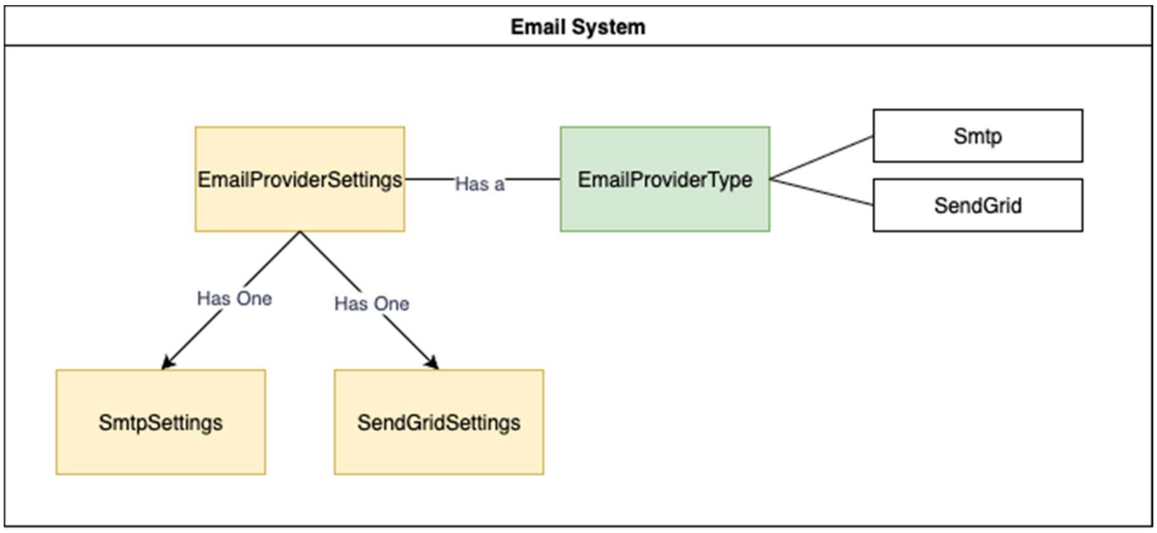
CacheSystem is an enumeration.

CacheSetting has a flag **IsEnabled** which control whether the **CacheSystem** is used or not.

Every **CacheSystem** needs a concrete implementation.

7.7 Email System

The email provider settings implement either SMTP settings or SendGrid settings, controlled by the email provider type. The email system is designed to be open for additional providers in the future and can be switched dynamically depending on the selected provider type directly through the UI. Currently only Smtp is implemented.



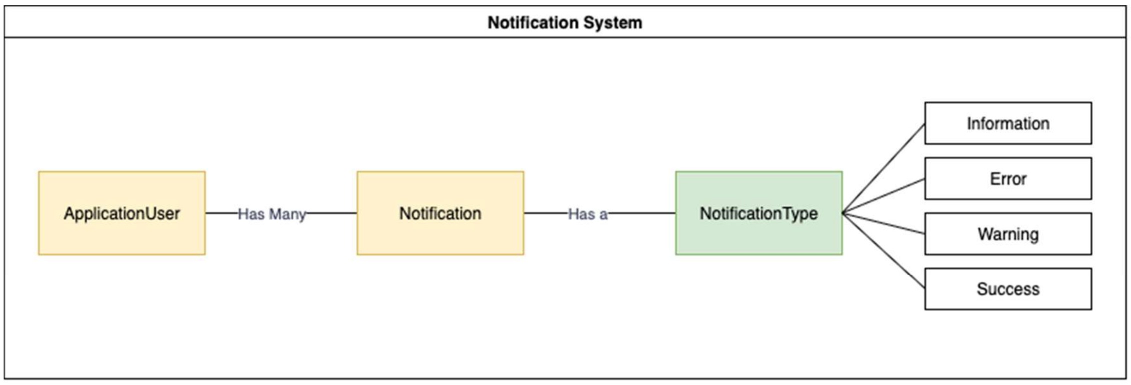
Legend

A **EmailProviderSettings** has either **SmtpSettings** or **SendGridSettings** depending on selected **EmailProviderType**.

EmailProviderType is an enumeration.

7.8 Notification System

A notification always belongs to a user. A notification can either be dismissed or marked as read. Each notification also indicates its severity level through notification type. Since the notification domain is tightly coupled to the backend, the notification texts are also translated in the backend.



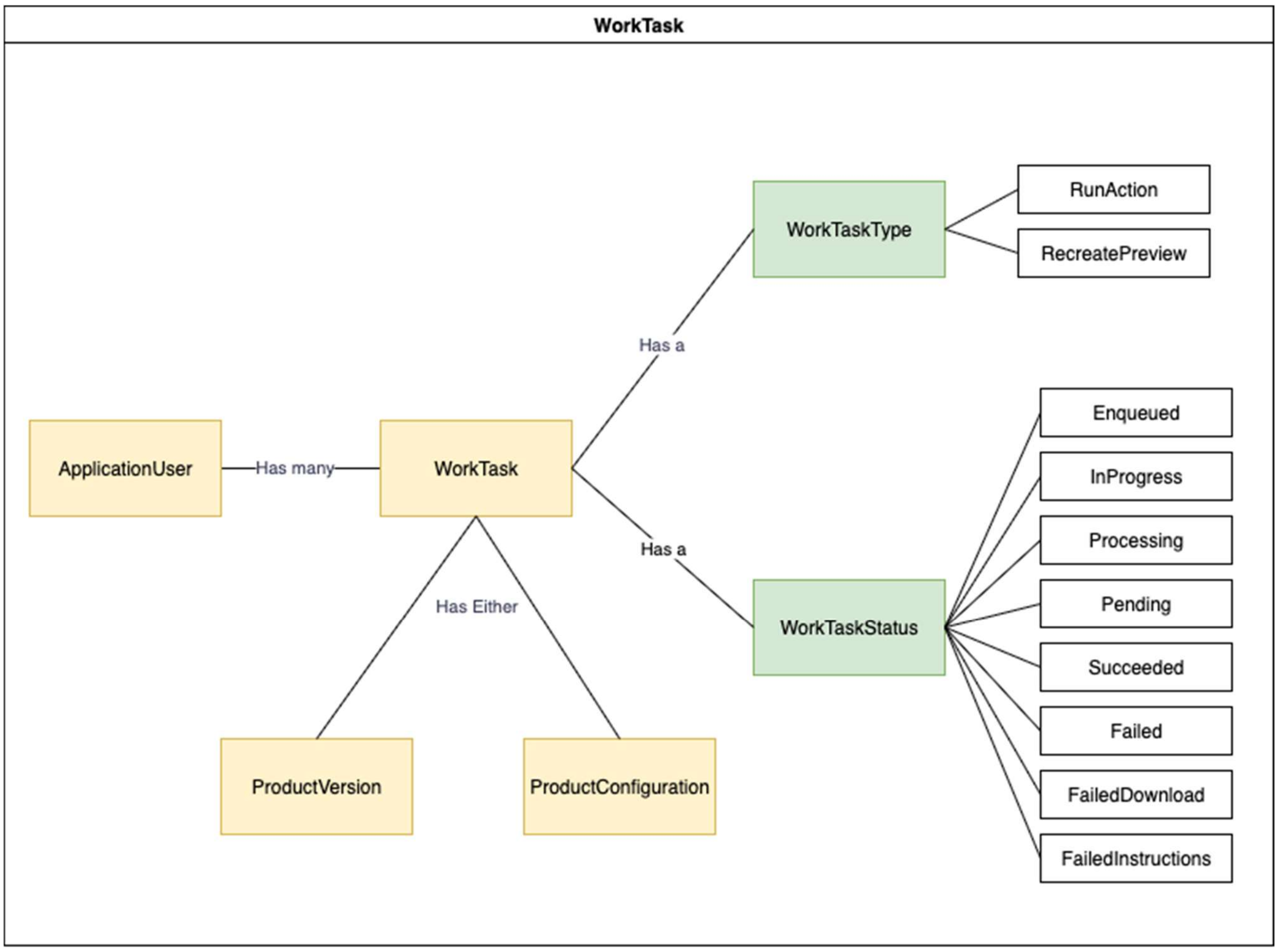
Legend

A **ApplicationUser** can have zero or multiple **Notifications**.

NotificationType is an enumeration.

7.9 WorkTask

A WorkTask is always assigned to a user. Each WorkTask must have a WorkTask type, which can be extended as needed. The WorkTask status is a combination of Hangfire statuses and Autodesk WorkItem statuses. A WorkTask must belong either to a product version or to a product configuration.



Legend

A **ApplicationUser** can have zero or multiple **WorkTasks**.

WorkTaskType is an enumeration.

WorkTaskStatus is an enumeration.

A WorkTask must be have either a **ProductConfiguration** or a **ProductVersion.**

7.10 Product Configuration

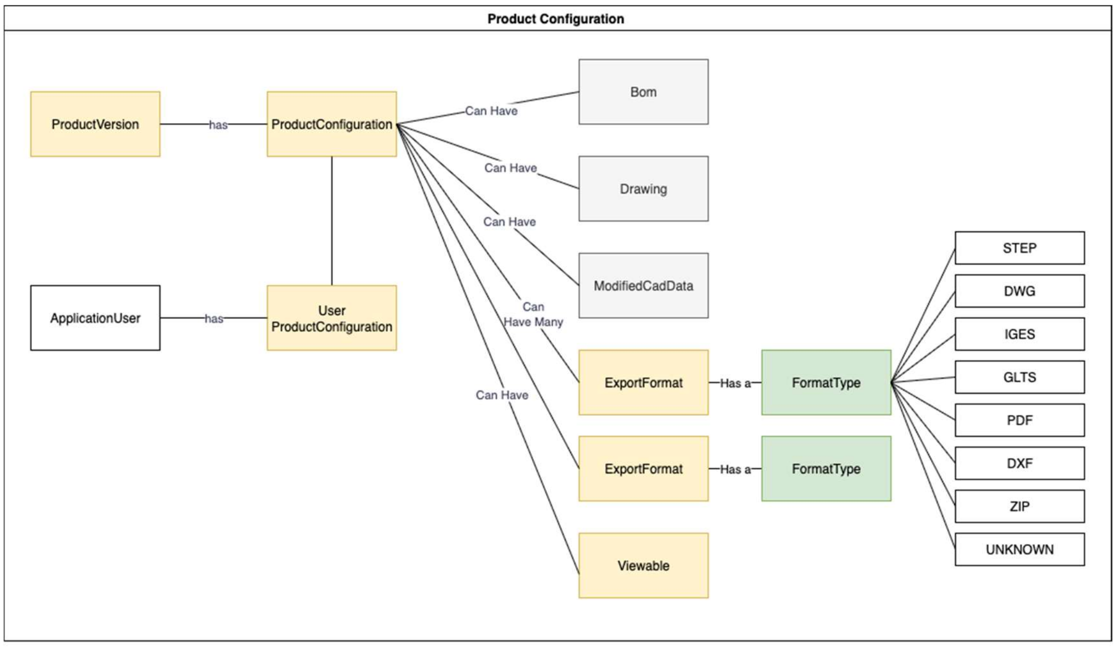
For a product version, one or more product configurations can be created. A user can save a product configuration, which then becomes a UserProductConfiguration.

Bom is short for bill of materials and describes what parts are involved in this configuration. Normally this is used for processing an order. It tells everyone what needs to be ordered or produced.

A drawing helps to understand what needs to be produced. Dimensions are used to describe what exactly needs to be produced. This can be detailed or just some external dimensions to get a rough idea about size.

ModifiedCadData is what is produced as output from running a configuration. This is the adjusted CAD-Data after applying the new defined parameters and can be used to order the product.

ExportFormat is everything else that is produced by AppBundles. For example, a dxf could be produced which is used by a laser cutting machine to produce a sheet metal part. Also, a price calculation can be exported and added as ExportFormat.



Legend

A **ProductVersion** can have zero or multiple **ProductConfigurations**.

A **ApplicationUser** can have zero or multiple **UserProductConfigurations**.

A **UserProductConfiguration** has one **ProductConfiguration**.

A **ProductConfiguration** can have zero or one **Bom**.

A **ProductConfiguration** can have zero or one Drawing.

A **ProductConfiguration** can have zero or one **ModifiedCadData**.

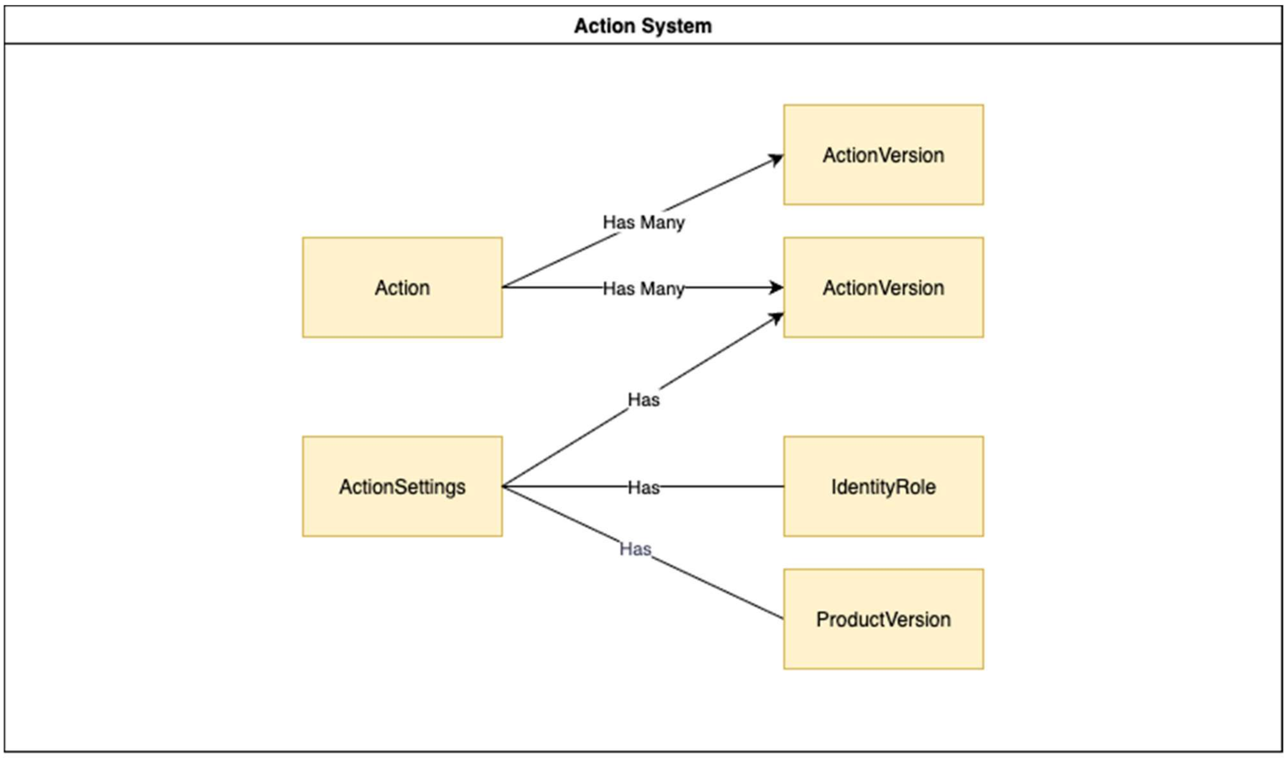
A **ProductConfiguration** can have zero or multiple **ExportFormats**.

A **ProductConfiguration** can have zero or one **Viewable**.

FormatType is an enumeration.

7.11 Action System

An Action references exactly one activity and is used to provide all the inputs needed for a WorkItem. For most of the AppBundles some settings can be provided. This is done by providing a string. What settings can be provided is documented in the customer documentation.



Legend

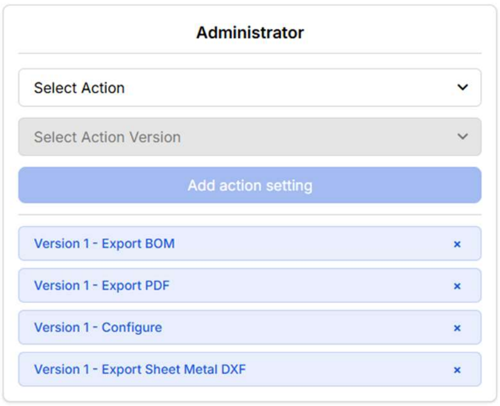
Action has one or many **ActionVersions**.

ActionSetting has one **ActionVersion**.

ActionSetting has one **IdentityRole**.

ActionSetting has one **ProductVersion**.

ActionSettings allow to define what actions are available for what UserRole. Each action and its specific version can be assigned to one of the UserRoles.



Administrator

Select Action

Select Action Version

Add action setting

Version 1 - Export BOM ×

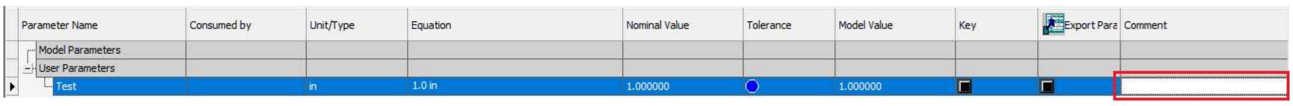
Version 1 - Export PDF ×

Version 1 - Configure ×

Version 1 - Export Sheet Metal DXF ×

7.12 Model Parameter Constraints

To allow a CAD-User (someone that knows how to use Autodesk Inventor) how the frontend should look like the concept of model parameter constraints is created. A Json string put into the parameter constraint defines how this parameter should be presented in the frontend.



Parameter Name	Consumed by	Unit/Type	Equation	Nominal Value	Tolerance	Model Value	Key	Export Para	Comment
Model Parameters									
User Parameters									
Test		in	1.0 in	1.000000	●	1.000000	■	■	

The reason why comment was chosen for this, is simple that it is normally not used by the user and it allows to put whatever in it.
Writing a Json String and doing this in the comment field isn't something that can be expected from the user. The reason for this is:

1. Most designer don't know JSON
2. Writing multiline comments is not great inside of comment

To handle this a GUI is built as an Inventor Addin.

7.12.1 Language

To allow translation in the frontend many properties use the concept of multiple language strings with semicolon (;) as the separator between. Sample: "De=Text in Deutsch;Us=Text in Englisch"
Each Parameter is marked with “Allows Language” if translation needs to be supported.

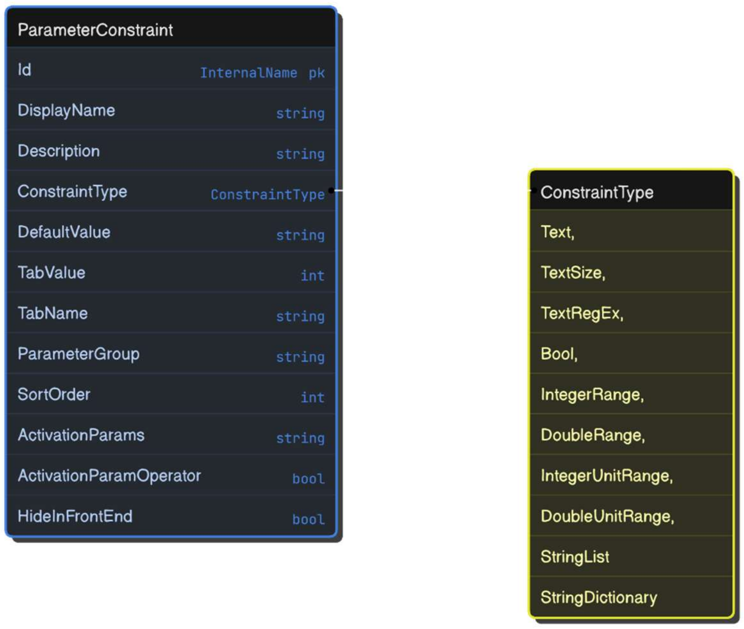
7.12.2 Constraint types

The following types are supported:

- Text
- TextSize
- TextRegEx
- Bool
- IntegerRange
- DoubleRange
- IntegerUnitRange
- DoubleUnitRange
- StringList
- StringDictionary

7.12.3 Common Properties

Each ConstraintType has the following Properties:



ID: Simple internal Name

DisplayName: What will be shown for this parameter in the frontend. (Allows Language)

Description: Description / Tooltip that is shown in the frontend. (Allows Language)

ConstraintType: One of the mentioned constraint types. Helps to decide how to handle it in the frontend.

DefaultValue: Default value for the parameter. Must be the same type as DefaultUnitId if used.

TabValue: Each parameter can be put into a TabGroup. They are sorted by this value.

TabName: What name should be displayed for the tab. (Allows Language)

ParameterGroup: What name should be displayed for the group. (Groups inside of tab for further structuring) (Allows Language)

SortOrder: In what order are the parameter groups.

ActivationParams: Parameters that are checked for true/false and then decide if the parameter is visible or not. Can be multiple parameters separated by;. All parameters need to be of type bool.

ActivationParamOperator: Decides if all the parameters need to be true or if it behaves like an or. If it is True, then it behaves like an and. All the parameters need to be True.

HideInFrontend: If True then the parameter will not be shown in the frontend but can be used for other things like as an activation param.

7.12.4 Text

Text does not have any additional properties.

7.12.5 TextSize

The Text Size Constraint Definition (ConstraintTypeId: TextSize) restricts the length of the parameter value.

TextSize has the following additional properties:

MinCharacters: Minimum length of the text.

MaxCharacters: Maximum length of the text.

7.12.6 TextRegEx

The Text RegEx Constraint Definition (ConstraintTypeId: TextRegEx) restricts the parameter value with a RegEx pattern.

TextRegEx has the following additional properties:

RegExPattern: Regular expression pattern that the text needs to match.

7.12.7 Bool

The Boolean Constraint Definition (ConstraintTypeId: Bool) restricts the parameter value to true or false, 1 or 0 and defines the type of control.

Bool has the following additional properties:

ControlType: Defines the type of control. It can be CheckBox or RadioButton.

BoolValueType: Defines the type of value that is stored. Can be TrueFalse or OneZero.

OptionTrueDisplayName: What should be displayed if the value is true. (Allows Language)

OptionFalseDisplayName: What should be displayed if the value is false. (Allows Language)

7.12.8 IntegerRange

The Integer Range Constraint Definition (ConstraintTypeId: IntegerRange) limits an integer value to a specific range, and, optional, a specific interval.

IntegerRange has the following additional properties:

MinValue: Minimum value of the range.

MaxValue: Maximum value of the range.

Interval: Interval of the range.

ControlType: Defines the type of control. Can be Slider or TextBox.

7.12.9 DoubleRange

The Double Range Constraint Definition (ConstraintTypeId: DoubleRange) limits a double value to a specific range, and, optional, a specific interval.

DoubleRange has the following additional properties:

MinValue: Minimum value of the range.

MaxValue: Maximum value of the range.

Interval: Interval of the range.

ControlType: Defines the type of control. Can be Slider or TextBox.

7.12.10 IntegerUnitRange

The Integer Unit Range Constraint Definition (ConstraintTypeId: IntegerUnitRange) limits an integer value to a specific range, and, optional, a specific interval, and includes unit specification.

IntegerUnitRange has the following additional properties:

MinValue: Minimum value of the range.

MaxValue: Maximum value of the range.

Interval: Interval of the range.

ControlType: Defines the type of control. Can be Slider or TextBox.

