



eTryOn – Virtual try-ons of garments enabling novel human fashion interactions

Project Title:	eTryOn – Virtual try-ons of garments enabling novel human fashion interactions
Contract No:	951908 - eTryOn
Instrument:	Innovation Action
Thematic Priority:	H2020 ICT-55-2020
Start of project:	1 October 2020
Duration:	24 months

Deliverable No: D1.2

First working version of the mobile application and software for the self-scanning application

Due date of deliverable:	30 November 2021
Actual submission date:	9 December 2021
Version:	1.7 (final)
Main Authors:	Thomas De Wilde (QC), Axl François (QC), Yannick Francken (QC), Jorge Niño Castañeda (QC)

Project ref. number	951908
Project title	eTryOn – Virtual try-ons of garments enabling novel human fashion interactions
Deliverable number	1.2
Deliverable title	First working version of the mobile application and software for the self-scanning application
Deliverable version	1.7 (final)
Previous version(s)	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6
Contractual date of delivery	30 November 2021
Actual date of delivery	9 December 2021
Deliverable filename	eTryOn_D1.2_final.pdf
Nature of deliverable	Demonstrator
Dissemination level	PU
Number of pages	56
Work package	WP1
Task(s)	T1.1, T1.2, T1.3, T1.4
Partner responsible	QuantaCorp (QC)
Author(s)	Thomas De Wilde (QC), Yannick Francken (QC), Axl François (QC) and Jorge Niño Castañeda Jorge (QC)
Editor	Thomas De Wilde (QC)
Reviewer(s)	Elisavet Chatzilari (CERTH)
Abstract	In this deliverable, a basic working version used for iterative testing of the rigged, animatable 3D avatars and the self-scanning mobile application is delivered.
Keywords	body matching, avatar morphing, 3D modelling, human body models, serverless architecture, SDK

Copyright

© Copyright 2020 eTryOn Consortium consisting of:

1. ETHNIKO KENTRO EREVNAS KAI TECHNOLOGIKIS ANAPTYXIS (CERTH)
2. QUANTACORP (QC)
3. METAIL LIMITED (Metail)
4. MALLZEE LTD (MLZ)
5. ODLO INTERNATIONAL AG (ODLO)

This document may not be copied, reproduced, or modified in whole or in part for any purpose without the written permission from the eTryOn Consortium. In addition to such written permission to copy, reproduce, or modify this document in whole or part, an acknowledgement of the authors of the document and all applicable portions of the copyright notice must be clearly referenced.

All rights reserved.

Deliverable history

Version	Date	Reason	Revised by
1.0	31/05/2021	LaTeX template created	Thomas De Wilde (QC)
1.1	24/10/2021	Initial body matching chapter added	Jorge Niño Castañeda (QC)
1.2	26/10/2021	Initial architecture chapter added	Axl Francois (QC)
1.3	12/11/2021	Added morphing and prior information	Yannick Francken (QC)
1.4	22/11/2021	Added SDK sections	Thomas De Wilde (QC)
1.5	03/12/2021	Internal review	Thomas De Wilde (QC)
1.6	06/12/2021	Review	Elisavet Chatzilari (CERTH)
1.7	09/12/2021	Revision based on feedback	Thomas De Wilde (QC)

List of Abbreviations and Acronyms

Abbreviation	Meaning
AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
B2B	Business-to-Business
BMI	Body Mass Index
CLI	Command Line Interface
CPU	Central Processing Unit
DOF	Degrees Of Freedom
EC2	Elastic Compute Cloud
ECR	Elastic Container Registry
EFS	Elastic File System
FIFO	First In First Out
GPU	Graphics Processing Unit
ID	Identifier
LIFO	Last In First Out
OS	Operating System
PnP	Perspective-n-Point
QC	QuantaCorp
S3	Simple Storage Service
SDK	Software Development Kit
SNS	Simple Notification Service
SQS	Simple Queue Service
STAR	Sparse Trained Articulated Human Body Regressor
UI	User Interface
UX	User Experience
UUID	Universally Unique Identifier
vCPU	Virtual Central Processing Unit
VPC	Virtual Private Cloud
VPN	Virtual Private Network
VM	Virtual Machine