UnitId: Unit of the value. These are the units provided by the user. This is converted to ConstraintUnitId before checking against limits and interval. Possible values:

- 1 = nm
- 2 = my
- 4 = mm
- 8 = cm
- 16 = m
- 32 = km
- 64 = decimal inch
- 128 = decimal feet
- 256 = decimal yards
- 512 = miles
- 1024 = sea miles
- 2048 = Decimal Degrees (full circle = 360.0°)
- 4096 = Radians
- 8192 = Grads (full circle = 400 grads)
- 16384 = Bar
- 32768 = Pascal
- 65536 = Pound-force per square inch
- 131072 = Watts
- 262144 = kW
- 524288 = PS

DefaultUnitId: Default unit of the value. One of the UnitId values, specifying the units of the default value, which may be the last used Inventor parameter value, or the value provided as ParameterValue. If ParameterValue is omitted, specify the units of your Inventor model here.

ConstraintUnitId: One of the UnitId values, specifying the units of MaxValue, MinValue, and Interval.

7.12.11 DoubleUnitRange

The Double Unit Range Constraint Definition (ConstraintTypeId: DoubleUnitRange) limits a double value to a specific range, and, optional, a specific interval, and includes unit specification.

DoubleUnitRange has the following additional properties:

MinValue: Minimum value of the range.

MaxValue: Maximum value of the range.

Interval: Interval of the range.

ControlType: Defines the type of control. Can be Slider or TextBox.

UnitId: Unit of the value. These are the units provided by the user. This is converted to ConstraintUnitId before checking against limits and interval. Possible values: see above (IntegerUnitRange)

DefaultUnitId: Default unit of the value. One of the UnitId values, specifying the units of the default value, which may be the last used Inventor parameter value, or the value provided as ParameterValue. If ParameterValue is omitted, specify the units of your Inventor model here.

ConstraintUnitId: One of the UnitId values, specifying the units of MaxValue, MinValue, and Interval.

7.12.12 StringList

The String List Constraint Definition (ConstraintTypeId: StringList) limits a string value to a specific number of string values. This does not allow for any translation. Use the StringDictionary for this.

StringList has the following additional properties:

ListValues: Values to be displayed in the frontend. Example:

```
"ListValues": [
 "Value 1",
 "Value 2"
]
```

PredefinedValueOnly: If True then only the values in ListValues are allowed. If False, then the user can enter any value.

ControlType: Defines the type of control. Can be ComboBox.

7.12.13 StringDictionary

The String Dictionary Constraint Definition (ConstraintTypeId: StringDictionary) limits a string value to a specific number of string values but returns the key to the selected dictionary value as the parameter value. This allows translation.

StringDictionary has the following additional properties:

DictionaryValues: Values to be displayed in the frontend. Example:

```
 "DictionaryValues": {
   "Key1": [
     {
       "Language": "En",
       "Text": "Englisch"
     },
     {
       "Language": "De",
       "Text": "Deutsch"
     },
     {
       "Language": "Fr",
       "Text": "Franz\u00F6sisch"
     }
   ]
 }
```

ControlType: Defines the type of control. Can be ComboBox.

7.13 Settings

In the settings domain, we manage various configuration areas:

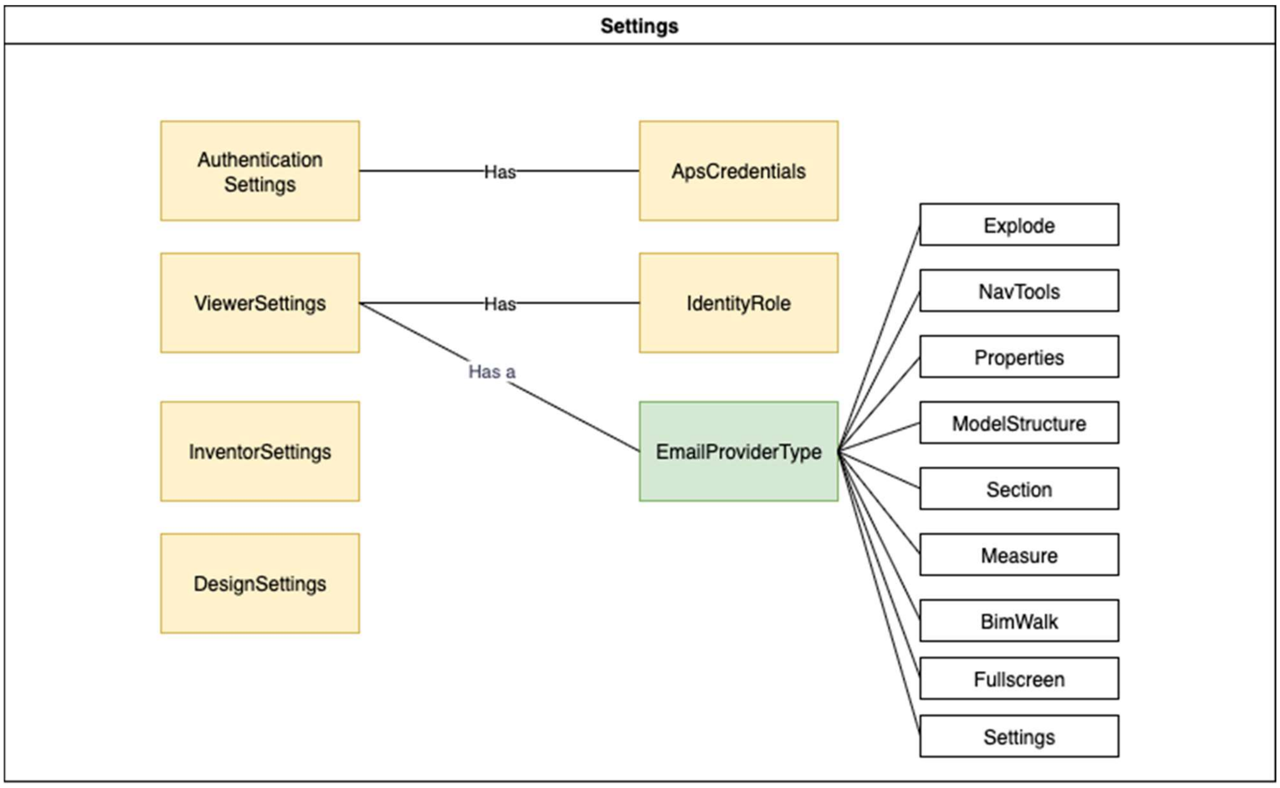
Authentication settings – control user-related preferences, for example, whether to allow only restricted email addresses by maintaining a whitelist.

APS credentials – store access details for Autodesk Platform Services, such as client ID, client secret, and scopes required to generate an OAuth token.

Design settings – define corporate design aspects, including the primary color and other branding elements.

Inventor settings – cover configurations for provisioning an Inventor machine.

Viewer settings – specify which features are available to users within the Autodesk Configurator.



Legend

AuthenticationSettings has one **ApsCredentials**.

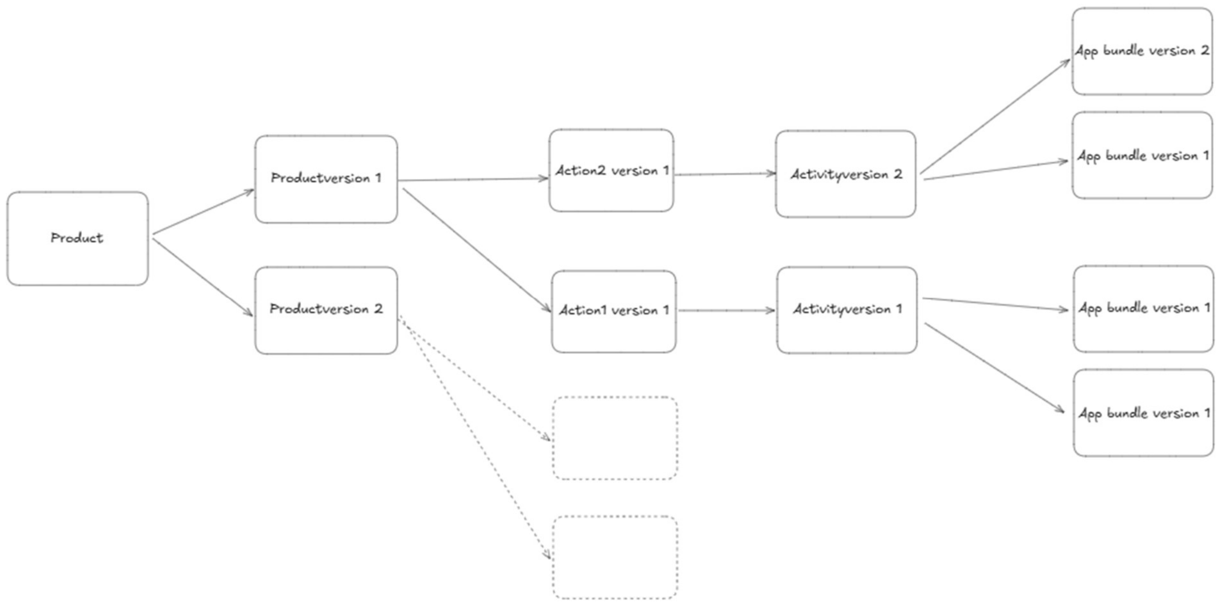
ViewerSettings has one **Identity Role**.

EmailProviderType is an enumeration.

7.14 Versioning introduction

From the beginning it was defined that each product configuration created should be recreatable. This way it is also not needed to save any data because it could be recreated at any time.

The diagram below shows the dependency between the objects.



7.14.1 Product Version

When creating a new product, the first version is automatically created and CAD-Data uploaded to it. From this point on naming, descriptions and translations can be changed on the product version, but the CAD-Data cannot be updated. A new version needs to be created. Only then it’s possible to add new CAD-Data. This way we can guarantee that for this version we exactly now what CAD-Data belongs to it with what parameters.

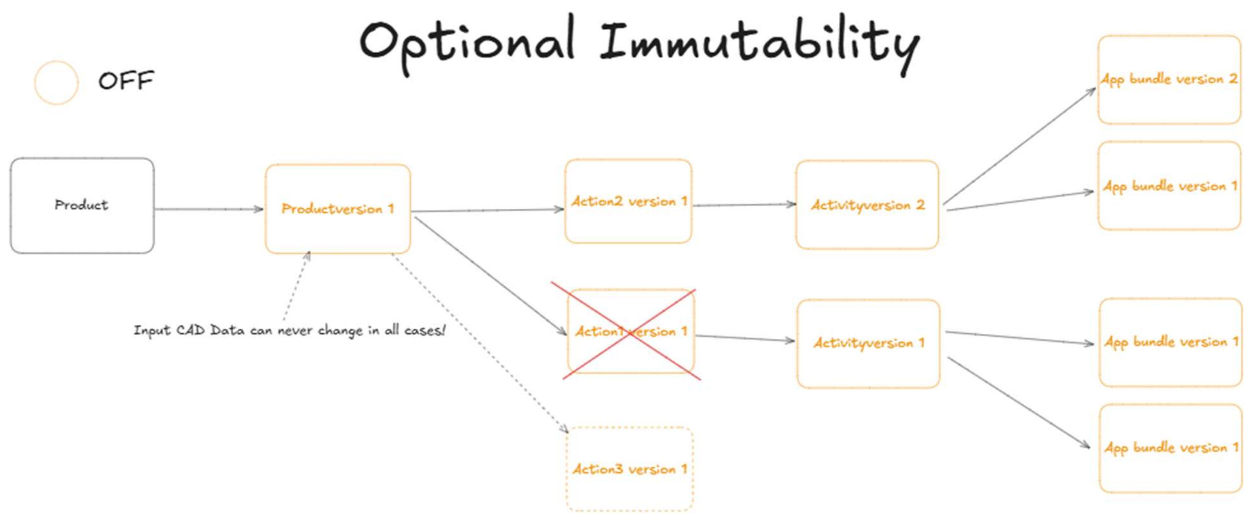
Product versions can also have 1 of 4 states:

- Active
- Inactive
- Development
- Internal

Only one can be active any time. This is enforced automatically. Active is what is shown to the user. Other roles allow to see inactive, development and internal. An admin for example has a dropdown on the configurator that allows to select a development version that should not yet be shown to the customer.

This also allows to set different tabs visibility, default views and other things to each version.

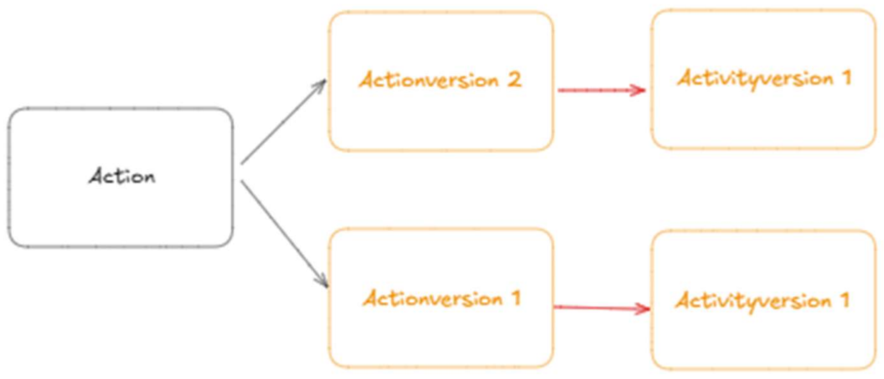
Currently it is possible to change the assigned actions and with that breaking the exact recreation. This allows for some flexibility, by trading in the exact recreation. In the future it is planned to implement something called Immutability / Optional Immutability. This would allow to lock a product version so actions can no longer be added, removed or changed in any way. Currently this is not implemented. The current implementation is what we named optional immutability, but it is not possible to switch to the other.



The diagram shows the dependency from the product down to the app bundle. This is how it is currently implemented. It is possible to switch, add, remove an action at any time.

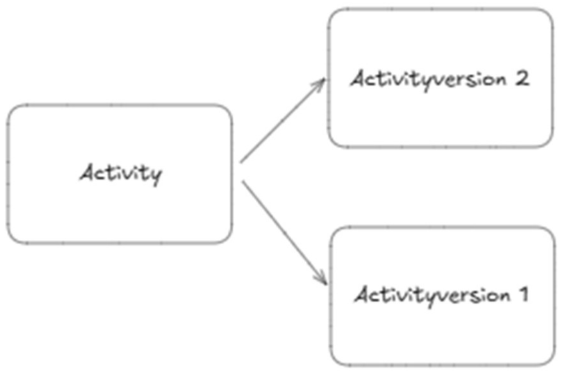
7.14.2 Action version

Actions are versioned as everything else. After creating an action version, it is no longer possible to edit it. A new version needs to be created. This makes sure that after a version is created it will always reference the exact selected activity.



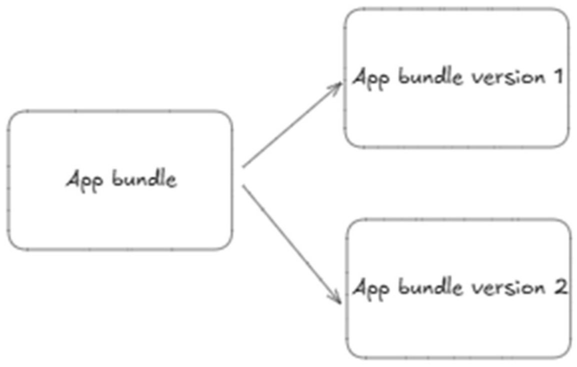
7.14.3 Activity Version

Activities reference 1 or more app bundles. To make sure they are versionable it is needed to version app bundles and activities. If a certain activity version is used by an action, it is always possible to tell what exact version of an app bundle / app bundles it uses.

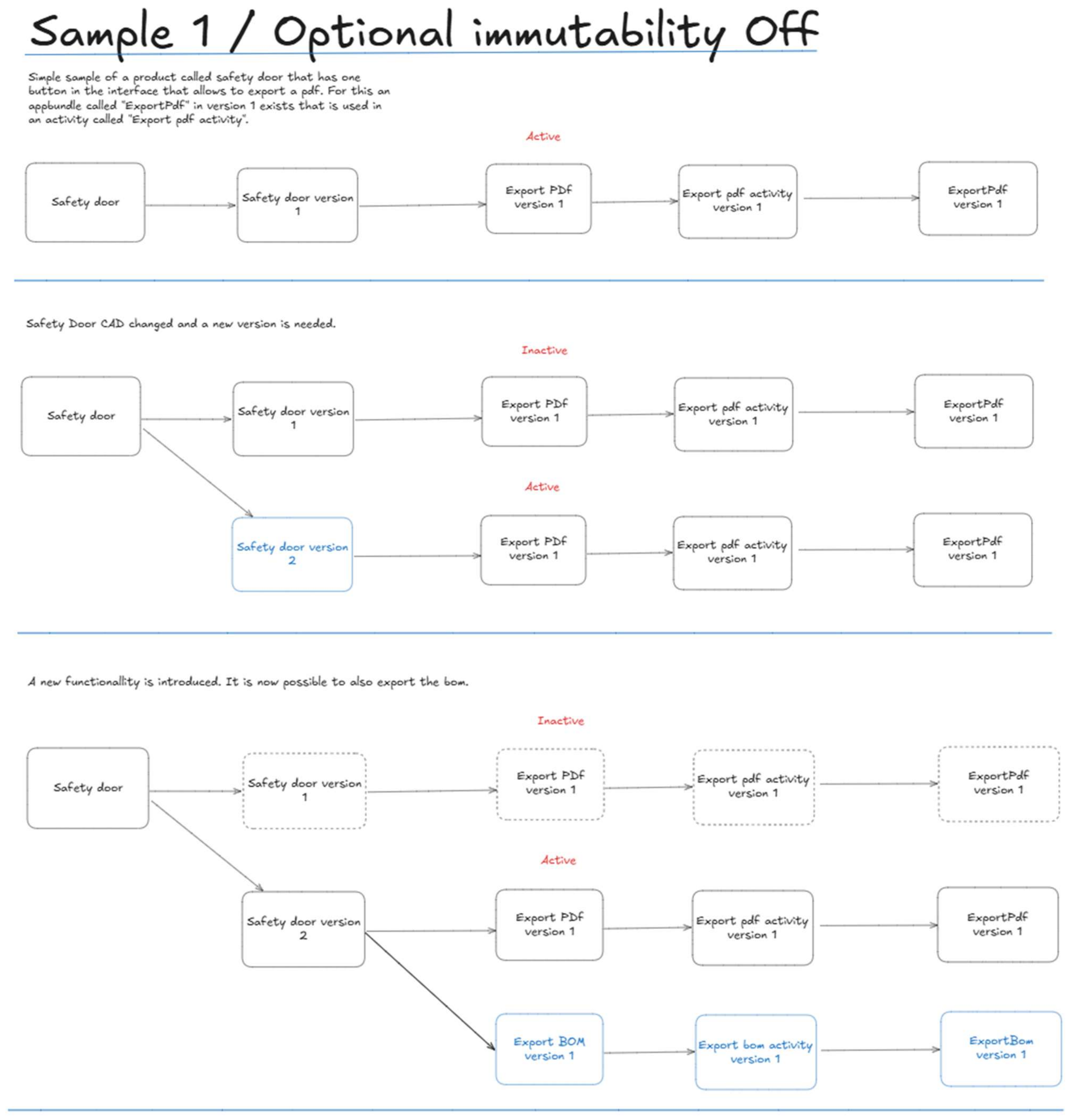


7.14.4 App bundle Version

If an app bundle changed a new version needs to be uploaded. It is by design not possible to change a once uploaded app bundle. This is a constraint by Autodesk. We enforce this by making an app bundle not editable.



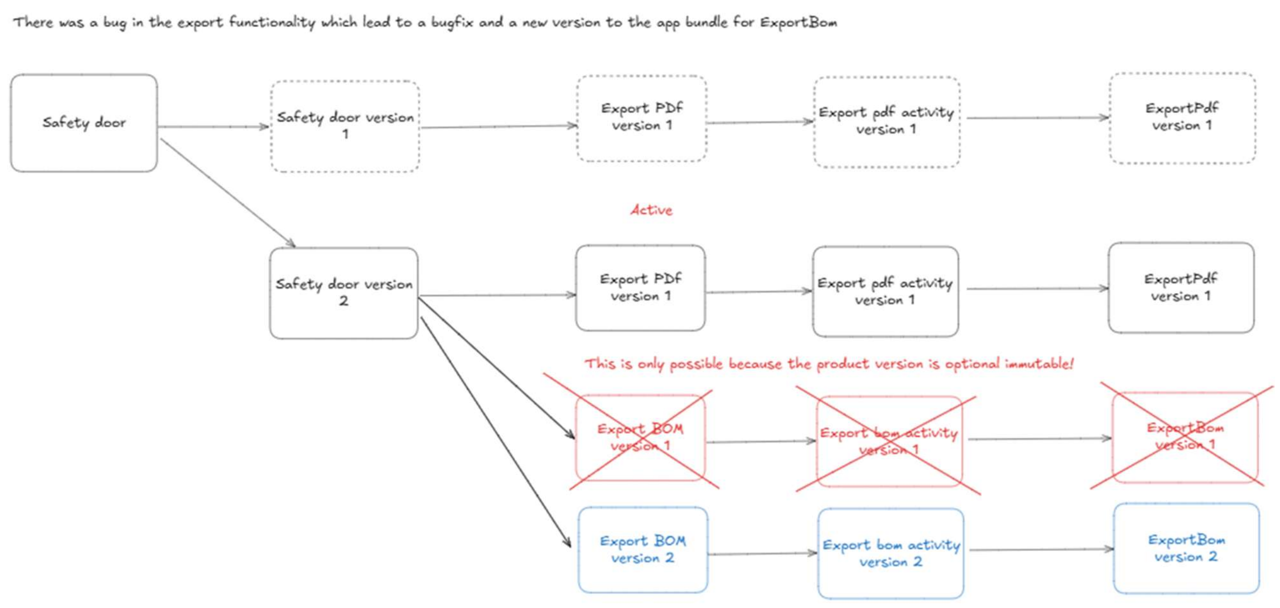
7.14.5 Versioning samples

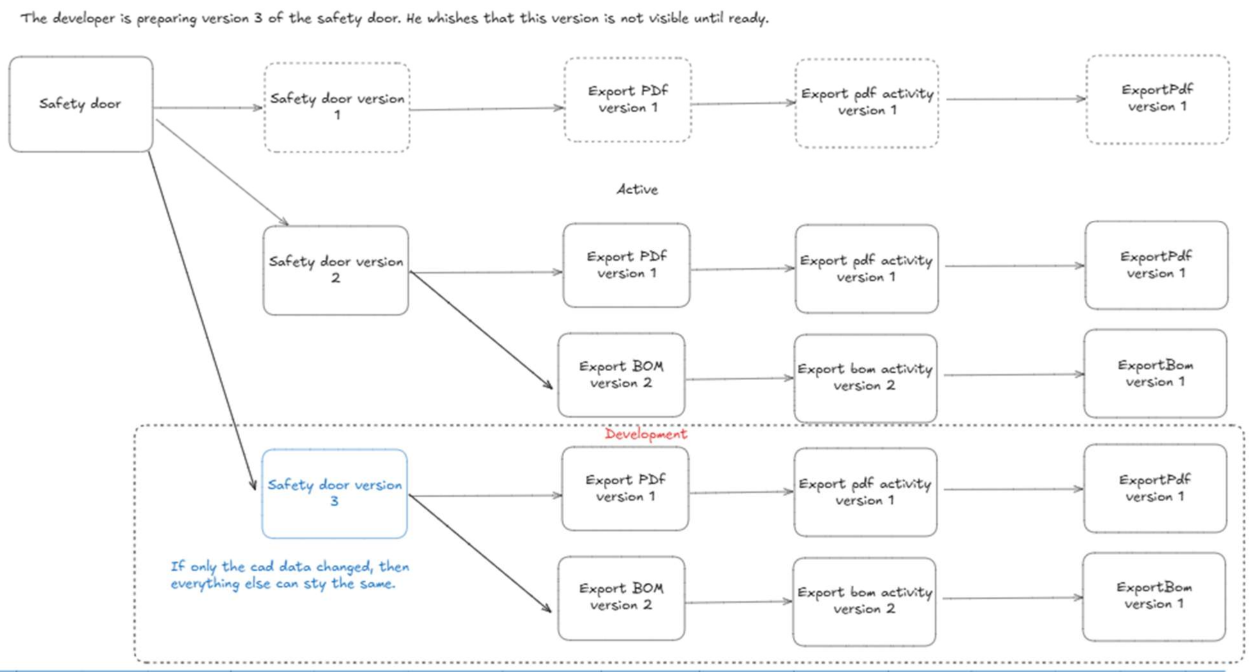

Sample 1 / Optional immutability Off

Simple sample of a product called safety door that has one button in the interface that allows to export a pdf. For this an appbundle called "ExportPdf" in version 1 exists that is used in an activity called "Export pdf activity".