List of Figures

2.1	First row: initial STAR match. Second row: uncorrected 3D contours. Third row: corrected 3D contours. Fourth row: morphed output.	14
2.2	Left: initial STAR match. Middle: morphed results. Right: differences between initial STAR match and morphed results. Legend: top is no difference, bottom is maximum difference.	16
2.3	Ground truth comparison. Left: initial match on synthetically generated (previous model, not yet STAR). Middle: morphed output. Right: scan from full body 3D scanning booth.	16
2.4	Top: hands not aligned with the body in the side pose. Middle: hair not tied up and arm hiding the curvature of the back. Bottom: no tight clothing. Left: input image. Center: silhouette. Right: morphed output.	17
2.5	3D scan sample from the CAESAR dataset [1].	19
2.6	Matched avatar for model <i>H.</i> . The upper blue circumference shows the location of the chest. It can be noticed that we cannot capture the boundaries of the chest. The lower blue circumference is the topmost one that we can reliably measure in the upper torso.	20
2.7	Left: T-pose for custom body model based on SCAPE [2]. Right: T-pose for STAR model [3]	20
2.8	Changes in the initial body matching by increasing the number of shape parameters from 10 to 20 and adding shoulder displacement in the pose variables. Green is segmentation (projected silhouette of the real person), Red is the projection of the matched STAR model. White is the intersection of both.	21
2.9	Pinhole camera model, where rays from the object pass through the pinhole in the front of the camera and form an image on the back plane. Because this image is upside-down, it is more convenient to consider the virtual image that would have been created if the image plane was in front of the pinhole. Taken from [4].	22
2.10	Representation of an object being photographed using the pinhole camera model.	22
2.11	Illustration of the <i>euler</i> angles calculated from the mobile phone accelerometer readings.	24
2.12	Matched avatar for model <i>A.</i> . Left: current method, Right: Improvement by segmenting head and hair.	26
2.13	Matched avatar for model <i>A.</i> with hair/head segmented. Arm detail. Left: Original model mesh and corrected circumferences. Right: Resulting morphed avatar.	26

2.14	Matched morphed avatar for model <i>A</i> . with hair/head segmented. Left: 10 shape parameters. Right: 100 shape parameters. The fit to the image is better with 100 parameters, but we found experimentally that the accuracy in measurements is lower.	27
3.1	Current approach, using EC2	35
3.2	Hosting Python code inside the VPC using a VPN	35
3.3	Using spot instances and a spot instance broke to launch services	36
3.4	Lambda with EFS to provide dependencies	37
3.5	Lambda with ECR to provide base Docker images	37
3.6	A mix of all possible architectures.	38
3.7	Spot price evolution in the past 3 months for c5.large.	39
3.8	Spot price evolution in the past 3 months for c5.xlarge	39
3.9	Price calculation to run 1500 invocations of the function, assuming hot starts	40
3.10	Price calculation to run 300 invocations of the function, assuming cold starts	40
4.1	Screenshots of the scan view component. On the left is the UI of the front view, on the right the side view. The user needs to fit the scan subject inside the silhouette and press the capture button when ready.	50
4.2	Screenshot of SDK integration demo application. The form shows what the minimum required information is in order to take a scan.	51
4.3	Communication between SDK, APIs and Clouds.	53
4.4	In the middle, the scan UI for tablets, on the left a layout for smartphones, and on the right an example of an instructional module.	54

List of Tables

3.1	Costs for two types of C5 instances.	38
3.2	All supported callbacks.	42

Code Listings

3.1	The most basic Python consumer	30
3.2	Concrete example of a potential refinement consumer	32
4.1	QCScanController	45
4.2	QCApiSession	45
4.3	QCScanDelegate	46
4.4	QCApiToken	47
4.5	QCCreateBodyDTO	47
4.6	QCPicture	48
4.7	QCScanError	48

Contents

1	Executive Summary	12
2	Body Matching	13
2.1	Introduction	13
2.2	Silhouette-based 3D Model Morphing	13
2.2.1	Algorithm Overview	15
2.2.2	Mesh Deformation	15
2.2.3	Results	15
2.2.4	Challenges	17
2.3	Prior Information	18
2.4	Body Matching - Improving Chest Accuracy	18
2.4.1	Introduction	18
2.4.2	Occlusion of the Chest Area	19
2.4.3	Current Upper Torso Matching Performance	19
2.4.4	Solution Proposed	20
2.5	Body Matching - Impact of Camera Parameters	21
2.5.1	Introduction: Definition and Importance of Camera Parameters	21
2.5.2	Intrinsic Parameters	23
2.5.3	Extrinsic Parameters	23
2.5.4	Case for eTryOn: Estimation of Extrinsic Parameters	23
2.5.5	Impact of Estimated Parameters on Accuracy of Avatar and Body Measurements	25
2.5.6	Future Improvements for Estimating Camera Pose	25
3	Architecture	28
3.1	Introduction	28
3.2	Basic Concepts	29
3.3	Deployment	33
3.3.1	Costs	38
3.3.2	Lambda Cold Starts	40
3.4	Async Uploading	41
3.5	Integration with eTryOn	42
4	SDK	43
4.1	Introduction	43
4.2	Objective-C SDK	44
4.3	ReactJS SDK	51

4.4	Integration with eTryOn	52
4.5	UI/UX	53

Chapter 1

Executive Summary

In this deliverable, we describe the work put into the development of the first version of the Software Development Kit (SDK). This version will be used for end-to-end testing of the rigged, animatable 3D avatars and the integration with the eTryOn apps. Specifically, the integration of the SDK with the MagicMirror mobile application and the DressMeUp web application. We will have a look at the user interface (UI) of the scan component within the SDK and explain how the SDK, APIs and clouds communicate.