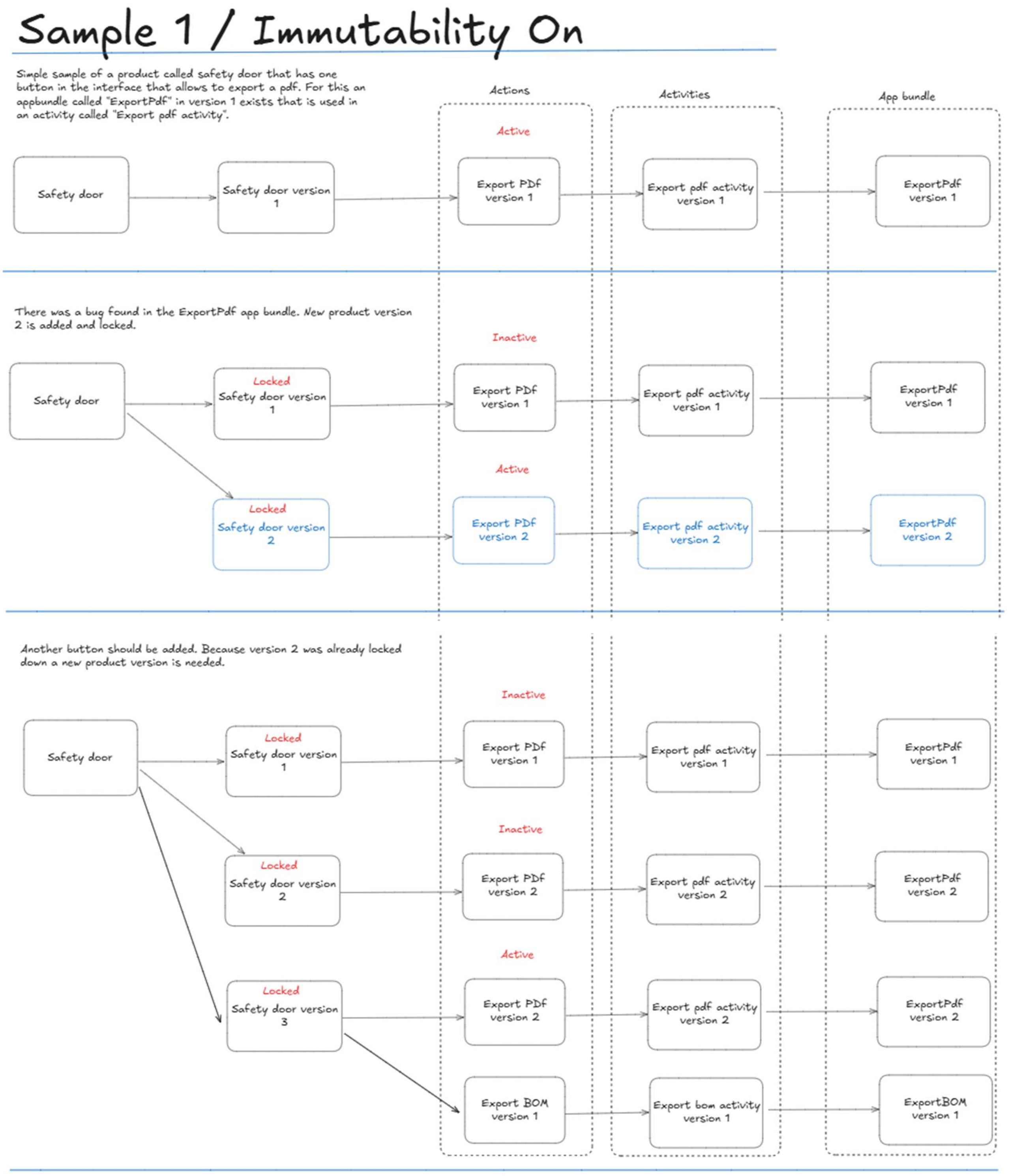
Safety Door CAD changed and a new version is needed.

A new functionallity is introduced. It is now possible to also export the bom.


There was a bug in the export functionality which lead to a bugfix and a new version to the app bundle for ExportBom
Safety door
Safety door version 1
Export PDF version 1
Export pdf activity version 1
ExportPdf version 1
Active
Safety door version 2
Export PDF version 1
Export pdf activity version 1
ExportPdf version 1
This is only possible because the product version is optional immutable!
Export BOM version 1
Export bom activity version 1
ExportBom version 1
Export BOM version 2
Export bom activity version 2
ExportBom version 2


The developer is preparing version 3 of the safety door. He whishes that this version is not visible until ready.
Safety door
Safety door version 1
Export PDF version 1
Export pdf activity version 1
ExportPdf version 1
Active
Safety door version 2
Export PDF version 1
Export pdf activity version 1
ExportPdf version 1
Export BOM version 2
Export bom activity version 2
ExportBom version 1
Development
Safety door version 3
Export PDF version 1
Export pdf activity version 1
ExportPdf version 1
If only the cad data changed, then everything else can sty the same.
Export BOM version 2
Export bom activity version 2
ExportBom version 1

The next sample shows how immutability would work. This was not implemented and is an extended goal. (Issue 174)


Sample 1 / Immutability On
Simple sample of a product called safety door that has one button in the interface that allows to export a pdf. For this an appbundle called "ExportPdf" in version 1 exists that is used in an activity called "Export pdf activity".
Actions
Activities
App bundle
Active
Safety door
Safety door version 1
Export PDF version 1
Export pdf activity version 1
ExportPdf version 1
There was a bug found in the ExportPdf app bundle. New product version 2 is added and locked.
Inactive
Locked
Safety door version 2
Export PDF version 2
Export pdf activity version 2
ExportPdf version 2
Another button should be added. Because version 2 was already locked down a new product version is needed.
Safety door version 3
Export BOM version 1
Export bom activity version 1
ExportBOM version 1

8 System Architecture & Design

8.1 Architecture

The architecture of our web application is built around a few key components: the UI, the API, the database, the Aspire Dashboard for telemetry, various Autodesk services, and a mail server.

The UI serves as the main entry point for users. It communicates exclusively with the API over HTTP. For real-time updates, the backend also pushes notifications to the UI using WebSockets. This channel is one-way: only the backend can send messages to the UI, ensuring a clear separation of responsibilities. Authentication between the UI and the API is handled using cookie-based authentication.

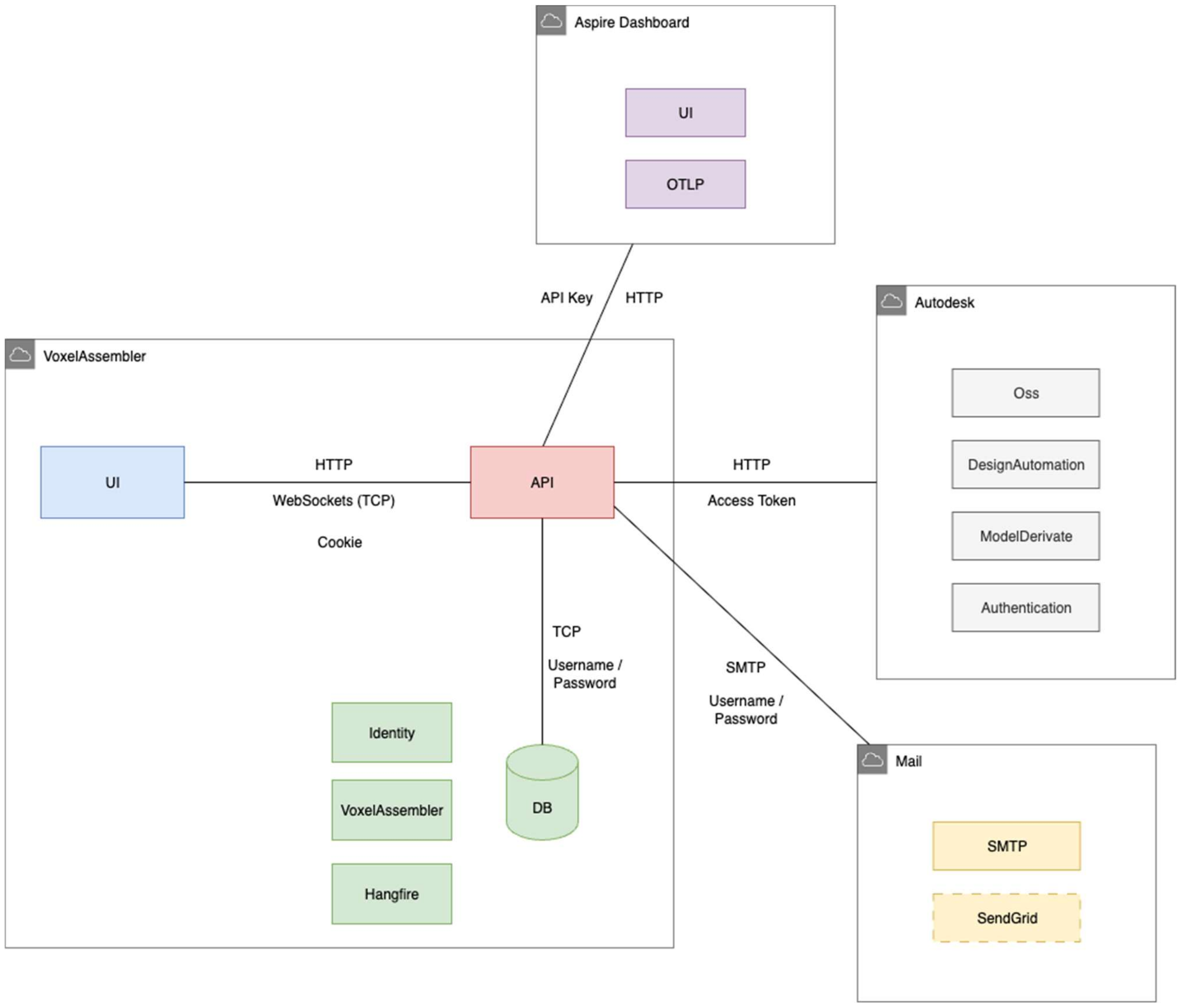
At the core lies the API, which connects all parts of the system. It talks to the database, which is structured into three schemas: the Identity schema powered by the .NET Identity Framework for authentication and authorization, the VoxelAssembler schema, which holds the main business data and domain logic and the Hangfire schema, automatically managed by Hangfire for background processing and asynchronous tasks. Database access requires username and password credentials for PostgreSQL.

The API also integrates with external systems. For example, it uses an SMTP mail server to send transactional emails such as password resets and registration confirmations. In the future, we plan to extend this setup with additional providers like SendGrid to increase flexibility and reliability.

For observability, the backend is instrumented with OpenTelemetry. All telemetry data (traces, logs, and metrics) are exported via gRPC to the Aspire OTel Collector, which powers the Aspire Dashboard. Access to Aspire is secured via an API key, ensuring only authorized components can send telemetry data. This gives us a simple but effective way to monitor the health and performance of the entire system.

Finally, the application connects to several Autodesk services. These include OSS, Automation (formerly Design Automation), Model Derivative, Viewer SDK and Authentication, which together provide the foundation for our product configurator and enable seamless integration with Autodesk's ecosystem. All Autodesk service calls are authenticated via OAuth access tokens, ensuring secure interaction with Autodesk's APIs.

In summary, the UI provides the user-facing entry point, the API orchestrates both internal and external interactions, the database manages core application data and background jobs, telemetry flows into Aspire for monitoring, and external integrations with mail servers and Autodesk services round out the system's capabilities.


Aspire Dashboard
UI
OTLP
API Key
HTTP
Autodesk
Oss
DesignAutomation
ModelDerivate
Authentication
VoxelAssembler
UI
HTTP
WebSockets (TCP)
Cookie
API
HTTP
Access Token
TCP
Username / Password
SMTP
Username / Password
Identity
VoxelAssembler
Hangfire
DB
Mail
SMTP
SendGrid

8.1.1 Authentication

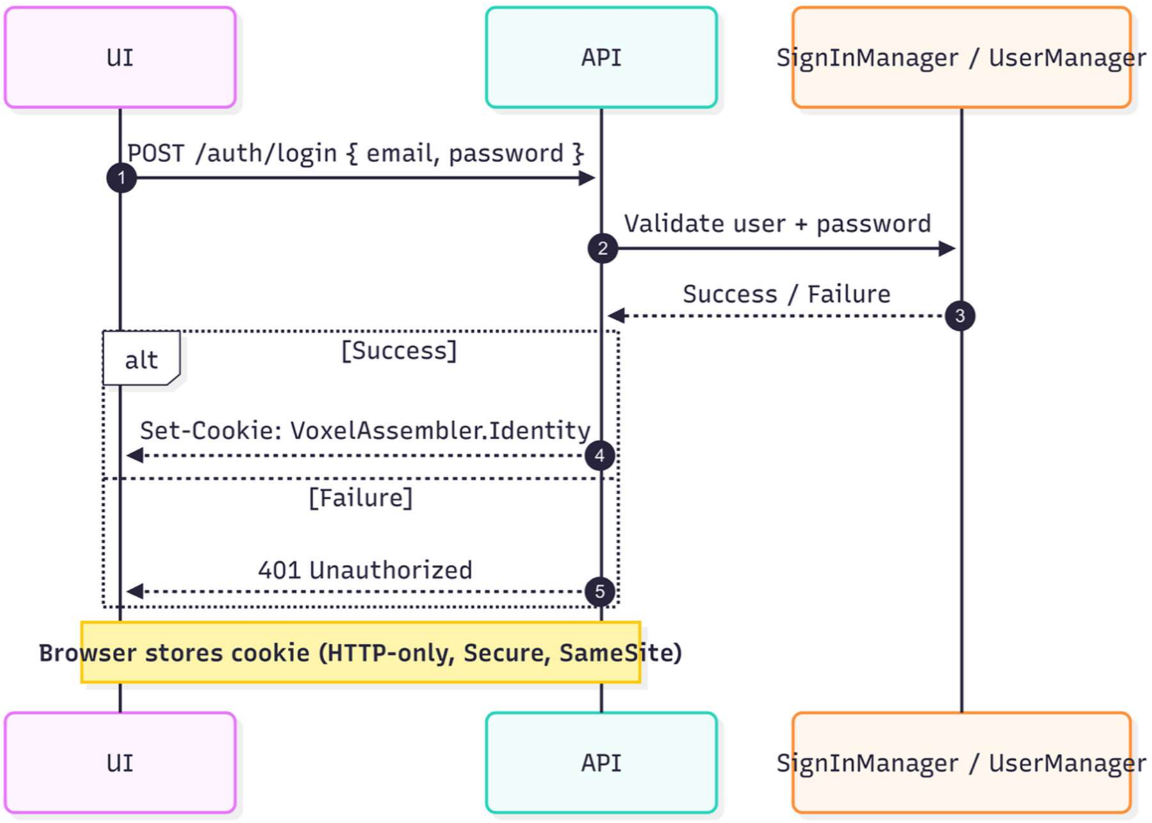
We use cookie-based authentication for our web application. The decision to use cookie authentication is documented in the attached ADR (**02_Additional_Product_Documentation\05_Architectural_Decision_Records\001-authentication-method-cookies**).

Our implementation relies on .NET Identity. The API is responsible for issuing and encrypting the authentication cookie, and for setting it in the Set-Cookie response header.

On subsequent requests, the client automatically sends the cookie back to the server. For requests targeting protected endpoints, the cookie is decrypted using the Data Protection keys. The system verifies whether the user is authenticated and then checks the user's roles to ensure they are authorized to access the endpoint, meaning they have the required application permissions.

The authentication cookie is configured with the following security settings:

- HttpOnly
- Secure
- SameSite

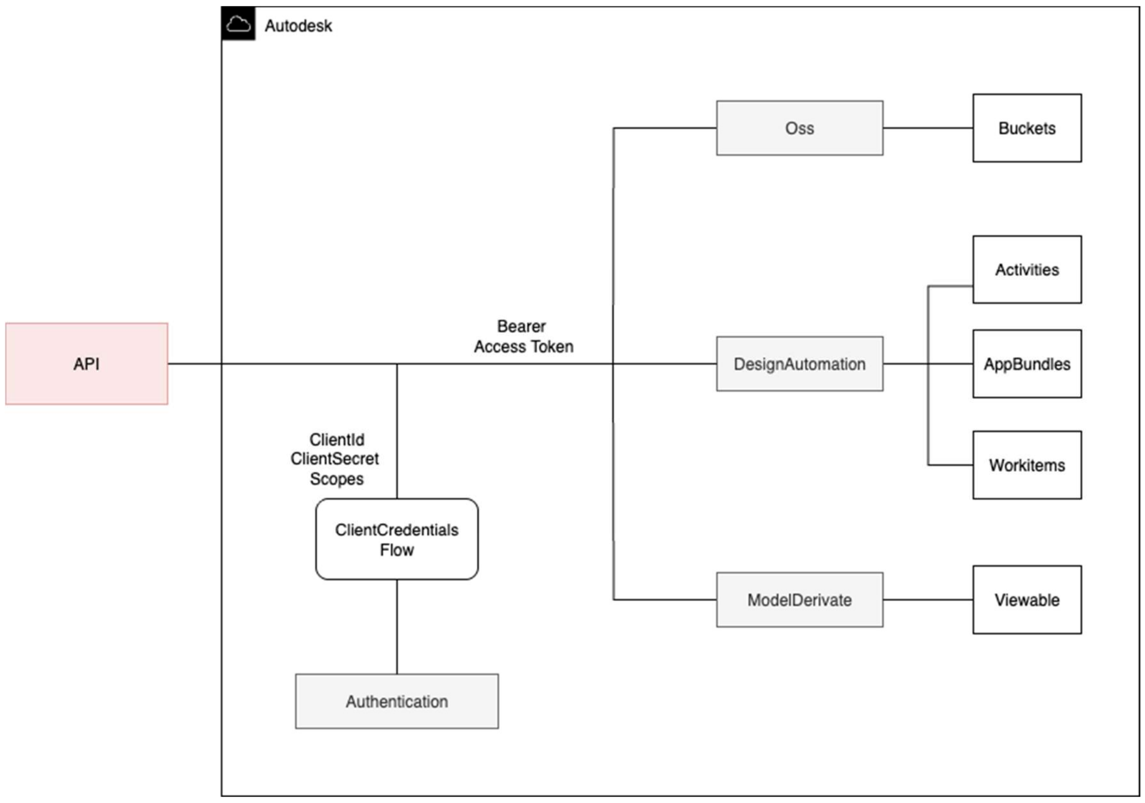

UI
API
SignInManager / UserManager
POST /auth/login { email, password }
1
Validate user + password
2
Success / Failure
3
alt
[Success]
Set-Cookie: VoxelAssembler.Identity
4
[Failure]
401 Unauthorized
5
Browser stores cookie (HTTP-only, Secure, SameSite)
UI
API
SignInManager / UserManager

8.1.2 Autodesk Services (Autodesk Platform Services)

Our application integrates with several Autodesk services through the official SDKs that Autodesk provides as NuGet packages. These SDKs allow us to communicate with Autodesk APIs in a straightforward and consistent way.

For authentication, we use the OAuth 2.0 client credentials flow. Autodesk offers a dedicated Authentication SDK, which we use to obtain an access token. This token is stored in memory and checked before every request to ensure it is still valid. To avoid issues with tokens expiring mid-operation, we apply a safety margin of 60 seconds, meaning the token is refreshed one minute before its official expiration time.

With a valid access token, our application interacts with three main Autodesk services: OSS, Design Automation, and Model Derivative. The OSS service essentially acts as an Autodesk-provided wrapper around AWS S3 buckets, giving us a managed way to store and retrieve design files. Design Automation allows us to run custom processes and scripts on design files directly in the cloud, which enables automated transformations and workflows. Finally, the Model Derivative service makes it possible to convert design files into different formats and extract metadata, which we use for visualization and structured data access.



8.1.2.1 Authentication API

Most Autodesk API’s need authentication which is one of the building blocks that is used by VoxelAssembler.

8.1.2.2 Automation API (formerly Design Automation)

The Automation API allows to run Inventor in the Cloud. By running a WorkItem resources in AWS are used to provide a CAD in the Cloud and run functionality that is normally only available locally by installing Autodesk Inventor.
By using the Automation API no Installation and no License is needed. Only the used processing time needs to be paid.

8.1.2.3 Data Management API

The data management API is used for hosting data. It allows to upload CAD-Data and produced results directly on the cloud. Data is free and not limited.

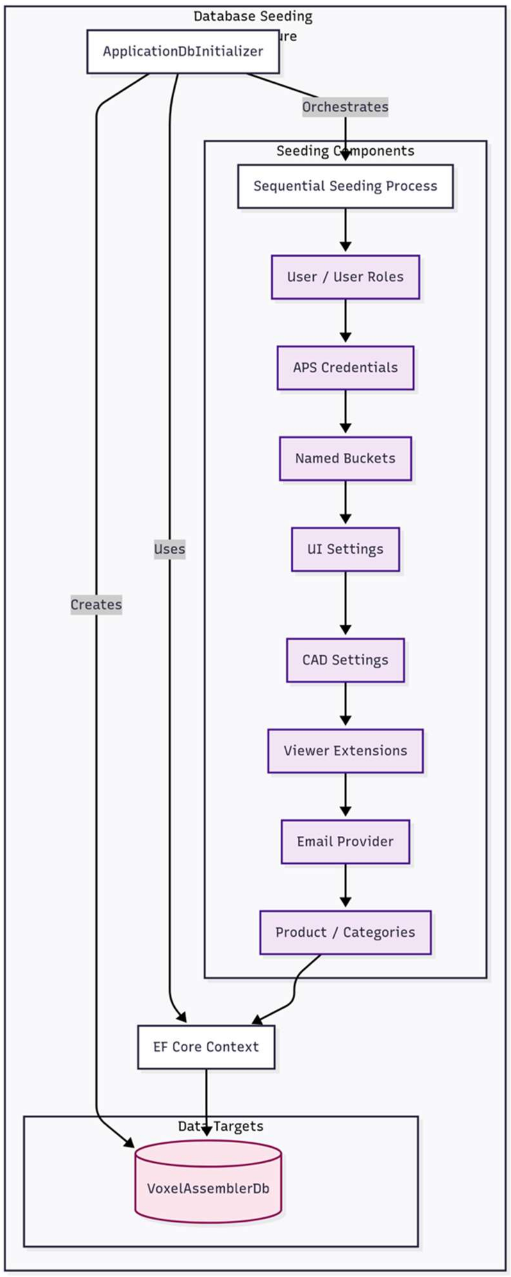
8.1.2.4 Model Derivative API

The model derivative API is used for translation. In the case of VoxelAssembler it is currently only used for creating Viewables which is what is needed to display something in the Viewer.

8.1.2.5 Viewer SDK

The viewer SDK is what is used to display Viewables to the user. It is based on ThreeJS but provides lots of functionality like all the extensions (Measure, Explode, Navigation etc.)

8.1.3 Db Initializer



The Db Initializer is a console application that we use extensively in local development, but it also plays a key role when running in the cloud or in production. Its job is to bring the database into an initial state: creating schemas, seeding essential data, and setting up sensible defaults so that users can start working with the system right away instead of having to configure everything manually. During initialization, the database is not only created but also populated with core application data. This includes user roles and default accounts for each role, so that you can log in immediately with testable credentials (passwords must of course be changed after first login). Authentication settings are also seeded, ranging from Autodesk credentials like Client ID, Client Secret, and scopes, to general application settings such as whitelisted domains.
The initializer goes further by preparing the integration with Autodesk: it can create initial buckets for storing artifacts, configure Inventor settings like polling intervals or whether an instance should be kept alive to avoid cold starts, and set up default values for the Autodesk viewer and configurator.
On the application side, it also takes care of UI defaults such as company name, logo, and color scheme, as well as email provider settings by seeding SMTP credentials. For local testing purposes, the initializer can even create example products and categories.
Almost all this seeded data can later be customized through the UI, and adding new entities to the seeding process is straightforward. The initializer itself is instrumented with OpenTelemetry and exports its telemetry to Aspire, making the entire process transparent and easy to observe.

8.1.4 DesignAutomation

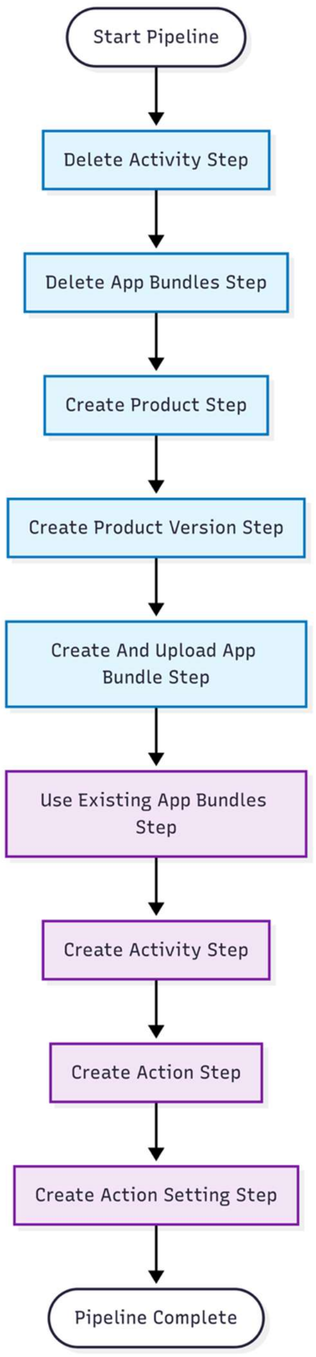
Design Automation is a console application that, like a database initializer, sets up an initial configuration so that users can start with a default setup right away—without having to configure everything manually in the UI first.

The core of the project consists of individual steps, each representing a specific action (e.g. *Create Product*, *Create Activity*, etc.). These steps are combined into a pipeline. We have multiple pipelines, each composed of different steps. The most used one is the FullConfigurePipeline, which runs through all steps. This pipeline has often been used for testing purposes.

Pipelines and steps are fully extensible and can be expanded as needed.

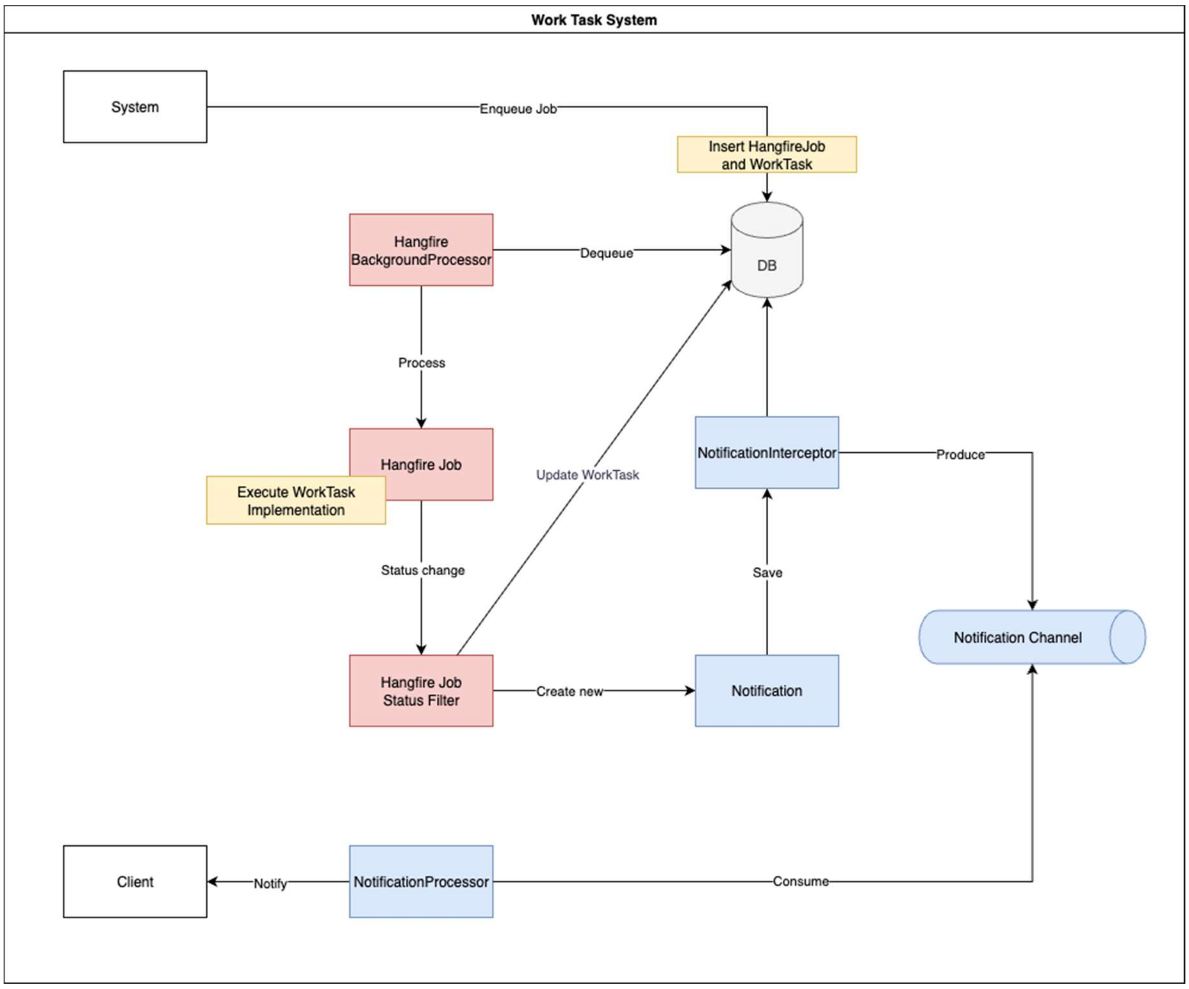
The process does not only write objects into our database but also performs initializations directly in Autodesk, such as creating activities or app bundles. Because of this, it is crucial that the database is already initialized before the Design Automation project is started.

The choice of which pipeline to execute is determined via command-line arguments.

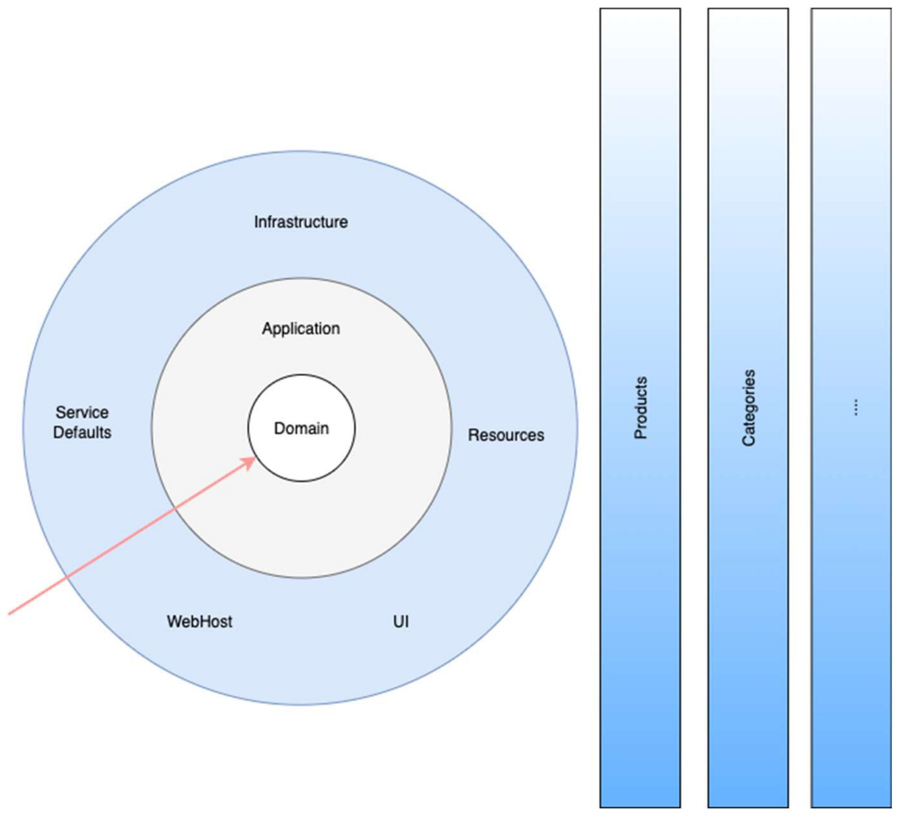


8.1.5 Work Tasks System

Our WorkTask system is responsible for processing asynchronous, long-running background jobs and notifying the client about their status. It is designed to be flexible and extensible so that virtually any kind of long-running job can be implemented with it. To add a new WorkTask, you only need to implement the IWorkTask interface and the WorkTask system takes care of the rest. In practice, we primarily use it for jobs related to our product versions and product configurations, which is why a WorkTask also contains specific properties in this context. Each WorkTask triggers a Hangfire job and stores its corresponding Job ID. The WorkTask and the Hangfire job are created within a single transaction to ensure consistency and avoid mismatches. A Hangfire Job Status Filter ensures that whenever the job status changes, the associated work task is also updated so that both remain in sync. For certain Hangfire statuses, such as Failed, InProgress, or Succeeded, we additionally write a notification into the database. An Entity Framework interceptor then pushes this notification into a Channel (from the System.Threading.Channels library).
Finally, a Notification Processor consumes messages from the channel and sends them via SignalR to the client of the user who originally triggered the work task.



8.2 Solution Design



The solution is built on a modern architectural foundation that combines Vertical Slice Architecture with the principles of Clean Architecture. This approach creates a highly organized and maintainable codebase by emphasizing a clear separation of concerns and a logical flow of dependencies.

The design organizes code by feature, or "vertical slice," rather than by technical layers (e.g., Controllers, Services, Data). Each slice represents a distinct business capability (e.g., Product, Category, ProductVersion) and contains all the necessary components to implement that feature from the API endpoint down to the database. The solution is divided into distinct projects, each with a clear responsibility. Key Benefits of this Design are:

High Cohesion: All the code for a single feature is in one place, making it easier to understand, develop, and maintain.

Low Coupling: Slices are isolated from one another. A change to the "Product" feature is unlikely to impact the "Category" feature.

Enhanced Maintainability: Adding or changing a feature involves working within a single slice, minimizing the risk of introducing bugs elsewhere in the system.

Scalability: The clear separation of concerns and low coupling make it easier to scale development teams and the application itself.

8.2.1 VoxelAssembler.Domain

This is the core of the application. It contains the business entities, value objects, and domain logic. It has no dependencies on any other layer, ensuring the business rules are independent of technical implementation details. This is also checked by an architecture Test.

8.2.2 VoxelAssembler.Application

This project contains the application logic and orchestrates the domain layer to perform tasks. It's where the vertical slices are most evident. Each feature folder (e.g., /Product, /Category) includes:

Contracts: Interfaces for services (IProductService).
Requests/Responses: Models for interacting with the API (CreateProductRequest, ProductResponse).
Services: The implementation of the use cases, containing the main logic for the feature.

8.2.3 VoxelAssembler.Infrastructure

This project handles external concerns. It contains implementations for interfaces defined in the Application layer, such as database access (using Entity Framework Core), file storage, and other external services.

8.2.4 VoxelAssembler.WebHost

This is the entry point and presentation layer. It contains the ASP.NET Core API controllers. The controllers are kept thin, with their primary role being to receive HTTP requests, call the appropriate service in the Application layer, and return an HTTP response. We want to keep this layer free from business logic to ensure a clean separation of concerns and maintainability.
Because no business logic is tied to this layer, the presentation technology can easily be replaced or extended. For example, if we decide to expose the system through GraphQL, gRPC, or another protocol in addition to REST, we can add or replace the WebHost implementation without affecting the core application logic.

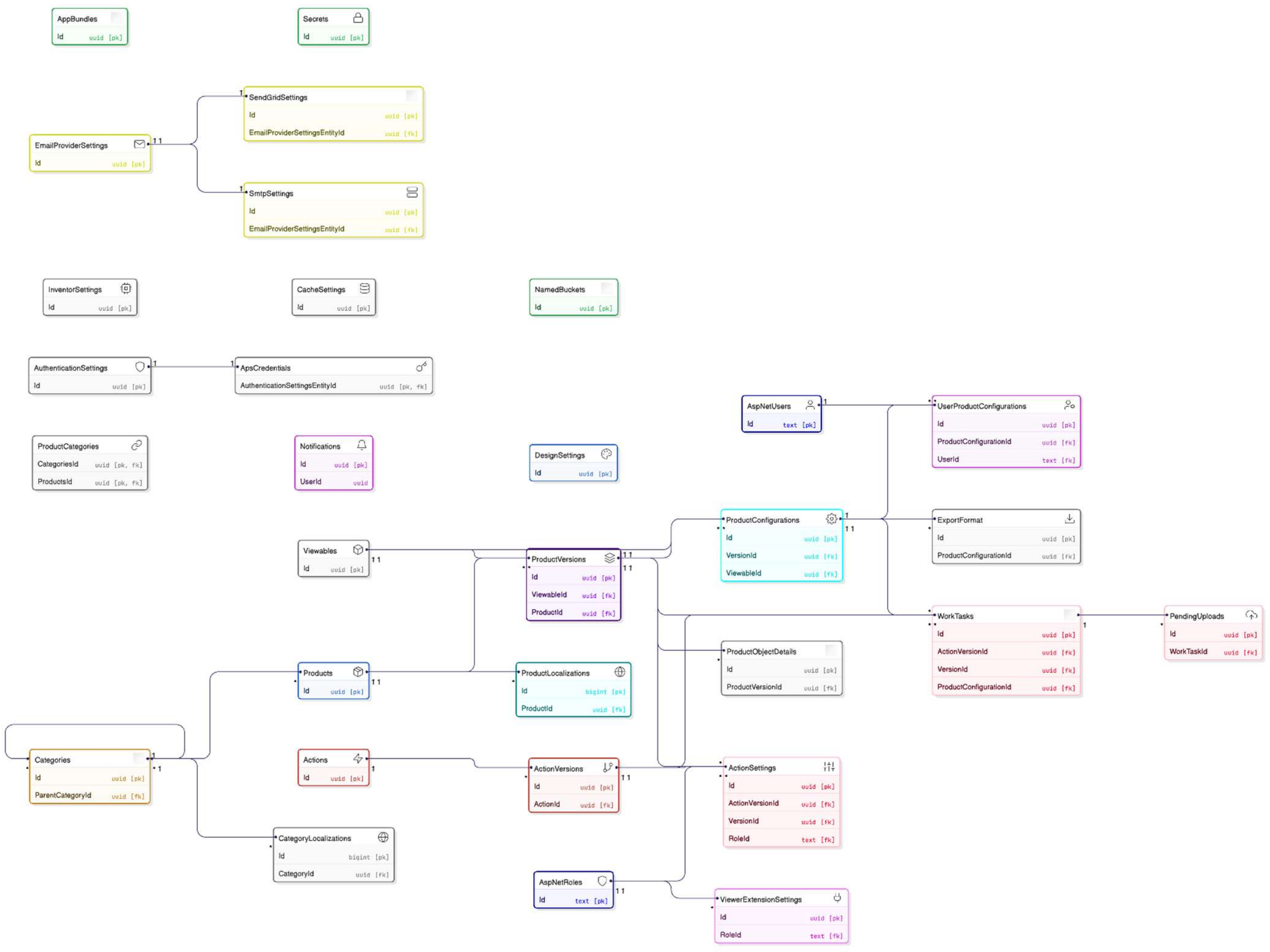
8.2.5 VoxelAssembler.UI

This project is the frontend React & TypeScript app. It contains all the dependencies and libraries that were used in the frontend, such as TailwindCSS for styling, Zustand for state management, i18n for localization, TanstackQuery for data fetching and caching and more.
To ensure a consistent and type-safe integration with the backend, a TypeScript API client is automatically generated into the UI project whenever the WebHost is built. This is done using a NSwag configuration that runs on every WebHost build. By doing so, mismatches between the API

and the frontend are avoided, duplication of models and types is prevented, and API calls are simplified and accelerated.
The project aligns with the Application layer's vertical slice architecture. Every module/feature contains all elements that are relevant to that feature, such as pages, components, types and hooks. This increases the application's maintainability and scalability, making it easier to extend the app in the future.

8.3 Database Design

In this diagram, we have only included the tables with their PK and FK relationships. The complete ER diagram can be found in the attachments under **"\02_Additional_Product_Documentation\02_ER_Diagram"**. Furthermore, we have only included the essential tables from the Identity schema and have omitted the Hangfire schema entirely.



8.4 Design Principles

The VoxelAssembler application follows several key design principles to ensure maintainability, scalability, and code quality:

- SOLID Principles
- Clean Architecture
- Domain-Driven Design (DDD)
- Vertical Slice Architecture

8.4.1 Additional Principles

- KISS (Keep It Simple, Stupid): Solutions favor simplicity over complexity. Complex problems are broken down into manageable, understandable components. For example, we do not use any repositories because complexity does not justify the complexity.
- Immutability: Preference for immutable objects and value types where possible to reduce side effects and improve predictability.
- Fail-Fast: Input validation and error handling using the Result pattern (ErrorOr) to catch issues early and provide meaningful feedback.
- Testability: Architecture designed to support comprehensive testing through dependency injection and clear separation of concerns.

These principles work together to create a maintainable, extensible system that can evolve with changing business requirements while maintaining code quality and developer productivity.

8.5 Technologies & Frameworks

Our application is a modern web-based system built using a clean and modular architecture. It follows the principles of Domain-Driven Design and separates concerns across multiple layers. Below is a breakdown of the technologies and tools used in each area of the system.

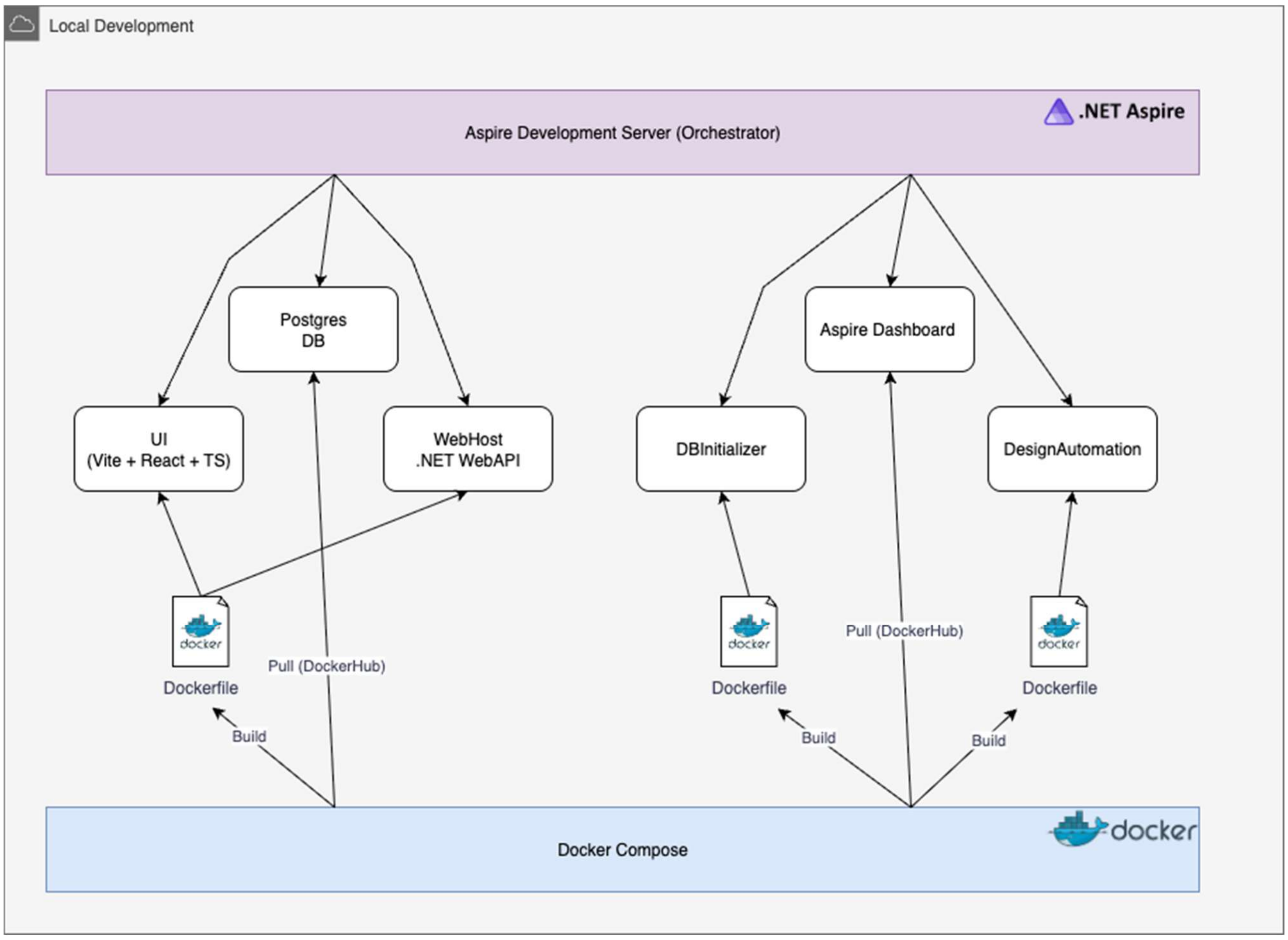
Scope	Technology / Framework	Description / Reason
User Interface	React, Typescript, Vite	It provides a fast and interactive SPA. This app was a great chance to improve and deepen our knowledge and skills in React & TypeScript in a real world scenario.
CSS Framework	TailwindCSS	Tailwind allows us to write UI components faster, since we don’t have to style anything using CSS.
API Layer	.NET Core, REST, NSwag	All team members are very familiar with .NET and REST. C# is enough powerful to cover all our needs. NSwag is used to generate the API client for the frontend so that we can save time and don’t have to develop any API calls or models in the frontend ourselves.
Persistence	EF Core, PostgreSQL	We use EF Core as our ORM for data access. PostgreSQL (referred to as VoxelAssemblerDb) is used as the main database. A DB initializer sets up the database schema and seeds data.
Authentication & Authorization	Identity	We use ASP.NET Core Identity for managing authentication and authorization. Cookie based authentication is implemented to manage sessions.
Localization (backend)	.resx files	.resx resource files are a common and easy way to solve localization in .NET. It allows us to manage and deliver localized strings based on the user’s culture. This enables easy support for multiple languages on server-side responses and error messages.