We explain that the SDK will make use of several micro-services running in an Amazon Web Services (AWS) cloud environment. These services are part of a new asynchronous scan architecture and take care of individual steps of the processing pipeline. This pipeline takes a front and side picture with accompanying metadata as input. The result of this body matching process is an as accurate as possible 3D representation of the scanned person. We will use the newly adopted STAR body model in order to help the 3D reconstruction problem of capturing humans to create avatars from the very limited data input. We will go over the basic concepts of the new architecture and provide you with some insights on deployment.

We also present our newly developed and improved techniques for finding the best matching STAR body parameters. First, we describe our new silhouette-based 3D model morphing technique to produce 3D models that better match with the extracted body measurements. The prior information needed for the body matching is described. We have also improved the human chest reconstruction and have performed an analysis of the sensitivity to deviations of the estimated camera parameters. We look at how we determine the camera position, and make suggestions on how to improve the camera pose estimate in the future.

Chapter 2

Body Matching

2.1 Introduction

The first working version of the SDK makes use of several micro-services running in a cloud environment. These services take care of processing the input pictures and metadata into an as accurate as possible 3D representation of the scanned person. QuantaCorp's efforts have always been focussed on extracting accurate measurements rather than delivering visually attractive 3D models. Therefore, we have further improved the body matching services to deliver more visually attractive 3D avatars with increased dress-ability so that they can meet the end user requirements for the eTryOn apps.

Within this chapter, we present the newly developed and improved techniques for the body matching. First, we describe our new silhouette-based 3D model morphing technique to produce 3D models that better match with the extracted body measurements. Next, we describe the prior information required for the body matching. The last two topics are the improvement of chest reconstruction and an analysis of the sensitivity to deviations of the estimated camera parameters.

2.2 Silhouette-based 3D Model Morphing

During the matching process, parameter values are found to represent the body as good as possible with the STAR model. We use only a limited parameter set to express a body. This is important to avoid ending up with a model that perfectly projects onto the silhouettes by adding high frequency mesh deformations to compensate low frequency errors. In other words, a globally inaccurate match should not be corrected by adding local, typically incorrect, deformations.

The drawback of this approach is that we are working with a less expressive body space. In order to overcome this shortcoming, we opted for adding an additional segmentation-based morphing of the matched STAR model.

Within the eTryOn project, the visual appearance of the 3D mesh is important. Hence, a refinement step as described is needed to deliver the required level of detail for accurately rendering bodies. In the past QuantaCorp's focus has always been on the extracted measurements rather than visual appearance. Therefore, having a proprietary method

to correct output measurements only, sufficed. In the context of eTryOn, a 3D shape morphing approach has been developed to be able to present 3D shapes that better correspond with the extracted measurements.