Localization (frontend)	i18n	The i18n package is one of the most widely used, if not the most widely used package for localization in the React world. Using i18n keys allows us to adapt the UI texts to the user's preferred language in real time, enhancing accessibility and user experience.
Containerization	Docker	Used for containerizing our app and making deployment possible.
Asynchronous Tasks	Hangfire	Used for long running backend tasks. Mainly for creating configurations on aps.
Testing (backend)	xUnit	Using xUnit for writing Unit, EtoE and Integration tests.
Testing (frontend)	Vitest, MSW	Using MSW allows us to setup a fake API without having to start the whole setup and run the actual server. Using Vitest in combination gives us a very easy solution to test the frontend and how it behaves for certain API responses.
Observability	OpenTelemetry	Provides distributed tracing, logging, and metrics collection across our application. Exports telemetry data via gRPC to the Aspire Dashboard for monitoring API performance, background jobs, and external service integrations like Autodesk services.
Code Quality	StyleCop.Analyzers, EsLint, Prettier	These tools allow us in both, the frontend and the backend, to keep our codes consistent over the whole app. We enabled auto format on save, that the source codes look the same for everyone. Analyzers throw compiler errors when the code does not comply with the set rules.

8.6 Building Block View

8.6.1 Development

For our local development setup, we decided early on to create an Aspire project to make our development process as efficient as possible. The goal was to have all resources of our application orchestrated through Aspire. For example, we also integrated our UI into .NET Aspire. The PostgreSQL database is provisioned through Aspire as well, along with the API, the database initializer, and the design automation project.
Aspire provides very simple provisioning of resources and handles service discovery within the local network. In addition, the dashboard is very useful for quickly analysing errors or restarting services, since all resources can be controlled from there and all telemetry data is exported to it.
Besides the Aspire setup, we also created a Docker setup. This allows us to start the entire application via a Docker Compose file. Our own images are built and run as containers, while for PostgreSQL and Aspire we pull the images directly from Docker Hub.



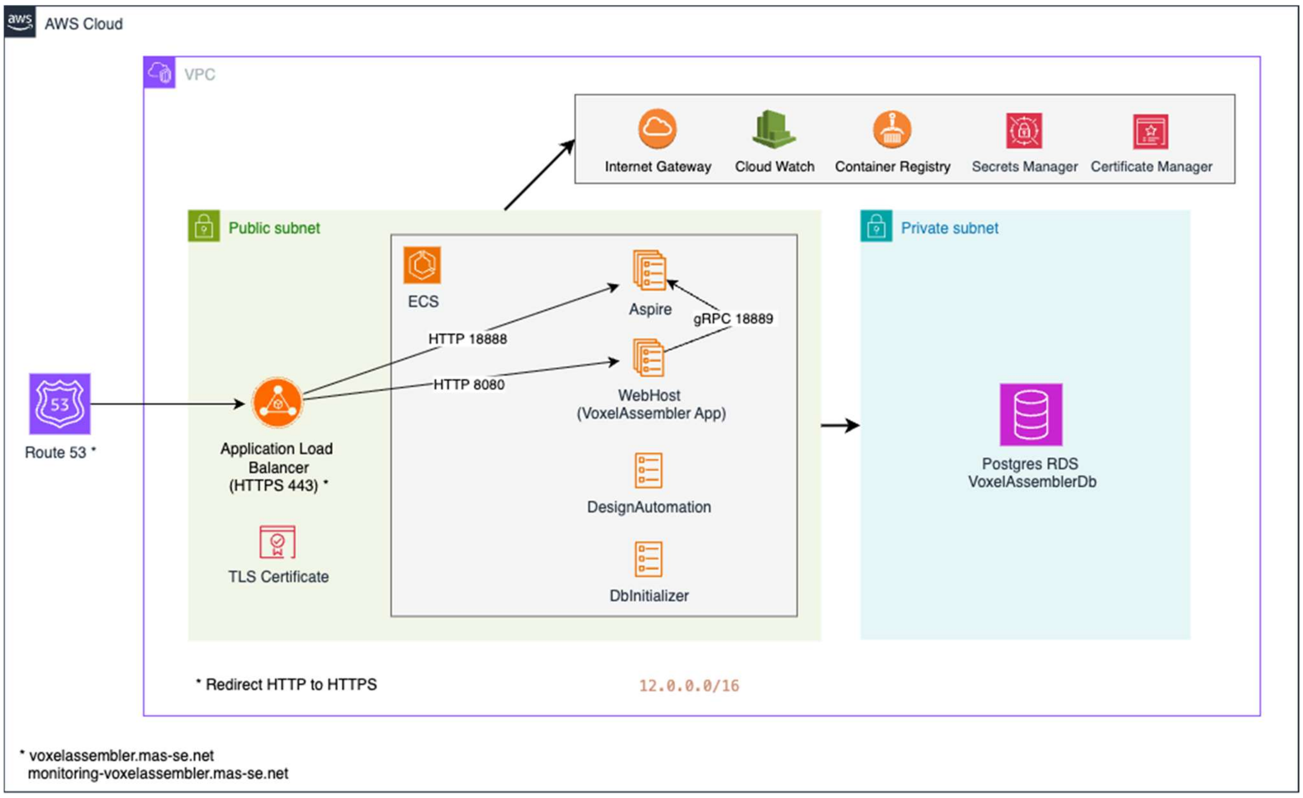
8.6.2 Cloud Deployment

Our application is deployed in AWS inside a dedicated VPC that is split into public and private subnets. The public subnet is connected to the internet through an Internet Gateway, while the private subnet is reserved for internal resources. Within the private subnet, we run a PostgreSQL RDS database. For security reasons, the database has its own security group and no public IP address, which means it cannot be reached from outside the VPC.

In the public subnet, we operate our container workloads using Amazon ECS. The ECS cluster hosts two main services: Aspire and the WebHost, which runs our VoxelAssembler application. In addition to these services, there are also tasks such as DbInitializer and DesignAutomation. These are not long-running services but rather one-time jobs that can be triggered when needed, for example during initialization. Container images are pulled either from our private VoxelAssembler registry or from Docker Hub in the case of the Aspire image.

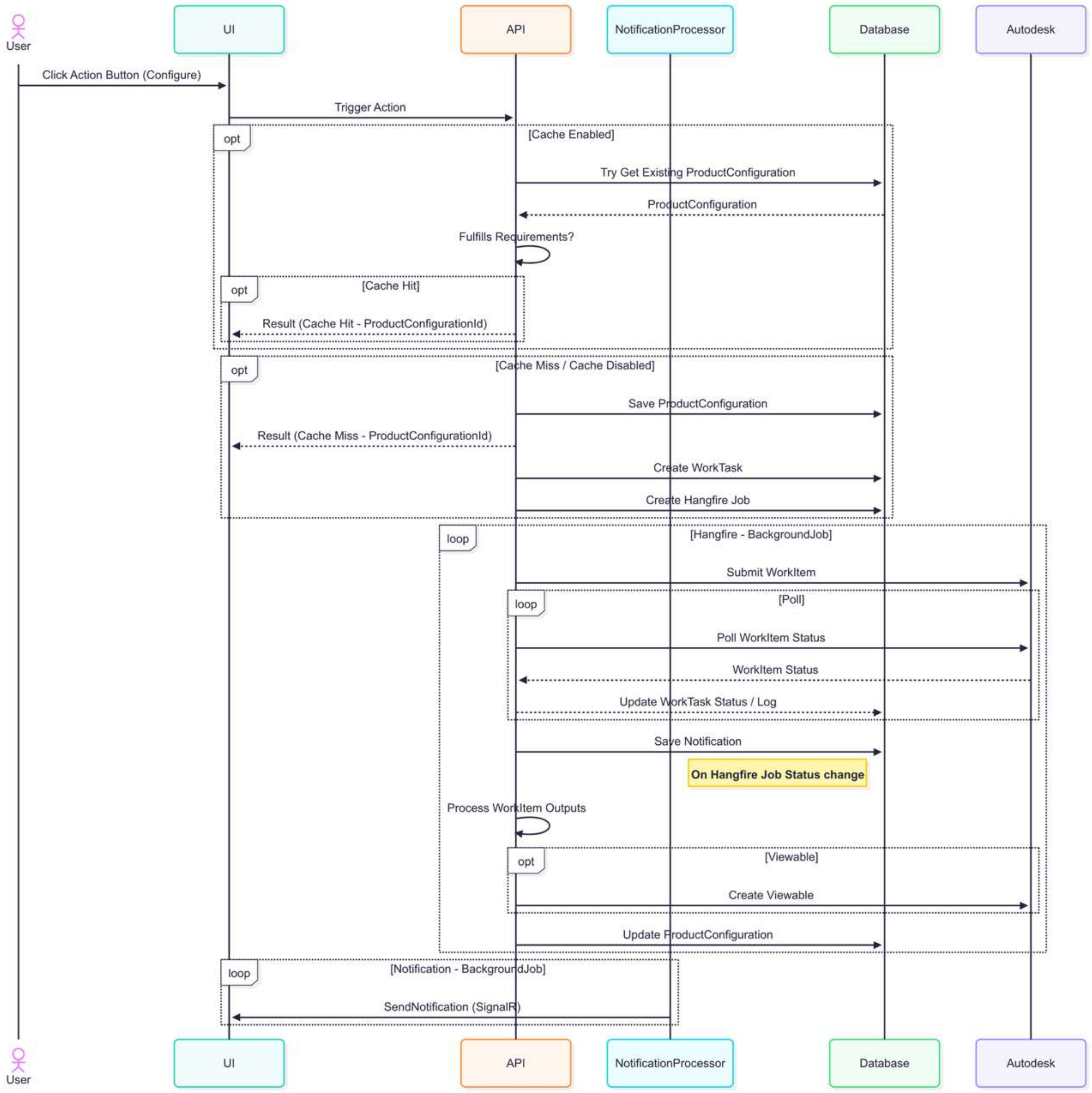
For inter-service communication inside ECS, we use AWS Cloud Map. Each ECS service is registered within a Cloud Map namespace, which enables service discovery via DNS. This allows services to communicate with each other reliably inside the VPC. For example, the WebHost service connects to the Aspire service over gRPC using the Cloud Map service endpoint, without requiring any public exposure.

All sensitive configuration details and secrets are managed centrally in AWS Secrets Manager. To keep track of resource usage and application health, ECS resources are monitored through CloudWatch.
Incoming traffic is handled by an Application Load Balancer, which also resides in the public subnet. It listens on port 443 and routes requests based on the host header: requests for monitoring-voxelassembler.mas-se.net are forwarded to the Aspire service, while those for voxelassembler.mas-se.net are sent to the WebHost service. Any HTTP requests are automatically redirected to HTTPS. The TLS certificate for our domain, *.mas-se.net, is stored in AWS Certificate Manager.
The domain itself was purchased directly through AWS. DNS is managed with Route 53, where we maintain a hosted zone. The A Record points to the Application Load Balancer, while a CNAME Record was used for the DNS validation.



8.7 Application Flow Sequence Diagram

This sequence diagram represents the main flow of our application. It describes how an action is triggered by clicking an Action Button (Configure) on a product configuration. The diagram shows the complete interaction between the User, UI, API, Notification Processor, Database, and the external service Autodesk. It covers optional cache handling, background job execution with Hangfire, polling of Autodesk work items, updating product configurations, and sending notifications back to the UI.


User
UI
API
NotificationProcessor
Database
Autodesk
Click Action Button (Configure)
Trigger Action
opt
[Cache Enabled]
Try Get Existing ProductConfiguration
ProductConfiguration
Fulfills Requirements?
opt
[Cache Hit]
Result (Cache Hit - ProductConfigurationId)
opt
[Cache Miss / Cache Disabled]
Save ProductConfiguration
Result (Cache Miss - ProductConfigurationId)
Create WorkTask
Create Hangfire Job
loop
[Hangfire - BackgroundJob]
Submit WorkItem
loop
[Poll]
Poll WorkItem Status
WorkItem Status
Update WorkTask Status / Log
Save Notification
On Hangfire Job Status change
Process WorkItem Outputs
opt
[Viewable]
Create Viewable
Update ProductConfiguration
loop
[Notification - BackgroundJob]
SendNotification (SignalR)
User
UI
API
NotificationProcessor
Database
Autodesk

9 Implementation

9.1 Feature Implementation

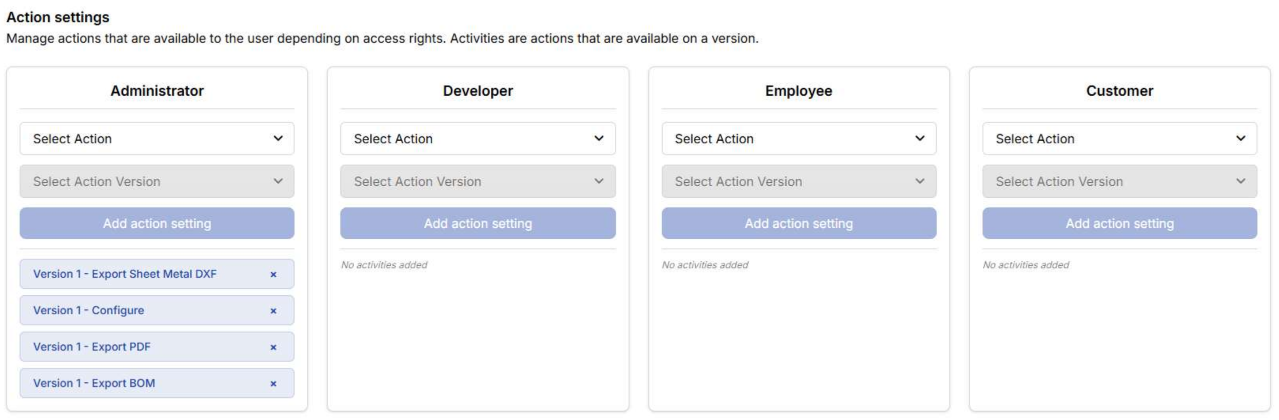
All the features and their implementation status are listed in the table below. For some features, that are bigger than others and need some more explanation, a description of how the feature was implemented is written below the table.

Feature Name	**Feature Type**	**Is Implemented**
Authentication & Authorization	Must-Have	☒
APS Authentication	Must-Have	☒
Category & Product Management	Must-Have	☒
Add App Bundles	Must-Have	☒
Action system	Must-Have	☒
Default Actions	Must-Have	☒
Initialize Buckets	Must-Have	☒
Model Parameter Constraints	Must-Have	☒
Configurator	Must-Have	☒
Configuration Management	Must-Have	☒
Account Settings	Must-Have	☒
Cache Settings	Must-Have	☒
Change Language	Must-Have	☒
User Documentation	Must-Have	☒
iLogic Library	Must-Have	☒
Pricing	Must-Have	☒
Toggle Categories & Products Visibility	Optional	☒
3D Viewer Fullscreen	Optional	☒
Design Settings	Optional	☒
Anonymous Users	Optional	☐
Domain Whitelist	Optional	☒
Clear Cache	Optional	☒
Create Actions from App Bundles	Optional	☒
3D Viewer Extensions	Optional	☒
Allow running instance	Optional	☐
Manage Bucket Items	Optional	☒
Add native data to vault	Optional	☐
Configuration sidebar	Optional	☒
Cloud hosting	Optional	☒
Work Tasks	Optional	☒
Work Tasks Administrator	Optional	☒
Load Configuration	Optional	☒
3D Viewer Default Perspective	Optional	☒
User Management	Optional	☒
Zip Assembly for VoxelAssembler	Optional	☒
Inventor Add-In with CI/CD	Optional	☒

9.1.1 Category & Product Management

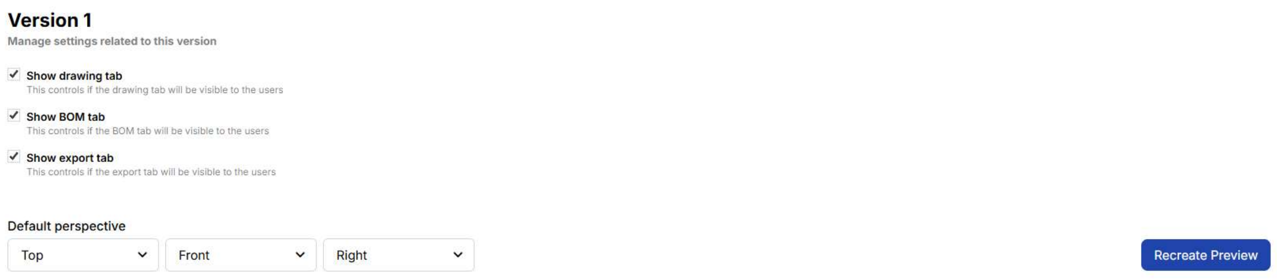
A category has a name, a description and optional an icon (a default icon will be displayed if no icon is provided by the user). A category can have a parent category which will result in a tree. This tree will be visible for the end user on the catalog page.

Products also have a name, description and an optional icon. They can also have a parent category and will act as leaves. A product is the base to start a configuration. Every product can have n versions and will have a default version on creating the product. These product versions have states to control their visibility. These states can be “Active”, “Inactive”, “Internal” or “Development”. Only administrators and developers can see inactive and development versions. Internal product versions are visible for company users, developers and administrators. The user can assign actions to a product version and determine which user roles see which actions in the configurator. This can be different from product version to product version.


Action settings
Manage actions that are available to the user depending on access rights. Activities are actions that are available on a version.

Administrator	Developer	Employee	Customer
Select Action	Select Action	Select Action	Select Action
Select Action Version	Select Action Version	Select Action Version	Select Action Version
Add action setting	Add action setting	Add action setting	Add action setting
Version 1 - Export Sheet Metal DXF ×	No activities added	No activities added	No activities added
Version 1 - Configure ×			
Version 1 - Export PDF ×			
Version 1 - Export BOM ×			

The user is also able to create a 3D preview for a product version and set the default perspective of this preview, as well as control the visibility of the tabs in the configurator. A default 3D preview called Viewable is created automatically.


Version 1
Manage settings related to this version

✓ Show drawing tab
This controls if the drawing tab will be visible to the users

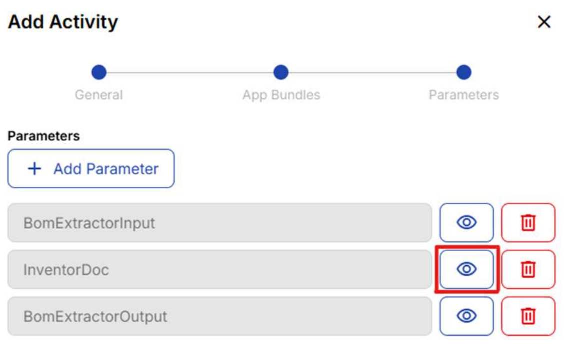
✓ Show BOM tab
This controls if the BOM tab will be visible to the users

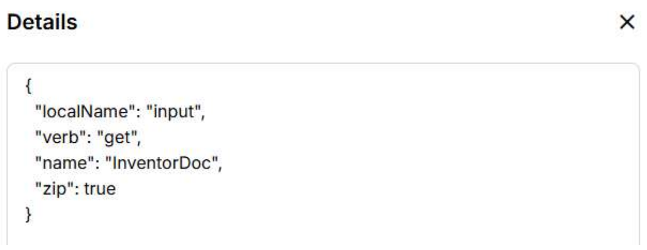
✓ Show export tab
This controls if the export tab will be visible to the users

Default perspective
Top | Front | Right

9.1.2 Action System

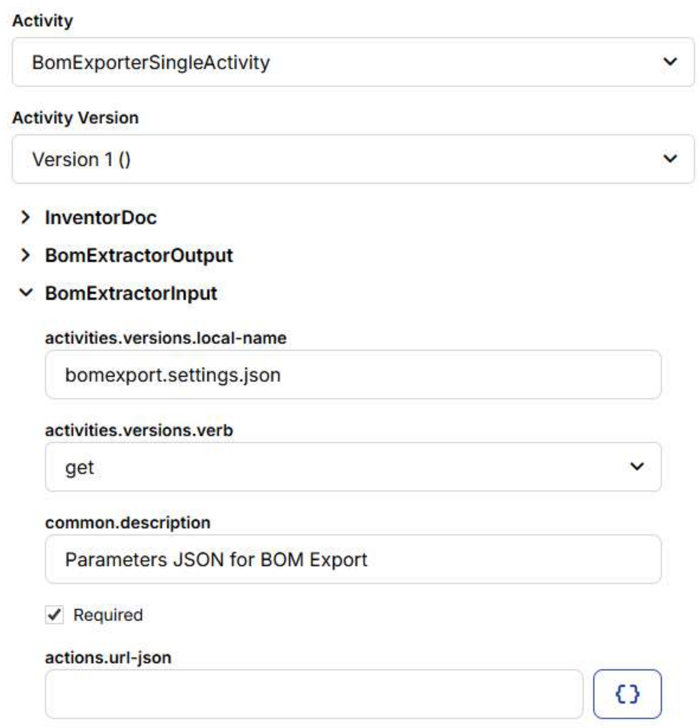
The user can create a so-called activity. An activity has an Id (may only contain letters), an engine to run on (Autodesk Inventor engine versions), an optional alias (can be set manually, will be set by the uploaded bundle if no alias is provided / read from dll by using reflection) and an optional description. On creating an activity, the user must select one or more app bundle versions. In the next step, the app automatically reads the selected app bundles and provides parameters. Parameters can be added manually by the user. These parameters are needed for the Autodesk services to run certain actions. An example parameter - “InventorDoc” – Controls what document will be opened. The user can check the parameters content with a JSON viewer built into the app.




Details

```
{
  "localName": "input",
  "verb": "get",
  "name": "InventorDoc",
  "zip": true
}
```

After creating an activity, the user must create an action. Actions are the buttons that are displayed in the configurator. An action has a name and an activity version. After selecting an activity version, the parameters from before are displayed. The key difference here is, that the user now can set the value to the parameters. As an example, a BomExtractorInput parameter requires additional information in form of a JSON string. The user can paste this JSON string in a JSON helper that will automatically escape it to properly send it to the backend and to the Autodesk services.



Activity
BomExporterSingleActivity

Activity Version
Version 1 ()

activities.versions.local-name
bomexport.settings.json

activities.versions.verb
get

common.description
Parameters JSON for BOM Export

✓ Required

actions.url-json

After setting the necessary values for each parameter, the user can save the action. After that, assigning the action to a product version is necessary, to display the action button in the configurator. This is described in the “Category & Product Management” section.