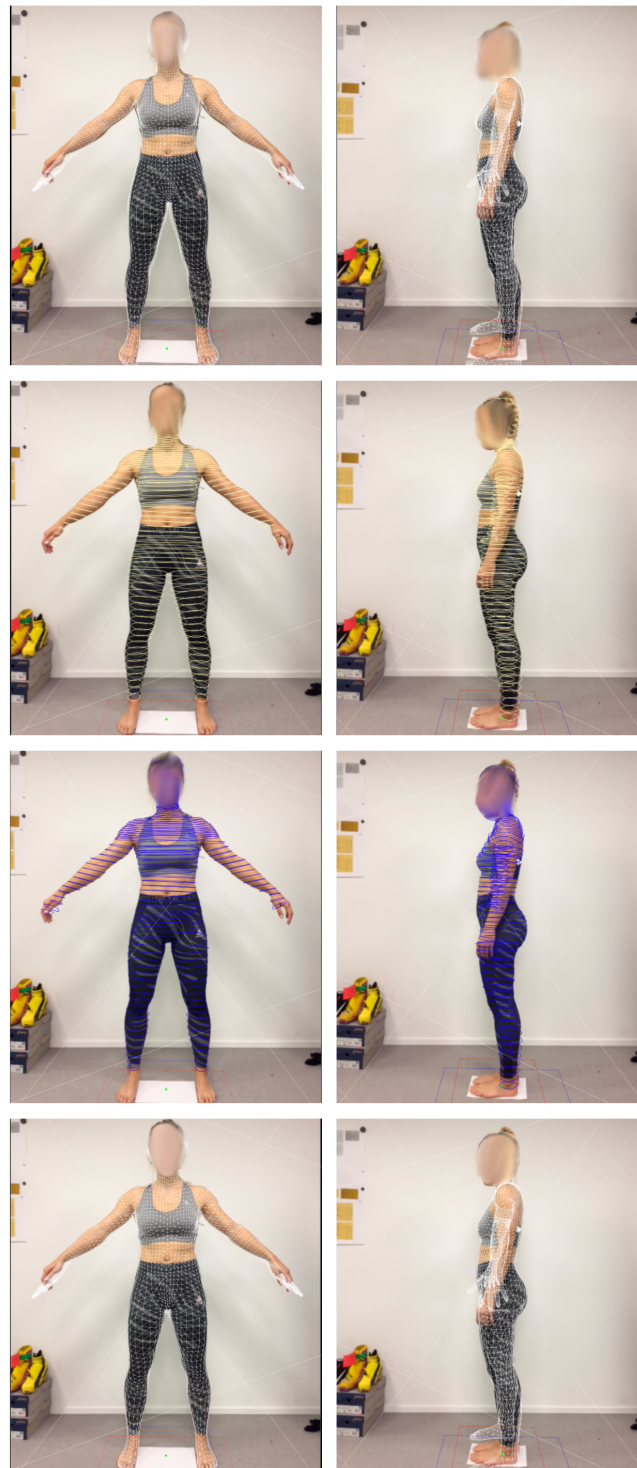


Figure 2.1: First row: initial STAR match. Second row: uncorrected 3D contours. Third row: corrected 3D contours. Fourth row: morphed output.

2.2.1 Algorithm Overview

The approach we take, first finds the most important set of shape parameter values that make the projections of the STAR model match with the front and side silhouette. Next, the resulting 3D shape is sliced into a fixed number of 3D contours. These 3D contours are then moved, expanded and shrunk based on the available front and side silhouettes.

The resulting corrected 3D contours are then merged together into a single 3D point cloud. This yields us a *target model*, to which we want to morph the initial STAR match. As a last step, the STAR match is morphed as close as possible to the target model, while keeping a certain smoothness into account. The morphing approach is further detailed in Section 2.2.2. Figure 2.1 shows the output of the individual steps.

2.2.2 Mesh Deformation

In order to deform the initial STAR match, or *source* model, to the *target* model, we take the following steps:

- Export the corrected 3D contours into a 3D single point cloud (see Figure 2.1, third row).
- For each 3D point (vertex) in the source model, search for the nearest vertex of the target model.
- Move each vertex to the position of that of the nearest vertex from the target model. Only apply this transformation to vertices that are allowed to move. Our source mesh topology is fixed, meaning that vertices can have different (x, y, z) values but their identifiers will remain fixed over different source meshes. Therefore, we can indicate per vertex identifier if it is allowed or not allowed to move. We achieve this by manually creating a boolean mask that indicates fixed areas, such as the head, hands and feet. Fortunately, the creation of such a mask is a one time operation and does not depend on the specific scan.
- The output of the previous step can result in a 3D model that has multiple vertices very close to each other or even coincide with each other. This is something that should be avoided since it is a bad practice and not supported by all software tools. We decided to resolve this issue by applying a smoothing filter that redistributes vertices evenly (approximate method). That way, vertices that ended up too close to each other are pushed away from each other. This operation makes sure that vertices remain close to a smoothed version of the morphed mesh.

2.2.3 Results

The impact of having an additional morphing step after the initial matching, is illustrated in Figure 2.2. This shows that the global pose and (low frequency) shape is a decent initializer, but that local corrections (high frequency details) are needed to add details.

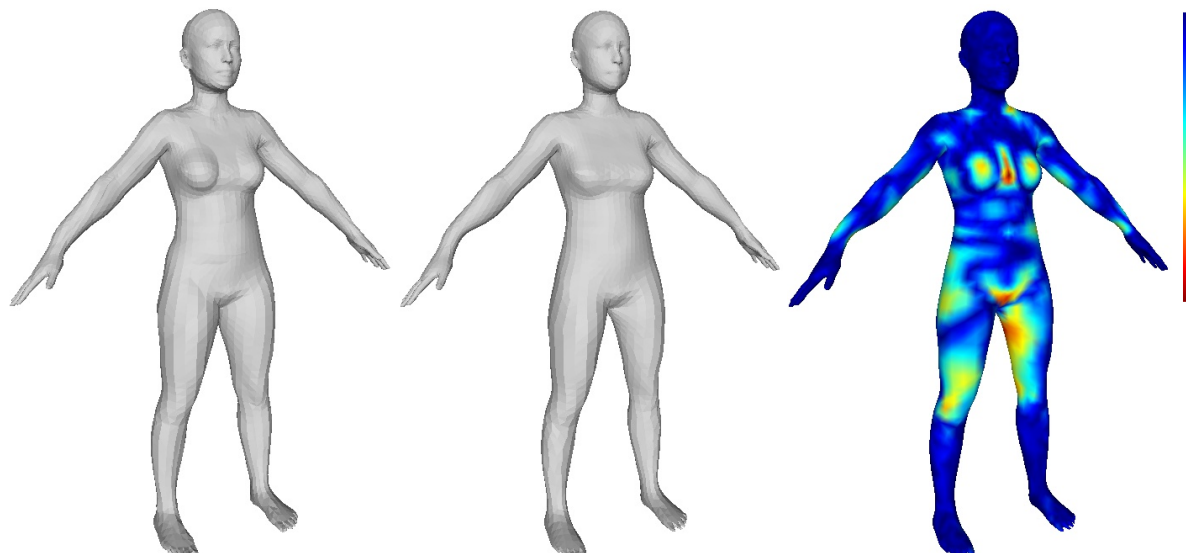


Figure 2.2: Left: initial STAR match. Middle: morphed results. Right: differences between initial STAR match and morphed results. Legend: top is no difference, bottom is maximum difference.