9.1.3 App Bundles

9.1.3.1 Description

Currently some base AppBundles were implemented. They are called:

- MuM.Inventor.BomExtractor
- MuM.Inventor.Parametrization
- MuM.Inventor.PdfExporter
- MuM.PriceCalculator
- MuM.SheetMetalDxfExporter

They are all based on a template that can be found here: Template

More can be added anytime. They could be just for one customer functionality or something that is useful for multiple. Because of the flexibility VoxelAssembler delivers each customer can have the functionality he needs.

Each AppBundle must implement an Interface called ‘IAppBundleDescriptor’ which allows to define Input and Output.

The VoxelAssembler allows to upload those AppBundles. An app bundle has an Id (may only contain letters), an engine to run on (Autodesk Inventor engine versions), an optional alias (can be set manually, will be set by the uploaded bundle if no alias is provided / read with reflection from DLL) and an optional description. The user can upload ZIP packages including the app bundle. App bundles have versions. For each version added, a new ZIP file must be uploaded. It is not possible to change the once uploaded ZIP.

9.1.3.2 Installation

There is no installation needed for building AppBundles. If new AppBundles need to be created the installation of this Template is recommended.

9.1.3.3 CI/CD

There is currently no CI/CD Pipeline. Still there is some automation. Because each AppBundle must follow a certain structure, this is automatically done when building. The output are multiple zip files for each AppBundle. Releases are manually put here. Note that each AppBundle has its own version.

9.1.3.4 Documentation

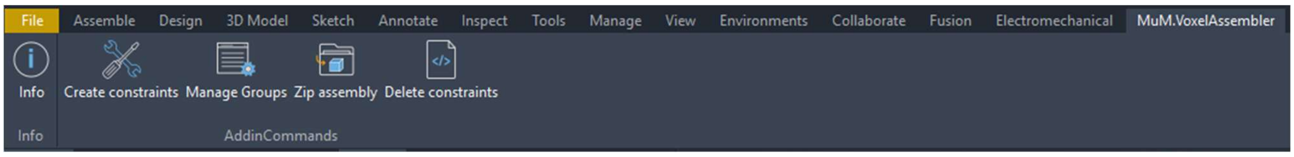
More documentation can be found directly in the customer documentation.

9.1.4 Inventor Add-In & Model Parameter Constraints

Model Parameter Constraints were implemented as a library with a CI / CD Pipeline. This will run on every commit and build a NuGet package that gets directly pushed to the GitLab registry. This is mainly done to allow other project to consume this library and having up to date constraint types.

9.1.4.1 Description

The Inventor Add-In is called MuM.VoxelAssembler and is built like a typical Inventor Addin. As defined by Autodesk an Interface called ApplicationAddInServer must be implemented.



The model parameter constraints NuGet is used to get access to the different constraints. This is an Inventor Addin used to create Model Parameter Constraints in a UI friendly way. It also helps to create a zip that is named correctly and packages everything that is needed for the VoxelAssembler.

9.1.4.2 Installation

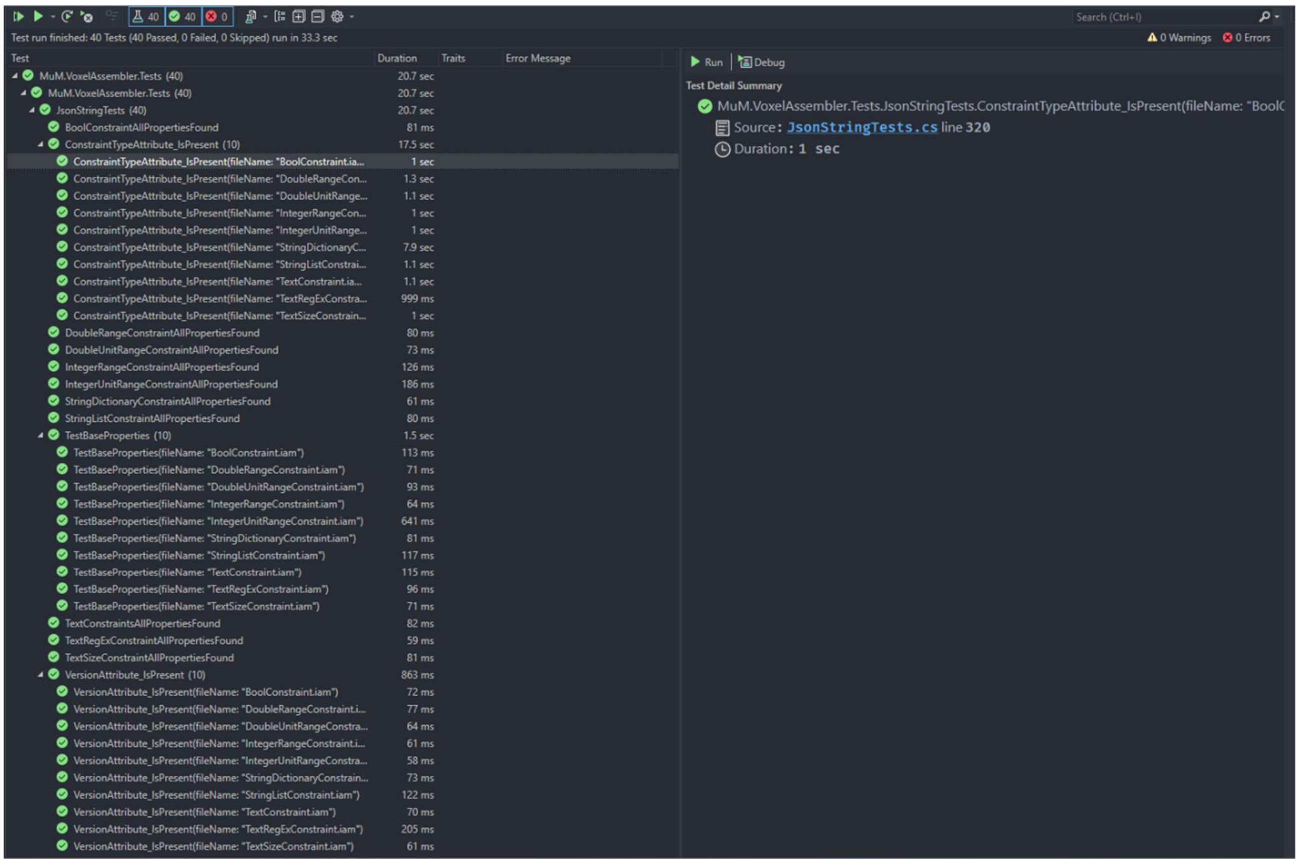
There is an installer that copies everything to the right place. The installer can be found in the Releases.

9.1.4.3 CI/CD

The master branch is protected. Create a new branch, like feature/xyz and merge it to master when finished. Create a tag like release/v0.0.2 to run the pipeline. The pipeline will create a release and upload the installer to the release. The installer can be found in the Releases.
The installer uses wix to create the msi. Make sure that the ci/cd pipeline runs on a windows machine and has wixtoolsets available. They should be installed through the NuGet packages and copied by a target in the csproj file.

9.1.4.4 Testing

There is a project called MuM.VoxelAssembler.Tests which contains tests. Those are mostly used to check if the produced Json strings are valid. It uses input files for each constraint that has one parameter with attributes. To run these tests an Inventor installation is needed. The tests cannot run in the pipeline because of this. A valid license is also needed.



```
Test run finished: 40 Tests (40 Passed, 0 Failed, 0 Skipped) run in 33.3 sec          ⚠ 0 Warnings  ✖ 0 Errors

Test                                                                    Duration  Traits  Error Message
◢ ✓ MuM.VoxelAssembler.Tests (40)                                       20.7 sec
  ◢ ✓ MuM.VoxelAssembler.Tests (40)                                     20.7 sec
    ◢ ✓ JsonStringTests (40)                                            20.7 sec
        ✓ BoolConstraintAllPropertiesFound                              81 ms
      ◢ ✓ ConstraintTypeAttribute_IsPresent (10)                        17.5 sec
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "BoolConstraint.ia...      1 sec
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "DoubleRangeCon...        1.3 sec
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "DoubleUnitRange...       1.1 sec
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "IntegerRangeCon...       1 sec
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "IntegerUnitRange...      1 sec
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "StringDictionaryC...     7.9 sec
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "StringListConstrai...    1.1 sec
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "TextConstraint.ia...     1.1 sec
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "TextRegExConstra...      999 ms
          ✓ ConstraintTypeAttribute_IsPresent(fileName: "TextSizeConstrain...     1 sec
        ✓ DoubleRangeConstraintAllPropertiesFound                       80 ms
        ✓ DoubleUnitRangeConstraintAllPropertiesFound                   73 ms
        ✓ IntegerRangeConstraintAllPropertiesFound                      126 ms
        ✓ IntegerUnitRangeConstraintAllPropertiesFound                  186 ms
        ✓ StringDictionaryConstraintAllPropertiesFound                  61 ms
        ✓ StringListConstraintAllPropertiesFound                        80 ms
      ◢ ✓ TestBaseProperties (10)                                       1.5 sec
          ✓ TestBaseProperties(fileName: "BoolConstraint.iam")          113 ms
          ✓ TestBaseProperties(fileName: "DoubleRangeConstraint.iam")   71 ms
          ✓ TestBaseProperties(fileName: "DoubleUnitRangeConstraint.iam")  93 ms
          ✓ TestBaseProperties(fileName: "IntegerRangeConstraint.iam")  64 ms
          ✓ TestBaseProperties(fileName: "IntegerUnitRangeConstraint.iam")  641 ms
          ✓ TestBaseProperties(fileName: "StringDictionaryConstraint.iam")  81 ms
          ✓ TestBaseProperties(fileName: "StringListConstraint.iam")    117 ms
          ✓ TestBaseProperties(fileName: "TextConstraint.iam")          115 ms
          ✓ TestBaseProperties(fileName: "TextRegExConstraint.iam")     96 ms
          ✓ TestBaseProperties(fileName: "TextSizeConstraint.iam")      71 ms
        ✓ TextConstraintsAllPropertiesFound                             82 ms
        ✓ TextRegExConstraintAllPropertiesFound                         59 ms
        ✓ TextSizeConstraintAllPropertiesFound                          81 ms
      ◢ ✓ VersionAttribute_IsPresent (10)                               863 ms
          ✓ VersionAttribute_IsPresent(fileName: "BoolConstraint.iam")  72 ms
          ✓ VersionAttribute_IsPresent(fileName: "DoubleRangeConstraint.i...   77 ms
          ✓ VersionAttribute_IsPresent(fileName: "DoubleUnitRangeConstra...    64 ms
          ✓ VersionAttribute_IsPresent(fileName: "IntegerRangeConstraint.i...  61 ms
          ✓ VersionAttribute_IsPresent(fileName: "IntegerUnitRangeConstra...   58 ms
          ✓ VersionAttribute_IsPresent(fileName: "StringDictionaryConstrain... 73 ms
          ✓ VersionAttribute_IsPresent(fileName: "StringListConstraint.iam")   122 ms
          ✓ VersionAttribute_IsPresent(fileName: "TextConstraint.iam")         70 ms
          ✓ VersionAttribute_IsPresent(fileName: "TextRegExConstraint.iam")    205 ms
          ✓ VersionAttribute_IsPresent(fileName: "TextSizeConstraint.iam")     61 ms

Test Detail Summary
✓ MuM.VoxelAssembler.Tests.JsonStringTests.ConstraintTypeAttribute_IsPresent(fileName: "BoolC
  Source: JsonStringTests.cs line 320
  Duration: 1 sec
```

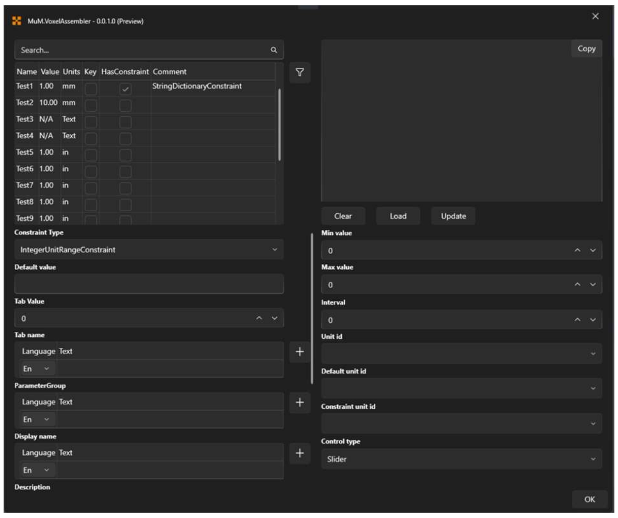
9.1.4.5 Documentation

The documentation for customers is part of the VoxelAssembler Docusaurus project.

9.1.4.6 Zip Assembly

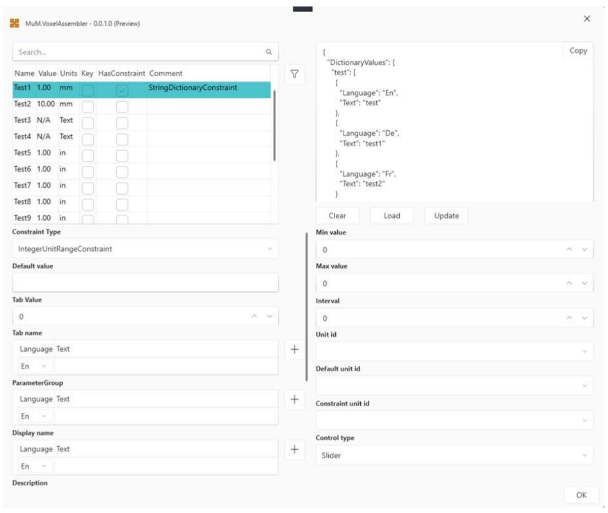
This is now handled by an external console application. The reason for this is that pack and go uses Apprentice internally and thus should not be used inside of an add-in. For this to work some things are important. The PackAndGo project needs to run as AnyCPU and with Prefer 32 bit unchecked. Also, the app.manifest is needed otherwise an error will be displayed.

9.1.4.7 Screenshots (Dark & Light Theme)



Name	Value	Units	Key	HasConstraint	Comment
Test1	1.00	mm	☐	✓	StringDictionaryConstraint
Test2	10.00	mm	☐	☐	
Test3	N/A	Text	☐	☐	
Test4	N/A	Text	☐	☐	
Test5	1.00	in	☐	☐	
Test6	1.00	in	☐	☐	
Test7	1.00	in	☐	☐	
Test8	1.00	in	☐	☐	
Test9	1.00	in	☐	☐	

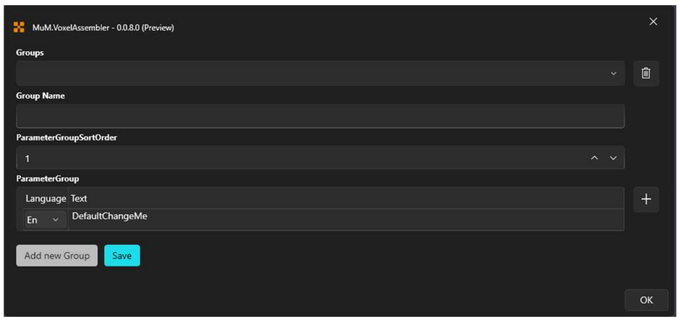
Constraint Type: IntegerUnitRangeConstraint
Default value:
Tab Value: 0
Tab name: Language Text / En
ParameterGroup: Language Text / En
Display name: Language Text / En
Description
Min value: 0
Max value: 0
Interval: 0
Unit id
Default unit id
Constraint unit id
Control type: Slider



Name	Value	Units	Key	HasConstraint	Comment
Test1	1.00	mm	☐	✓	StringDictionaryConstraint
Test2	10.00	mm	☐	☐	
Test3	N/A	Text	☐	☐	
Test4	N/A	Text	☐	☐	
Test5	1.00	in	☐	☐	
Test6	1.00	in	☐	☐	
Test7	1.00	in	☐	☐	
Test8	1.00	in	☐	☐	
Test9	1.00	in	☐	☐	

```
{
  "DictionaryValues": {
   "test": [
    {
     "Language": "En",
     "Text": "test"
    },
    {
     "Language": "De",
     "Text": "test1"
    },
    {
     "Language": "Fr",
     "Text": "test2"
    }
```

Constraint Type: IntegerUnitRangeConstraint
Default value:
Tab Value: 0
Tab name: Language Text / En
ParameterGroup: Language Text / En
Display name: Language Text / En
Description
Min value: 0
Max value: 0
Interval: 0
Unit id
Default unit id
Constraint unit id
Control type: Slider



Groups:
Group Name:
ParameterGroupSortOrder: 1
ParameterGroup: Language Text / En DefaultChangeMe

9.1.5 iLogic Library

9.1.5.1 Description

MuM.ILogicLibrary is an extended library designed to enhance the capabilities of iLogic in Autodesk Inventor. It provides a set of utilities for automating and configuring assemblies, facilitating operations such as transforming occurrences, managing rotations, translations, and logging, especially within Autodesk Platform Service.

9.1.5.2 Features

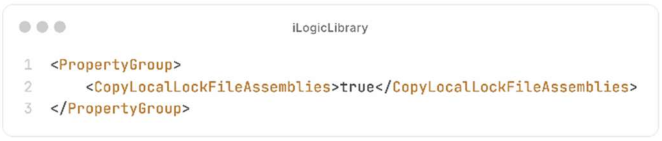
- Translation and Rotation: Perform absolute and relative movements of occurrences in an assembly.
- Logging: Offers advanced logging mechanisms to trace actions and track operations during iLogic script execution.
- Autodesk Platform Service Compatibility: Fully compatible with Autodesk Platform Service environments, allowing iLogic automation to function seamlessly both locally and in the cloud.

9.1.5.3 Installation

How to install is described here.

9.1.5.4 CI/CD

This project creates a NuGet package that can be referenced in AppBundles. In this case it is at least used for the parametrization AppBundle. Make sure to add this to the project:



```xml
<PropertyGroup>
    <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
</PropertyGroup>
```

This makes sure that the DLL file is included while building. Otherwise, it might be missing in the AppBundle output, because it is not directly referenced in the code.

9.1.5.5 Documentation

The documentation can be found here.

9.2 Work Task System & Background Jobs

For all the long running task that can be triggered by an action, an entity called WorkTask was implemented. When talking about the Automation API, Autodesk call their own task WorkItem. It is important to understand the difference of implementation.

9.2.1 WorkTask

WorkTask is what we call the Hangfire job. Those are kept track by the VoxelAssembler.
The WorkTask starts a WorkItem and keeps track of the state by polling.

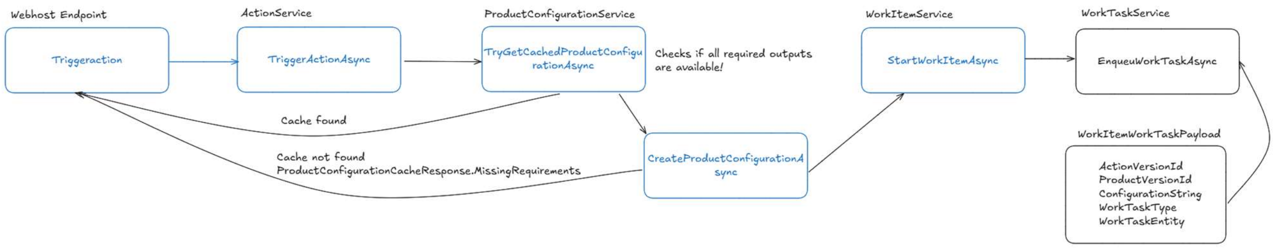
WorkTaskId is a GUID and looks something like this: 5f514de7-1017-4386-9301-34e161dfbc7d

9.2.2 WorkItem

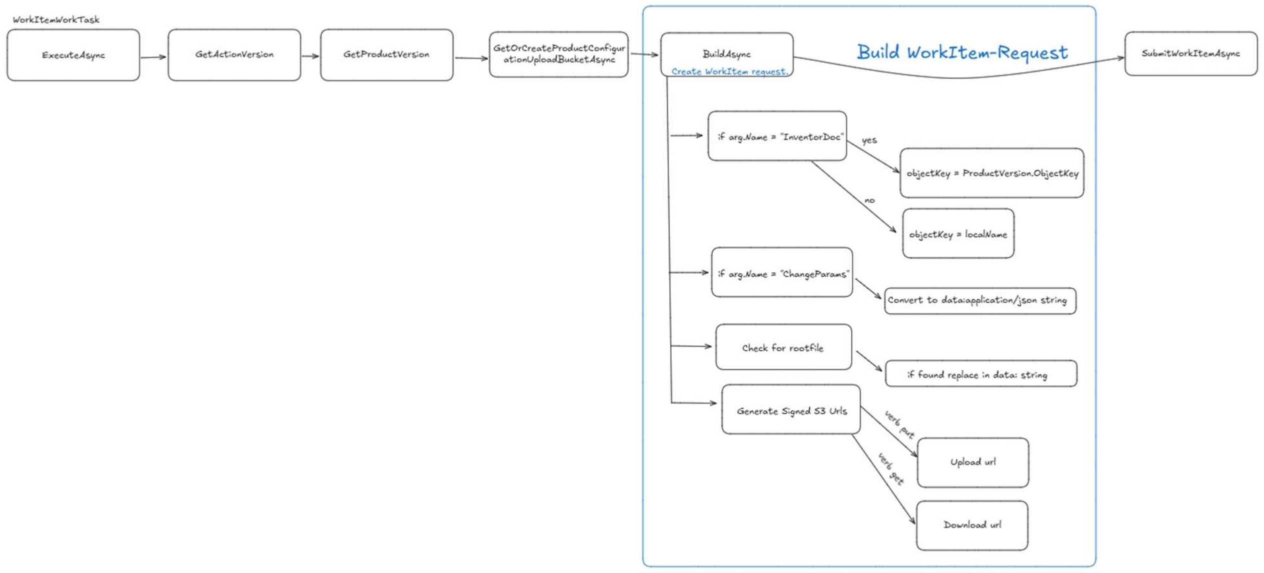
WorkItem is what Autodesk calls their tasks running on Autodesk Platform Service. WorkTask is a wrapper around them that allows to keep track of them in the VoxelAssembler.