A visual ground truth comparison was performed by starting from 3D scans captured in a full body 3D scanning booth (ImFusion booth). Virtual silhouette projections were created to avoid any potential errors due to imperfections in the silhouette segmentation. The method was evaluated during development by visually inspecting results on an internal test set of approximately 20 different human bodies. An example result is shown in Figure 2.3. In this example, mainly the area around the abdomen has been improved by the morphing. Notice that the shoulder artefacts are not solved. The method needs to be further improved and besides the visual inspection, a quantitative evaluation should confirm the increased accuracy when the method is finished. This is still ongoing work.

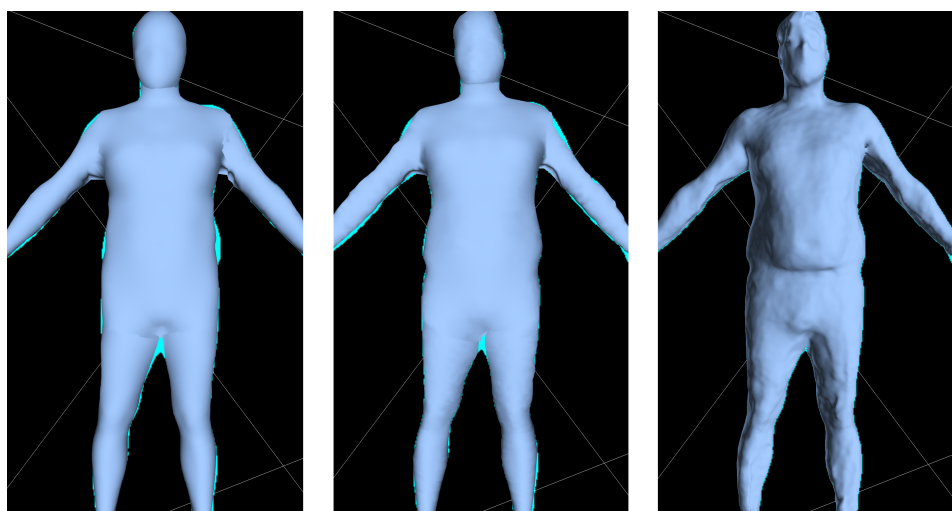


Figure 2.3: Ground truth comparison. Left: initial match on synthetically generated (previous model, not yet STAR). Middle: morphed output. Right: scan from full body 3D scanning booth.

2.2.4 Challenges

The presented technique cannot directly be applied to areas requiring a high level of detail. Therefore, the head, hands and feet need special attention. As described in Section 2.2.2, we solve this by distinguishing between fixed and morphable areas.

Even though this approach works well on the test samples, there is a high sensitivity to small errors in the initial pose and shape estimation. Using a binary mask might create sharp transitions between morphable and non-morphable regions. It is also important to notice that the morphed shape is strongly depending on the input silhouettes. In cases where the hair is not tied up, the arms and hands are not close to the body in the side pose or the person is not wearing tight clothing, the morphed model will be incorrect. Examples of such difficult cases are depicted in Figure 2.4.

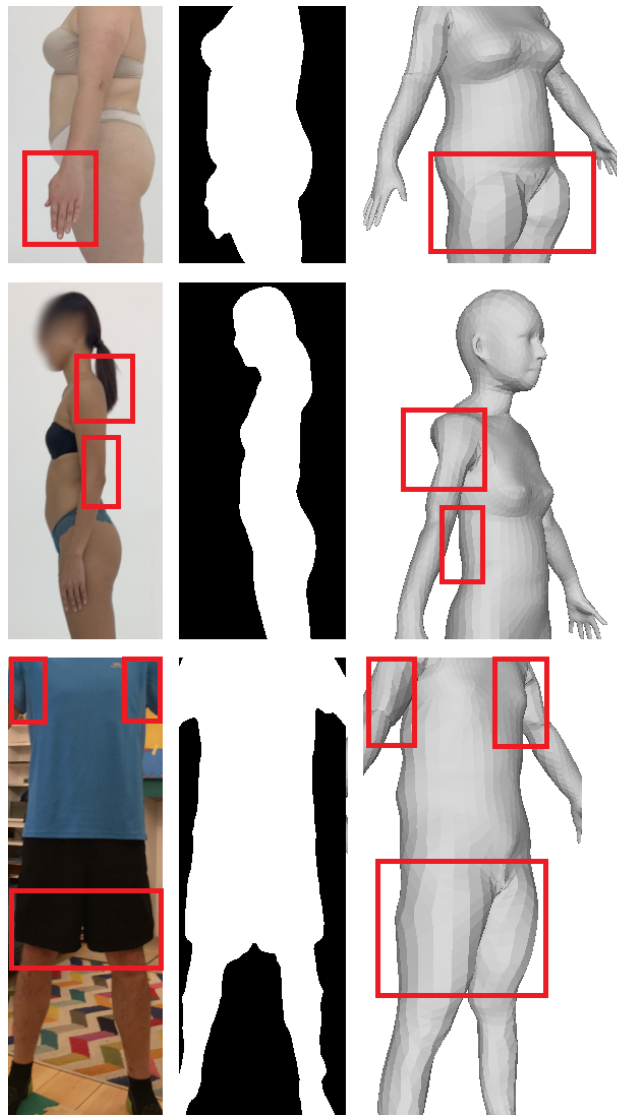


Figure 2.4: Top: hands not aligned with the body in the side pose. Middle: hair not tied up and arm hiding the curvature of the back. Bottom: no tight clothing. Left: input image. Center: silhouette. Right: morphed output.

2.3 Prior Information

In order to find the best matching 3D shape and pose of the person, we need, besides two input images, additional prior information. Scans are taken in a real environment and that needs to be modelled as a virtual environment to perform the body matching calculations. Therefore, we need to find the position and orientation of the camera with respect to the person, as well as the lens and sensor properties used in the image formation pipeline.

To estimate the position of the camera, we utilize the accelerometer values of the device. The lens and sensor properties are simplified to a single field-of-view parameter. Knowing the height of the person and combining this with the other parameters and the segmentation masks, we can estimate the 3D location of the camera.

Since our new STAR template model distinguishes between the genders male, female and neutral, this is also a parameter that could be provided to the body matcher software. It is optional though, and the matching switches back to neutral if no gender is provided.

We call all the above parameters the metadata and we send it in a JSON format. Together with three additional version strings, an example could be:

```
{
  "accelerometerX": 0.0037636982742697,
  "accelerometerY": -0.9993311762809753,
  "accelerometerZ": -0.036375608295202255,
  "fov_deg": 67.90680507369215,
  "gender": "F",
  "height": 1710,
  "hardware": "iPhone9,3",
  "versionIOS": "iOS/14.4",
  "versionQC": "Workwear/1.17.0.2"
}
```

2.4 Body Matching - Alternatives for Improving chest measurement accuracy

2.4.1 Introduction

We use a parametric body model in order to help the 3D reconstruction problem of capturing humans to create avatars. Until recently, we used an in-house body model based on the work of Anguelov et al [2]. Because of the way data was captured [1], where real human body instances were posed in the so-called "A" pose (see Figure 2.5), modelling the upper part of the torso was not ideal: the shape model in the chest area around the armpits ends up with artefacts. This was one of the reasons why we explored an alternative human body model and have chosen STAR [3], which addresses the armpits modelling better. More details on the STAR model can be found in Deliverable D1.1 section 5 Integration of New Body Space.

However, matching body circumferences in the chest area is still more challenging than

other torso parts, namely the waist and hip. In the following we explain the issues found and propose a potential solution to address them.



Figure 2.5: 3D scan sample from the CAESAR dataset [1].

2.4.2 Occlusion of the Chest Area

In order to obtain a reliable measurement of the chest circumference, the chest area needs to be captured. Currently for our scanning solution based on two pictures (1 camera, 2 shots, 2 poses, see Section 2.5.4), we use the A-pose with arms elevated around 45 degrees for the front view, and the I-pose with arms resting for the side view. In many cases, depending on the shape anatomy, this results on the chest area being occluded in the front view, as illustrated in Figure 2.6.

2.4.3 Current Upper Torso Matching Performance

The torso measurements are normally divided in three regions for many applications like garment size recommendation [5][6]. These correspond barely to the chest/bust/under bust in the upper torso, waist/pant waist in the middle and hip/crotch in the lower part of the torso. Our measurement accuracy for the upper torso is slightly lower than in the other regions.