WorkItemId is a string and looks something like this: 39c50d3a3e17482db113adb62807515f

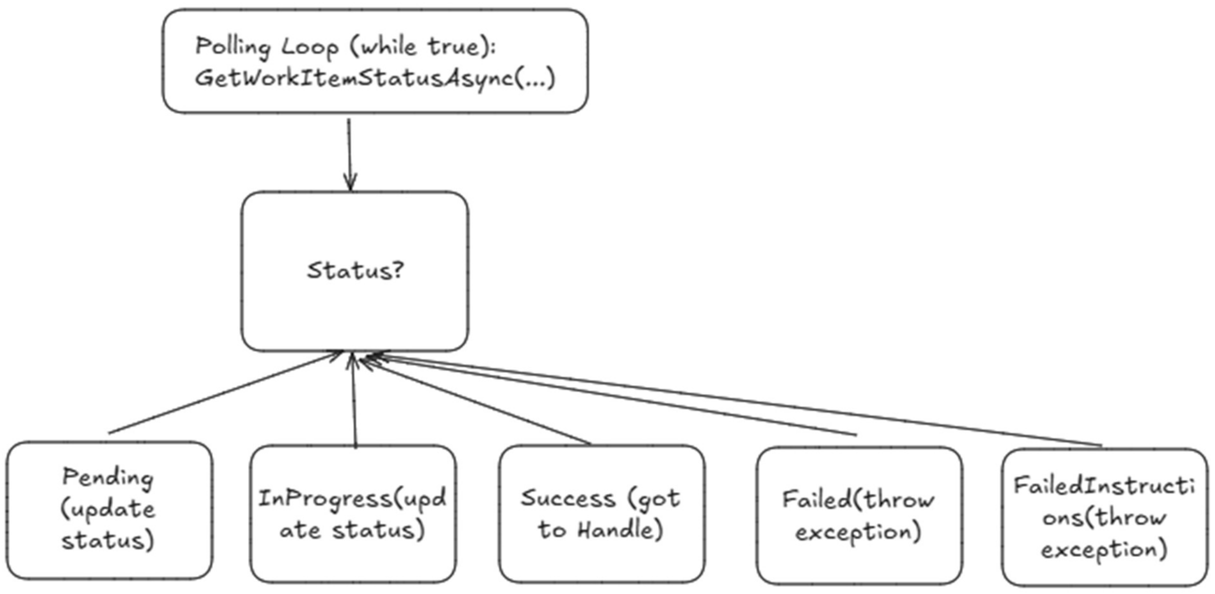
9.2.3 Trigger



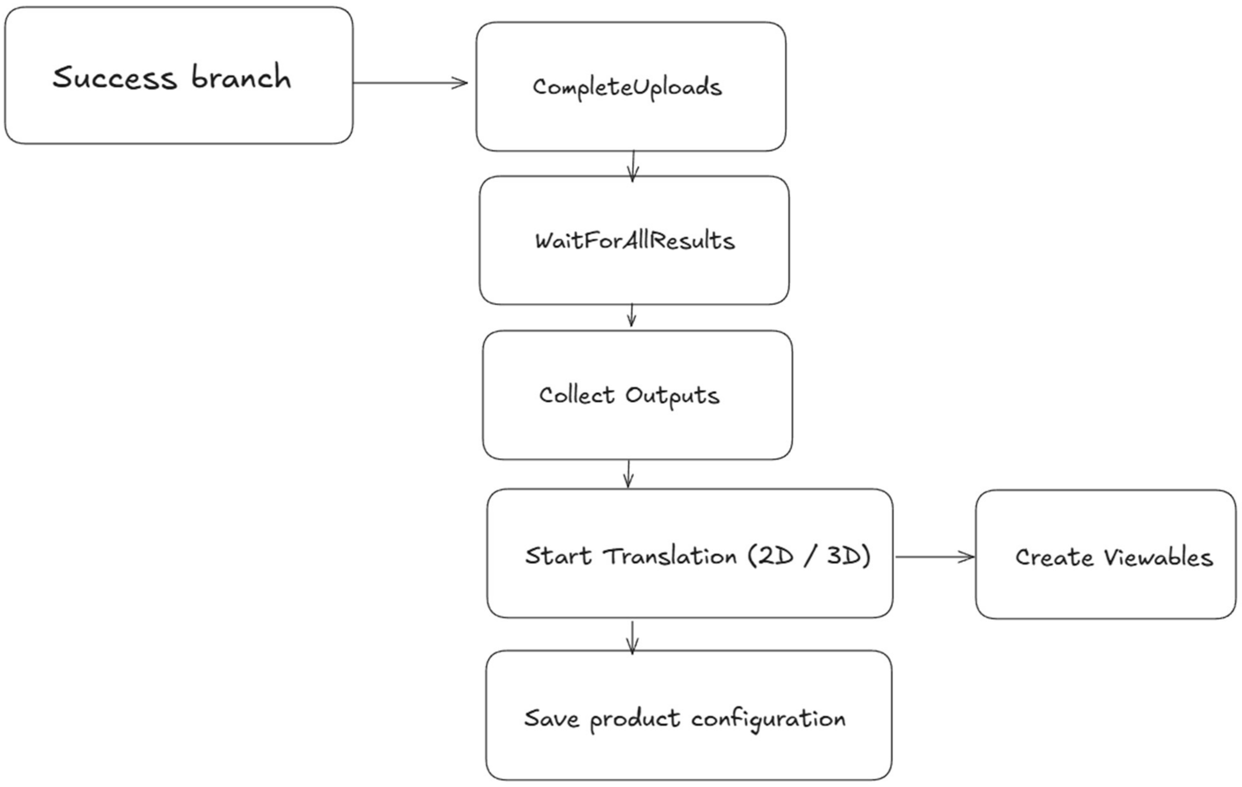
Now only one cache system is implemented, but more could be added using the interface ICacheSystem. An endpoint reacts on an action clicked, checks if cache has everything needed for this action. If yes, a positive response is sent back and directly loaded. If no cache was found or at least is not complete a negative is sent back and a WorkItem is started. This is handled by Hangfire.



After this a polling is started to check for the state of this WorkItem.


Polling Loop (while true):
GetWorkItemStatusAsync(...)
Status?
Pending (update status)
InProgress(update status)
Success (got to Handle)
Failed(throw exception)
FailedInstructions(throw exception)

On success the following steps are done:



9.3 Deployment Strategy

For deployment, we decided to bundle the UI and the API together and ship them as a single Docker image. This approach follows a lock-step deployment strategy, meaning that the UI cannot be deployed independently of the API (and vice versa). While this results in a stronger coupling between the two components, it also simplifies operations: the UI does not need to be served separately, and we avoid dealing with CORS or cross-origin configuration issues.

In the future, we may revisit this decision and host the UI independently e.g. in a CDN to achieve more flexibility.

The Docker build process works as follows:

1. First, the npm packages for the UI are restored.
2. Next, the UI is built, producing static assets.
3. Afterwards, the .NET API is built.
4. The compiled static UI output is copied into the wwwroot folder of the API project.

The entry point of the container is the Web API, which then serves both the backend endpoints and the static UI files.

9.3.1 Database Initializer / Design Automation

Both the Database Initializer and the Design Automation project are built and delivered as Docker containers. The Database Initializer is used to apply schema changes and seed initial data such as users and roles, while the Design Automation container handles Autodesk-specific automation tasks. Each container can be executed directly in the CI/CD pipeline or manually when needed, ensuring consistent execution across all environments.

9.4 Repository Overview

This is a high-level overview of the repositories that were created and used during the master thesis. Each of those are later explained in more detail where needed.

9.4.1 Documentation

Internal documentation repository containing decision logs, UI drafts, and feature definitions. Those are now used for this documentation and are no longer maintained. Start here to get an overview of the project. Documentation

9.4.2 Customer Documentation

Customer-facing documentation on how to use the VoxelAssembler and its API. This repository is used to generate the documentation on the website. Here all the information is stored that is relevant to the customer. CustomerDocumenation

9.4.3 App-VoxelAssembler

Main application repository for the VoxelAssembler project. This repository contains the main application code and is used to build the VoxelAssembler application. Mainly consists of the frontend and backend code. Also includes all the tests written. App-VoxelAssembler

9.4.4 MuM.VoxelAssembler

Repository that contains the Inventor-Addin that is used to add model parameter constraints to parameter in a graphical way. MuM.VoxelAssembler

9.4.5 ModelConstraintLibrary

Repository for managing model constraints. This is used as a library for many other repositories. A NuGet is built from this repository and then consumed by other projects.
ModelConstraintLibrary

9.4.6 MuM.iLogicLibrary

Provides an extended iLogic library for Autodesk Inventor. Used by configurations. This is used by the customer to create their own configurations. How this can be used is described in the customer facing documentation. MuM.iLogicLibrary

9.4.7 API Test Client

Repository containing a test client for API interactions. API Test Client

9.4.8 AppBundles

Contains all the developed app bundles used for APS. There is currently a repository for the default app bundles. There are high changes this will later be extend on a per customer basis. If a customer needs some other functionality running on APS a new app bundle can be developed. More details in the accordingly topic. AppBundles

9.4.9 TaskAndIssue-board

Repository used for tracking tasks and issues. Also, Milestone planning was done here. Issue-Board

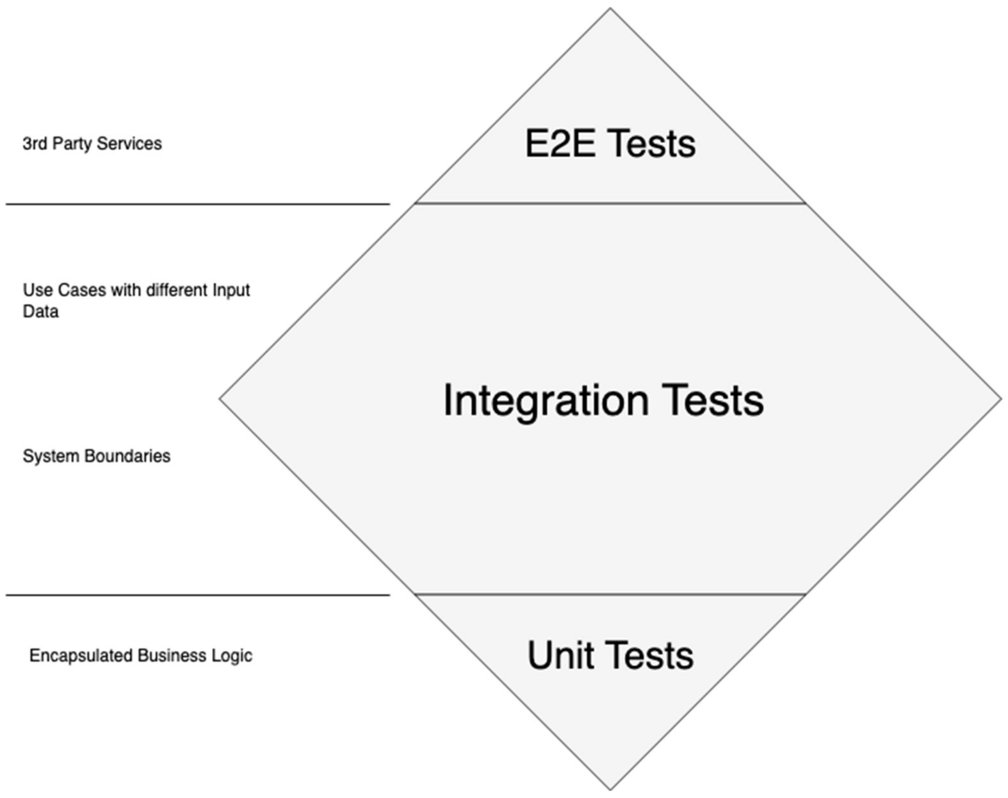
9.4.10 Infrastructure

This repository contains the Infrastructure as Code (IaC) setup for running App-VoxelAssembler on Amazon Web Services (AWS). It uses Terraform as the provisioning tool and defines all necessary AWS resources in a reproducible and version-controlled manner.
The repository serves as the single source of truth for creating, updating, and managing the cloud infrastructure that App-VoxelAssembler depends on.

10 Quality Assurance

10.1 Testing Strategy

The VoxelAssembler application uses a three-tier testing strategy designed to maximize confidence in system reliability while maintaining efficient development workflows. Our approach prioritizes integration testing as the primary testing methodology, supplemented by targeted unit tests and end-to-end tests that validate real external service integrations.
Rather than following the traditional test pyramid with many unit tests, fewer integration tests, and minimal E2E tests, we employ an adapted strategy where integration tests form the primary foundation, supported by focused unit tests and comprehensive end-to-end tests that validate external API integrations.



10.1.1 Unit Tests

In the beginning, we had planned to write more unit tests to cover isolated business logic. However, as the project progressed, we decided to focus more on covering individual business use cases through integration tests instead, since we no longer saw much benefit in adding additional unit tests. We kept the existing unit tests, as they are still useful in certain cases, such as architecture tests (e.g., checking which projects reference each other) or logic that would otherwise require extensive setup in an integration test. For these unit tests, we typically use an in-memory database, which allows us to keep the setup lightweight while still validating the logic in isolation. Technical components used for testing for unit tests can be found in chapter:

Projects:
VoxelAssembler.Application.Tests
VoxelAssembler.Architecture.Tests

10.1.2 Integration Tests

For our integration tests, we use the WebApplicationFactory to spin up our API in a realistic test environment. Alongside this, we start a PostgreSQL database in Docker and seed it with the necessary data to simulate real-world scenarios. The tests are executed against the running API using an HttpClient, where we log in, obtain a session cookie, and then include this cookie in subsequent requests to ensure proper authentication and authorization handling.

Our goal is to cover all business use cases end-to-end within this controlled environment. This includes both positive scenarios (verifying that valid operations succeed) and negative scenarios (for example, ensuring that user roles and permissions are correctly enforced). By doing so, we can validate not only the business logic but also the full request/response flow, data persistence, and security aspects.

Technical components used for testing for integration tests can be found in chapter:

Projects:
VoxelAssembler.WebHost.Tests

10.1.3 End-to-End Tests

The end-to-end (E2E) tests are structured in the same way as the integration tests and are treated similarly in the codebase, which keeps the overall testing approach consistent and easy to maintain. The main difference is that, unlike integration tests which work against our own system boundaries, the E2E tests call real Autodesk services. This allows us to validate not only our own business logic and API contracts, but also the correct interaction with external systems in a production-like scenario. As a result, these tests provide an additional layer of confidence that our solution works reliably in real-world conditions.
E2E tests use of the same components as the integration tests.

Projects:
VoxelAssembler.WebHost.Tests

10.1.4 UI Tests

In addition to unit, integration and end-to-end tests, we added UI tests to validate the user interface. The purpose of these tests is to make sure that the user interactions work as expected. Currently, only a few tests are written, such as authentication & authorization, category & product management, user management, account settings and email provider settings. More tests will be added in the future.

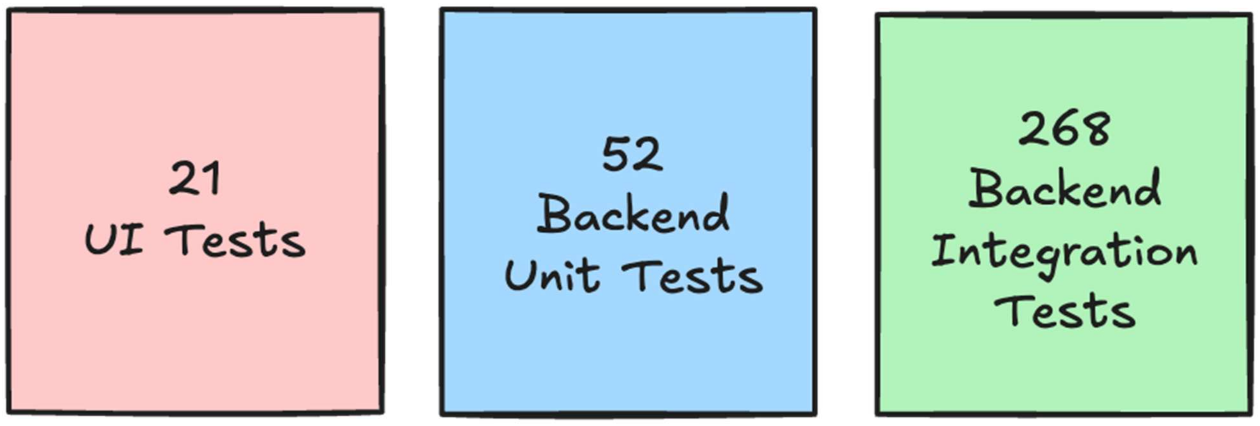
The tests are written with Vitest, since it's a very fast and lightweight testing framework and since we're already using Vite as build tool, it was the obvious choice for us to use Vitest here. To simulate API requests, we use the MSW (Mock Service Worker) package which allows us to prepare API responses so we can easily verify how the frontend reacts to different API responses. This setup provides a good and valid feedback during the development of the frontend.

We also discussed about using Playwright to have real browser-based UI tests. This would allow us to automate full user workflows across different browsers in a production-like environment. Since Playwright comes with a much higher setup and maintenance effort, we decided to move that to the Stretch-Goals milestone. It is something that we would implement in the future development of the application.

Project:
VoxelAssembler.UI (/src/test)

10.1.5 Test Results

The complete test reports can be found in the attachment under "03_Quality_Assurance".



10.1.6 Reviews

At the beginning, we had planned to only allow code contributions through merge requests to enforce reviews from the start. However, we quickly realized that this approach was somewhat cumbersome in the early phases of the project, when development speed and flexibility were more important. As a result, we introduced a shared dev branch that everyone could push to directly.

Even though this branch was not protected by a formal policy, we agreed as a team that every developer would test their changes locally and run the tests before pushing. This ensured a baseline level of quality and reduced the risk of breaking the shared development branch.

Once the project reaches completion, we will merge the dev branch into the main branch, which is protected by a policy that prevents direct pushes. From that point on, contributions will only be possible through merge requests. Each merge request automatically triggers a validation build, which runs the tests and verifies that the solution can be built successfully before the changes are merged. This ensures that code quality and stability are maintained.

10.1.7 Testing Sessions / Manual Tests

After each milestone, we conducted dedicated testing sessions as a team. For this, we scheduled a meeting where we tried out the newly implemented functionality together. Most of the time, this was done directly through the UI, but occasionally we also executed manual API calls against the backend to validate specific scenarios.
The outcome of these testing sessions was usually a set of issues capturing functionality that did not yet work as expected. In addition, the sessions often sparked valuable discussions about user experience for example, how certain workflows could be simplified or made more intuitive for end users. These insights helped us refine the product beyond pure correctness and focus on usability as well.

To ensure transparency and allow others to revisit the findings later, we recorded all testing sessions. Testing sessions were documented and can be found as an attachment in **“\03_Quality_Assurance\01_Testing_Sessions”**.

10.2 Static Analysis and Formatting Tools

To ensure a consistent and maintainable codebase, we enforce strict quality and architectural rules across both the backend (.NET) and the frontend (React/TypeScript).

10.2.1 Backend

Analyzers

We rely on the default Roslyn analyzers together with StyleCop.Analyzers to enforce coding standards and detect potential issues early. All analyzers are included via centralized build props, making sure that every project follows the same rules.

Build enforcement

With TreatWarningsAsErrors enabled, the build only succeeds when no analyzer warnings remain. This prevents “warning debt” from creeping into the codebase.

Package management

We use Centralized Package Management so that package versions stay aligned across all projects, reducing inconsistencies and version conflicts.

Code style & formatting

General style and formatting rules (e.g., avoiding var, prefixing private fields with _) are enforced through .editorconfig. The same configuration also controls the severity of analyzer rules.

Architecture validation

Using NetArchTests.Rules, we validate architectural boundaries through unit tests. For example, in our Clean Architecture setup, dependencies are only allowed to flow inward:
The Domain layer must not depend on Application, Infrastructure, or Web layers.
The Application layer must not depend on Web.

10.2.2 Frontend (React/TypeScript)

Prettier

Handles automated formatting to keep the codebase clean and consistent.

ESLint

Enforces coding standards, detects potential bugs, and ensures code quality in TypeScript/JavaScript.

10.3 DevOps & Build Automation

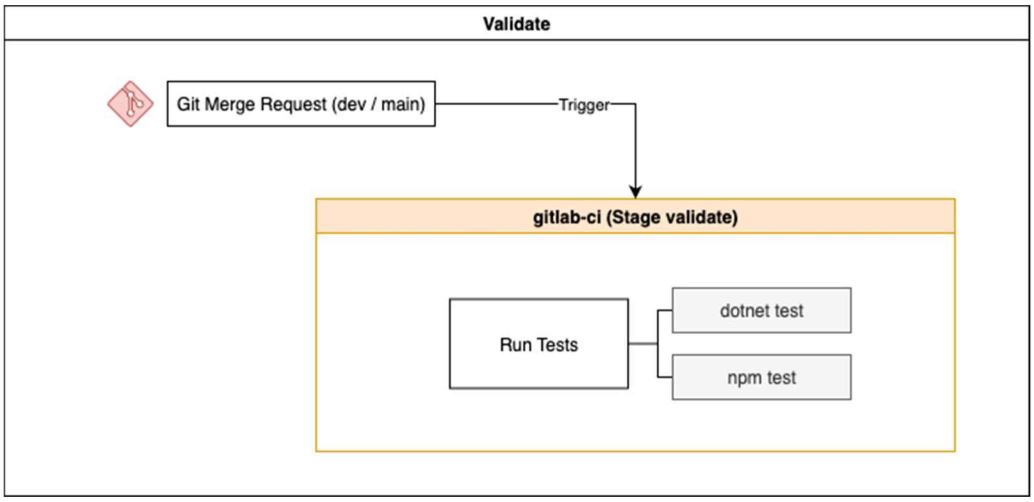
10.3.1 Branch Policies

In our development workflow, we enforce strict branch policies to maintain code quality and consistency. The main branch is protected, which means developers are not allowed to push changes directly to it. Instead, all changes must go through a merge request.

When a merge request is created, the pipeline automatically runs the test suite. Only if all tests pass successfully can the merge request be completed. This ensures that no failing or unstable code is introduced into the main branch.

Looking ahead, we are considering additional measures to further strengthen our process. One idea is to require at least one reviewer, apart from the author, to approve a merge request before it can be merged. Another possible extension would be to introduce a merge request checklist, covering items such as confirming that new tests have been written, linking the relevant issue, and verifying other quality requirements.

10.3.2 Merge Request

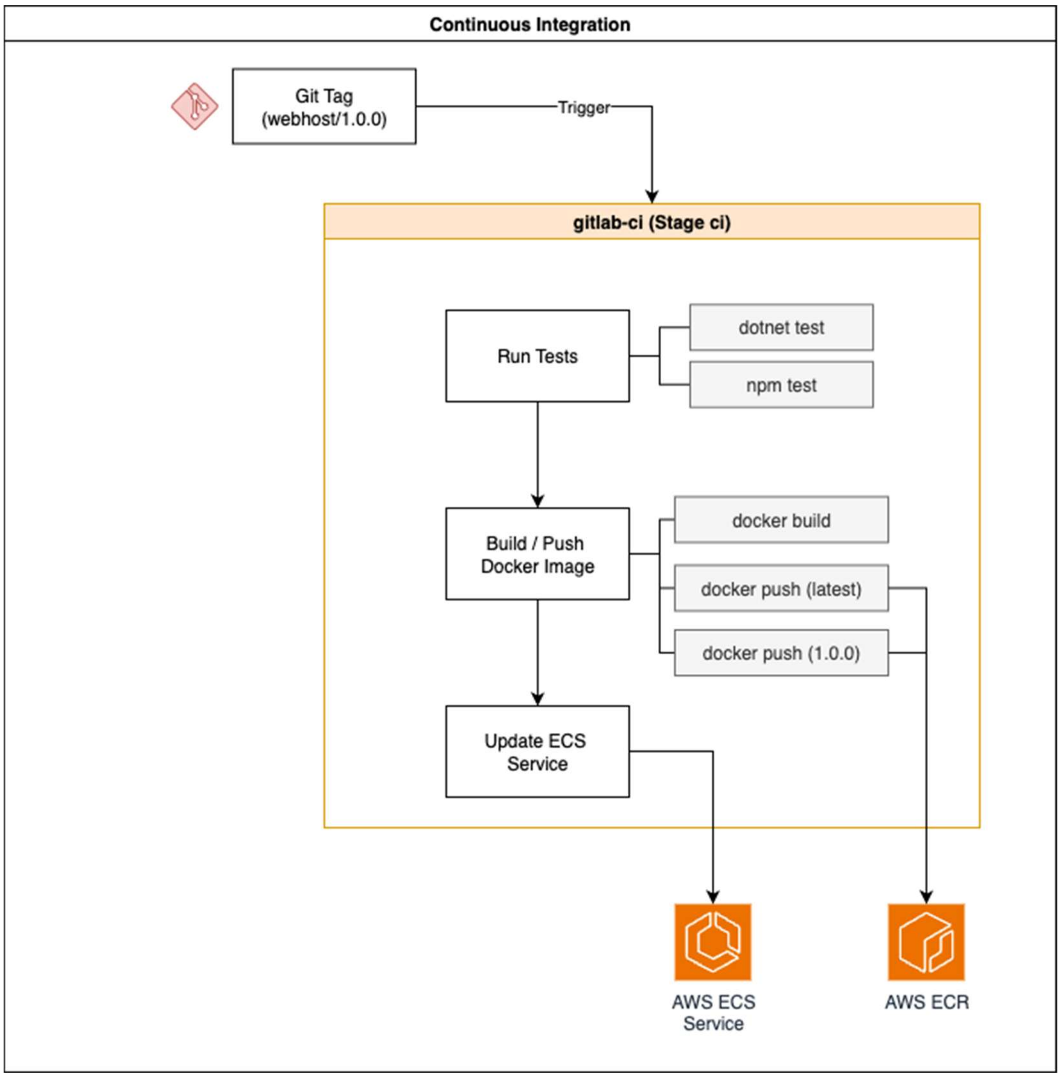


As soon as a merge request is created, the validate stage defined in the .gitlab-ci.yml file is executed. During this stage, the entire .NET solution is built, and the test suite is run. Test results are collected in the JUnit format using the Coverlet library. GitLab requires this format to display test results and statuses directly in the Test Explorer of the merge request.