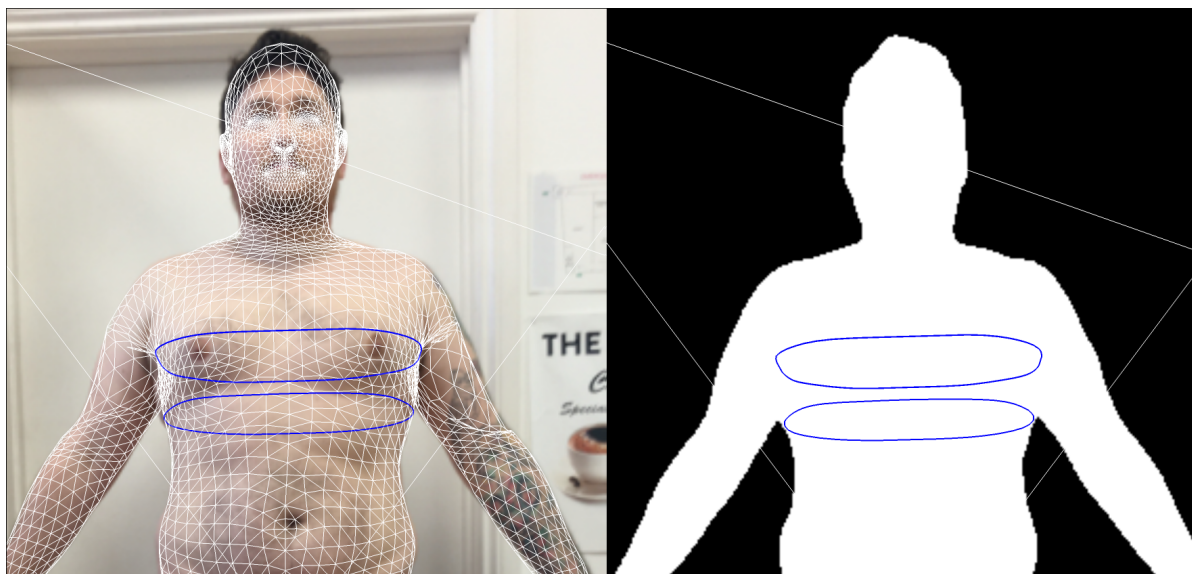


Figure 2.6: Matched avatar for model $H..$ The upper blue circumference shows the location of the chest. It can be noticed that we cannot capture the boundaries of the chest. The lower blue circumference is the topmost one that we can reliably measure in the upper torso.

2.4.4 Solution Proposed

With the aforementioned issues in mind, we investigated how to improve body matching in the upper torso. In order to tackle them, and since the STAR model offers a better modelling of the armpits area (see Figure 2.7), we decided to modify the pose in the front view by having the arms extended at around 90 degrees with respect to the torso axis (we will refer to this as the T pose). In this way, we managed to reach the armpits area in the front view.

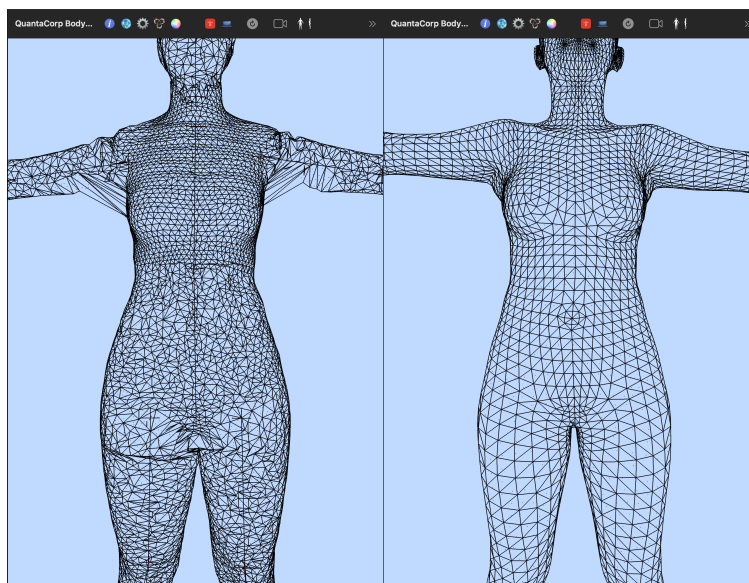


Figure 2.7: Left: T-pose for custom body model based on SCAPE [2]. Right: T-pose for STAR model [3]

Matching in T pose needed some changes to deal with misaligned shoulders. We found experimentally that by increasing the number of shape parameters from 10 to 20 and adding shoulder displacement in the pose variables, a higher accuracy in upper torso circumferences can be achieved. This was tested on an internal data set. The changes in matching performance (the initial body match before morphing) are shown in Figure 2.8.