To complete and merge a request, the pipeline must run successfully — meaning the solution builds without errors and all tests passes. This ensures that only working and verified code can be integrated into the protected branch.

At the moment, we are not yet able to measure or enforce a minimum code coverage threshold. However, this gap has been explicitly tracked as part of our technical debt and is planned to be addressed in the future.

10.3.3 CI / CD



A new version of an application can be released by creating a Git tag, following the principles of Semantic Versioning. Creating such a tag triggers the CI stage defined in the .gitlab-ci.yml file.

The first job executed in this stage runs the test suite, ensuring that no release is created without passing tests. Once this step succeeds, only the Docker image that corresponds to the created tag will be built.

Each application is released independently by tagging it with the application name and version, for example:

- webhost/1.0.0 – Backend API together with the UI
- db-initializer/1.0.0 – Prepares and seeds the database
- designautomation/1.0.0 – Handles Autodesk-specific automation tasks

During pipeline execution, we install the AWS CLI to fetch an authentication token for AWS ECR (our container registry). With this token, we perform a docker login to the registry and then push the generated image.
For each application release, two tags are pushed to the registry:

- the semantic version from the Git tag (e.g. webhost:1.0.0)
- the latest tag (e.g. webhost:latest)

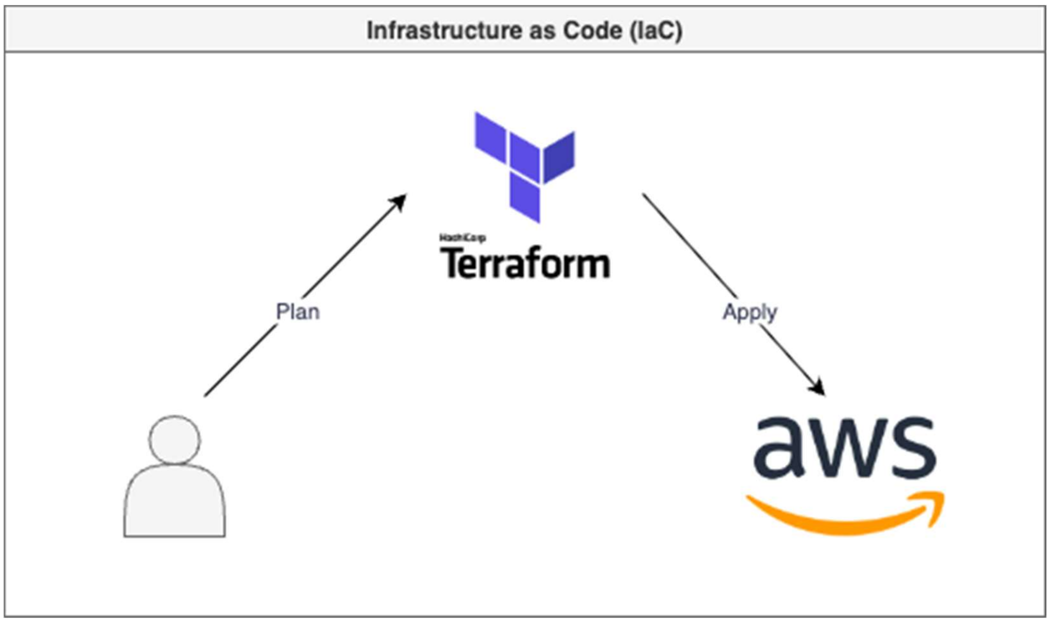
This way, deployments can either follow the moving latest tag for automatic updates, or pin to a specific version tag to allow rollbacks and stable releases.

The version is embedded directly in the Docker image during the build process and then passed as an environment variable to the UI. This allows the application to display its current version directly in the interface, making it easier for users and developers to confirm which release is deployed.

After the Docker image has been pushed, we update the AWS ECS service in the respective cluster. This triggers a redeploy of the service, ensuring that the newest image is used. The deployment is performed as a rolling update, so that new tasks are started with the updated image while old ones are gradually stopped, guaranteeing zero downtime for the application.

10.3.4 Infrastructure

We decided to manage our cloud infrastructure as code in a dedicated repository (IaC). For this, we are using Terraform with the AWS provider to describe our AWS resources. Currently, we are running on a private AWS cloud, but the goal was to manage everything through Terraform so that we can later provision the same AWS setup quickly on another AWS account.
Our rule is that the infrastructure must only be changed or extended through Terraform, never directly in AWS.
At the moment, the Terraform state is stored locally. In the future, proper state management needs to be set up, for example via AWS S3. We have already captured this as a technical debt issue.
Another missing piece is a CI/CD pipeline that ensures terraform fmt has been run (code formatting check) and then executes terraform plan and terraform apply. Currently, we deploy the infrastructure manually via the CLI.



10.3.5 Monitoring

For centralized monitoring we use OpenTelemetry, which enables us to collect logs, traces, and metrics across the system. Within the application we have added extensive logging at meaningful points and relay on the provided instrumentation from ASP.NET Core, EntityFrameworkCore, Hangfire etc. This gives us valuable metrics and traces out-of-the-box, without requiring much additional setup.

For production usage, we plan to integrate the Aspire Dashboard. Microsoft provides the dashboard as a standalone Docker image, which makes it easy to get started. Although it is currently still somewhat limited (for example, it does not yet support connecting to external storage), there are active efforts in the community to extend its capabilities. The deciding factor for us was the simplicity and usability of Aspire, which allows us to quickly adopt monitoring without heavy setup. If Aspire should eventually turn out to be insufficient, we can still switch to a service such as Elastic Cloud. Since the destination for our telemetry data can be changed easily in the code, this migration path is straightforward.

Looking ahead, we may add custom metrics on top of the built-in instrumentation to better analyse user behaviour and gain deeper insights into how the system is being used. This improvement has already been captured as part of our technical debt.

For authentication with the OpenTelemetry Collector (via gRPC), we use an API key. While OAuth would also be supported, API keys provide a simple and effective solution in our setup. Access to the dashboard UI itself is protected with a standard username/password combination. All sensitive credentials, including the API key and the UI password, are securely stored in the AWS Secrets Manager.

10.3.6 Health Checks

In our API, we use the standard ASP.NET Core health checks.

- The endpoint /alive returns whether the API itself is healthy.
- The endpoint /health additionally checks dependencies such as the PostgreSQL database connection. The response is a JSON document containing all evaluated health check entries.

Our goal is to extend this further by implementing custom health checks for all external dependencies, such as the SMTP server and Autodesk services. We have also captured this in the related technical debt issue.

10.3.7 Resilience

Resilience Third Party (APS)

Some APS calls have rate limits that need to be handled. This is done for most API Call's but not yet all of them. This can easily be further extended by using the existing resilience pipeline built with Polly.

Some APS calls are eventual consistent, meaning if they are used to fast after each other, then they can fail. API calls like that were also extended with a resilience pipeline that does retries. Some of the tests we have were run for multiple hours to check them and many, before flaky tests, worked now without errors.

10.4 Technical Debt

We tracked all our technical debt in a dedicated GitLab issue, ensuring that nothing would be forgotten over the course of the project. This issue served as a central backlog for topics that we identified but decided not to address immediately.

Once the team decided to tackle a specific technical debt item, it was extracted into its own GitLab issue with a proper description, estimation, and any necessary technical details. This approach allowed us to manage technical debt in the same structured way as regular features or bug fixes, making it visible, plannable, and actionable rather than leaving it as hidden work.

11 Value and Evaluation

11.1 Value for Users & Companies

Today’s customers increasingly expect products tailored to their specific needs instead of standard, off-the-shelf solutions. Almost every company in the mechanical and manufacturing sector faces demands for configurable products, creating both opportunities and challenges. Today most companies spend a lot of time to create those specialised products, which might be needed for an offer, which then might not even sell.

For example:

- Machinery and Equipment: Customers often require machines in specific sizes, capacities, or with optional add-ons that fit their production line or workshop layout.
- Industrial Components: Pumps, valves, and drive systems frequently need to be adapted to precise flow rates, pressures, or space limitations.
- Automotive and Transportation: Vehicle components, cargo systems, or specialized attachments are often configured to meet customer-specific operational needs.
- Construction and Metalwork: Staircases, railings, ventilation systems, or steel structures are rarely identical across projects, requiring custom dimensions and features.
- Consumer Goods: From custom bicycles to tailored home appliances, even end-users expect products that reflect their personal requirements.

So, for users the value is to be able to configure the product for themselves anytime they want.

For companies the value is saving time configuring products, by doing that for the customer or letting the customer do that by themselves or even together (sales representative with customer).

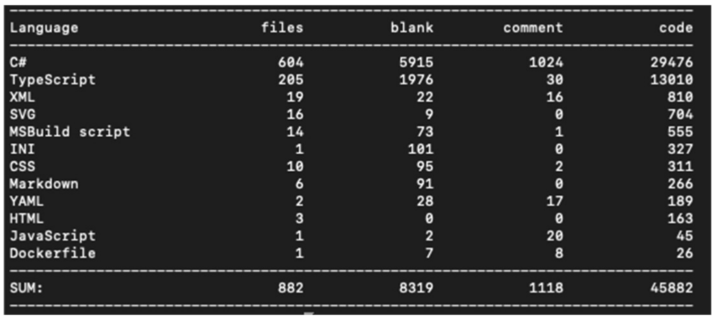
Even if the product cannot be fully customized within the configurator itself, it can still serve as the starting point for the engineering process. The generated result can be downloaded and further refined in a CAD system, allowing engineers or designers to make final adjustments before production.

11.2 Technological Complexity

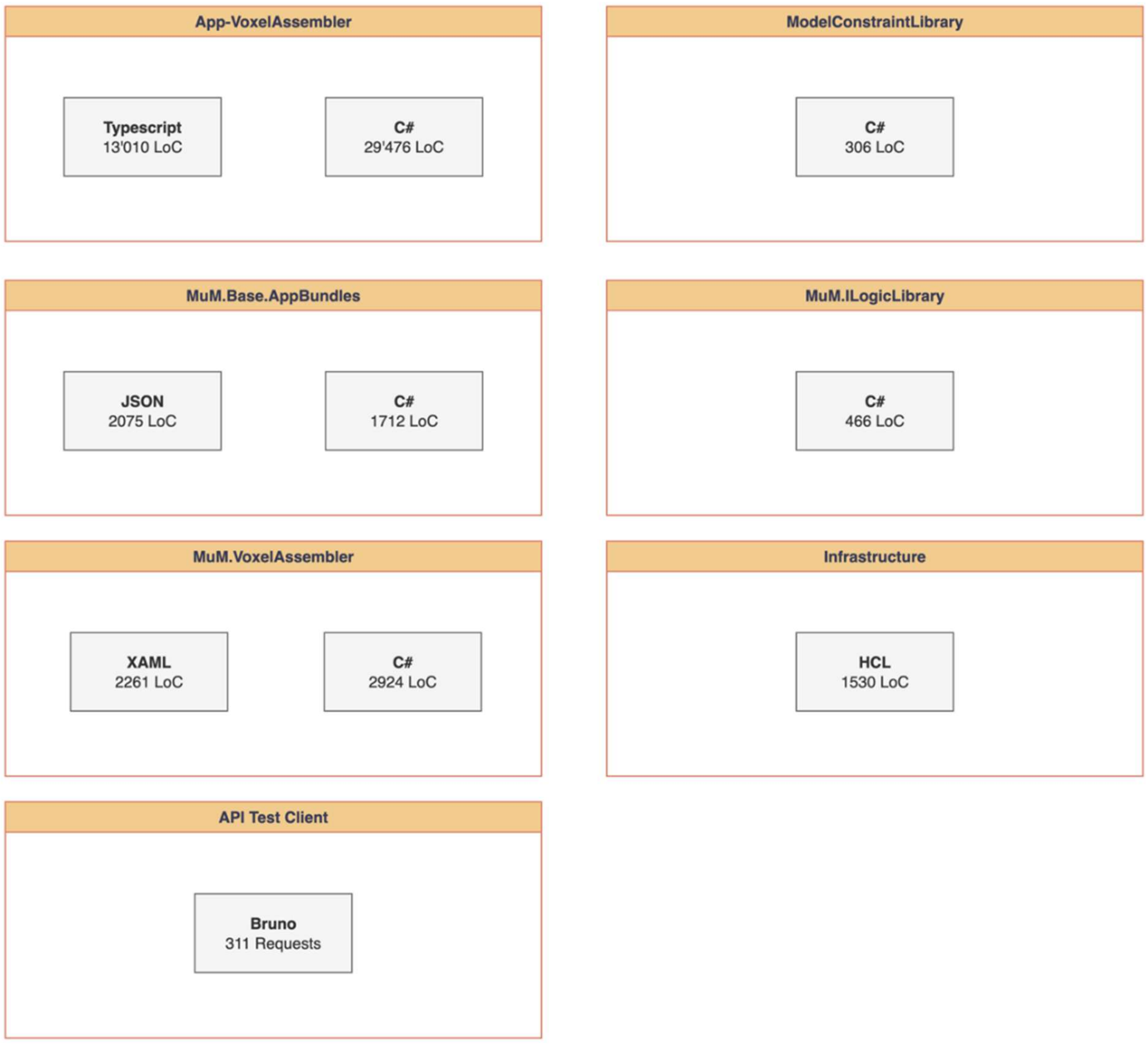
11.2.1 Code Metrics

We measured the lines of code (LoC) in the project using the cloc tool. To ensure accurate results, we excluded common build, dependency, and IDE directories (e.g., node_modules, dist, .git, bin) as well as automatically generated code such as the NSwag client. Additionally, blank lines and commented lines are excluded from the count, so the analysis only reflects the manually written source code and not generated files, build artifacts, metadata, or non-executable lines.

Ex. Output from App-VoxelAssembler repository:



```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C#                             604           5915           1024          29476
TypeScript                     205           1976             30          13010
XML                             19             22             16            810
SVG                             16              9              0            704
MSBuild script                  14             73              1            555
INI                              1            101              0            327
CSS                             10             95              2            311
Markdown                         6             91              0            266
YAML                             2             28             17            189
HTML                             3              0              0            163
JavaScript                       1              2             20             45
Dockerfile                       1              7              8             26
-------------------------------------------------------------------------------
SUM:                           882           8319           1118          45882
-------------------------------------------------------------------------------
```



12 Conclusion

12.1.1 Technical Review

From a technical perspective, the architecture and implementation of VoxelAssembler proved to be robust and well-suited for the project goals. By combining Clean Architecture, Domain-Driven Design, and Vertical Slice Architecture, we achieved a modular, testable, and maintainable system where each feature could be developed independently. This separation of concerns facilitated clear responsibilities within the team and enabled a consistent development flow.

The integration with Autodesk Platform Services worked as expected and confirmed that CAD automation in the cloud is a viable approach. The abstraction via AppBundles, Activities, and Actions allowed us to implement reusable workflows with minimal coupling.

The deployment strategy using Docker and AWS ECS provided reproducibility and scalability. Infrastructure as Code via Terraform allowed consistent environments, although manual steps for Terraform state management remain a pending improvement. Monitoring with OpenTelemetry and Aspire was helpful during development but Aspire still shows limitations for long-term production monitoring.

The chosen testing strategy with a strong focus on integration and E2E tests proved particularly valuable, given the complexity of external services. Nevertheless, unit test coverage remains comparatively low, and automation around CI/CD pipelines could have been established earlier to catch regressions sooner.

Performance and responsiveness of the configurator met expectations. The cache system significantly reduced repeated processing times for identical configurations, but further optimizations can and should be introduced later.

In summary, the technical foundation of VoxelAssembler is sound, extensible, and demonstrates that complex CAD automation can be successfully delivered as a modern web application. At the same time, certain technical debts (monitoring maturity, AppBundle versioning risks, test coverage gaps) have been identified and provide opportunities for future improvement.

12.1.2 Learning Outcome

12.1.2.1 Tobias Wiesendanger

Building web applications is not my daily business. For me this was the first time where I could apply a lot of the things learned in school or privately. Also writing tests is not something I do daily, which used up a lot of time, but while progressing showed how important they are.
I learned how to coordinate a team of multiple people, communicate clearly and create issues that allow to work independently.

I never had the challenge to create something like a client for the frontend. We used NSwag for this which worked great and was a learning experience for me. It allowed us to quickly create a client that could be used directly in the frontend.

At my workplace we often face the challenge to make a project compile and onboard other developers. This is where aspire shine and was used in this project. I learned a lot of its features, but also, it's shortcomings like using it in tests.

12.1.2.2 Roman Schweri

I'm working actively as a web developer since six years. But since we're using .NET and Blazor for every web application in my company, I never really had a project where backend and frontend are strictly separated. I enjoyed being responsible for the frontend only, while knowing that the backend is built structured and organized. Therefore, VoxelAssembler was the first project, where I was a fulltime frontend developer, which was nice. I enjoyed deepening my knowledge in React, TypeScript and TailwindCSS and already started to add it to the tech stack in my company. A big learning was to not start coding blindly but taking a step back and plan the whole project as good as possible. This starts from writing down the key features, to creating use cases, talking about tech stack and choose the proper one instead of always go with the one "we used since day one".
Also new to me was the usage of packages like Hangfire and NSwag, that I'll definitely consider using in my works project from now on, since they provide such a great user & developer experience.
I never really written tests or did anything about CI/CD pipelines. I learned that even though it can be really time consuming to implement, it can pay out in the mid-long term when a pipeline tells you, that your new code broke the existing one and therefore cannot be merged or deployed.
Overall, it was a great experience where I can take a lot into my daily business. I really enjoyed working with Mauro and Tobias, since we all three have a similar way of thinking and all three want to achieve the same result.

12.1.2.3 Mauro Hefti

Looking back, I got to explore quite a few new things in this project. I had worked with CI/CD before, but it was my first time using GitLab and its CI pipelines, which gave me a new perspective on automation and deployment flows. I also dived into Aspire for the first time, not just testing it out, but actually building a real project with it and even hooking up a UI integration inside Aspire. Another highlight was working with AWS hosting for the first time. It gave me insights into how AWS services are structured and managed, and how they compare to Azure, which I was more familiar with before.
On the backend side, I learned how to use resource files (resx) for translation, which turned out to be really nice to use and well integrated in the framework. And with Hangfire, I got some solid hands on experience, not only running background jobs but also experimenting with job filters to customize how they behave. All in all, it was a great mix of first time experiences and deeper dives, and I came away with a bunch of practical knowledge that I can carry into future projects.

12.1.3 Lessons Learned

12.1.3.1 Tobias Wiesendanger

I learned very early how important it is to write tests from the beginning. We benefited a lot from this while further developing. Breaking changes were detected quickly. What I would change regarding to this, is making the CI / CD pipeline work sooner. We ignored it a while before fixing it. Running the tests after each commit helps a lot.

We decided during the project to do testing sessions. During those all were present, one was clicking and everybody pointed out things. This allowed us to find many bugs from small to medium. Many issues were created from those sessions and then quickly fixed.
Many projects I start daily are not planned in detail and code is written quickly, which often leads to many problems due to design decisions. In this project I learned how good a project can work out if everything is already planned beforehand.

12.1.3.2 Roman Schweri

I learned to plan your software as good as possible and wait with the actual coding until other important things are very clear. It reduces the time you need to adapt in a later stage a lot. Also, unit and integration tests, combined with a good branching strategy and well-designed pipelines are a very powerful tool that I completely underestimated in the past. I also tended to code until perfection, before I deploy it to actual users to test. Having the testing sessions for each milestone helped a lot with the continuation with the project, since I could fix bugs, that potentially made upcoming features a lot harder & buggier, early on. I'll definitely start to make user tests more often and in an earlier stage.

12.1.3.3 Mauro Hefti

One takeaway for me was around our branching strategy. I actually liked that we could push straight into the dev branch, it helped us move forward quickly without much overhead. Still, I realized that in the long run it’s usually better to go through merge/pull requests and have at least one reviewer. That way you get more visibility into what everyone is working on, PR discussions can bring up interesting ideas, tests are guaranteed to run, and changes are challenged before they land in the main branch. Once something is pushed directly, it’s far less likely to be questioned.

Another lesson was about integration testing with Aspire. It’s a great tool, but I noticed it’s still a bit in its early stages, and some key features are missing, like being able to hook into the startup configuration or access the service provider. That’s why I think it was the right decision for us to switch back to WebApplicationFactory, which gave us more flexibility and reliability for our integration tests. I also worked with Aspire Standalone Monitoring and saw some limitations there. Important features are still missing, such as proper deployment behind a reverse proxy or persistence of monitoring data. Because of that, I’d say it’s not really suited for production use at this point and maybe that’s not even Microsoft’s intention right now. There are open issues about these topics, so the ecosystem is clearly still evolving.

13 Attachments

13.1 Timereports

Every working hour was logged by each team member. This was documented by adding them to a certain issue. We wrote a small tool to combine all these logged times to a report. They can be found as an attachment in 04_Project_Management\01_Timereports.

13.2 GitLab Issues

Each GitLab Issue was exported directly from GitLab to provide a summary that can be provided with this documentation. We still highly suggest to check the Issues directly in the corresponding repository. The exported Issues can be found in **“\02_Additional_Product_Documentation\04_GitLab_Issues_Export”**.

14 Declaration of Academic Integrity

Hiermit erklären wir, dass wir die vorliegende Masterarbeit im MAS Software Engineering mit dem Titel «VoxelAssembler - 3D Web configurator based on CAD-Data» selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht haben. Weiterhin erklären wir, dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise verwendet haben und in dieser Arbeit keine Adressen, Telefonnummern und andere persönliche Daten von Personen, die nicht zum Kernteam gehören, publizieren.

Ort, Datum Rapperswil-Jona, 14.09.2025

Name, Unterschrift: Tobias Wiesendanger [illegible]

Name, Unterschrift: Mauro Hefti [illegible]

Name, Unterschrift: Roman Schweri [illegible]