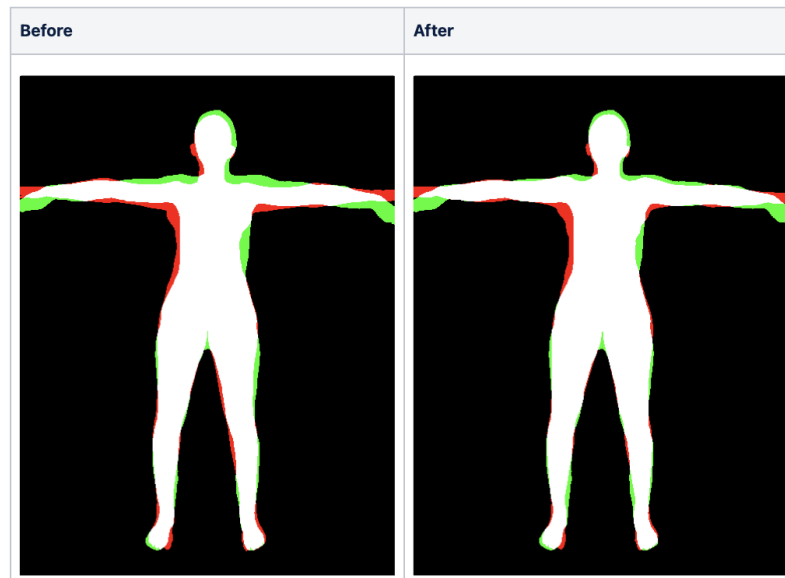


Figure 2.8: Changes in the initial body matching by increasing the number of shape parameters from 10 to 20 and adding shoulder displacement in the pose variables. Green is segmentation (projected silhouette of the real person), Red is the projection of the matched STAR model. White is the intersection of both.

Note: The solution proposed is in exploratory stage and is not currently available in the eTryOn scanning framework.

2.5 Body Matching - Impact of Camera Parameters

2.5.1 Introduction: Definition and Importance of Camera Parameters

Trying to obtain the 3D coordinates of a person from two colour images, one for the front view and one from a side view, is an *ill-posed* problem [7]: we are trying to solve 3D reconstruction with incomplete information, where the main limitation is the lack of depth (how far each image pixel is from the image). Our main purpose at solving this problem is to create realistic avatars using a consumer device (e.g. a smartphone) as body scanner. In order to produce avatars which are accurate to the body measurements of the scanned person, we need to compensate for the lack of depth information. We do this by relying on a 3D human body model, whose parameters are adjusted to fit colour images, which contain 2D projections of a person. Assuming a perfect 3D model, the realism of a resulting fitted avatar will depend on the accuracy of what we measure with

the camera. This accuracy is defined by how well we can estimate the two set of camera parameters: the intrinsic and extrinsic parameters [4]. To better explain this, we will use the pinhole camera model which is shown in Figure 2.9.

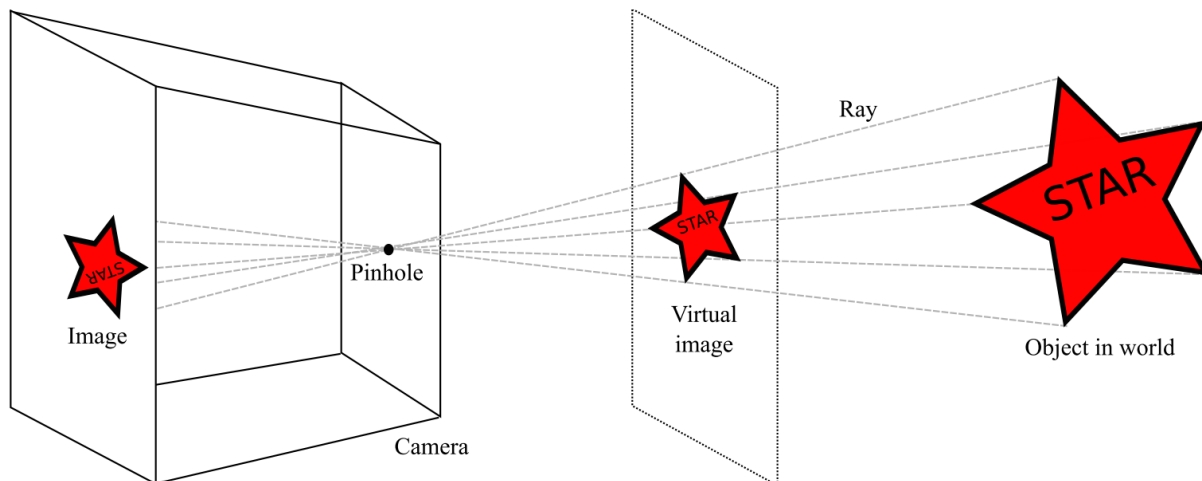


Figure 2.9: Pinhole camera model, where rays from the object pass through the pinhole in the front of the camera and form an image on the back plane. Because this image is upside-down, it is more convenient to consider the virtual image that would have been created if the image plane was in front of the pinhole. Taken from [4].

Figure 2.10 shows a representation of an object being photographed. The intrinsic parameters refer to the description of the camera itself. In Figure 2.10 they are represented by the focal distance and the principal point location in the 2D image coordinate system (x, y) . The extrinsic parameters refer to the location and orientation of the camera coordinate system (u, v, w) with respect to the real world coordinate system (x_w, y_w, z_w) where the object to be scanned is located (represented by the star in the picture).

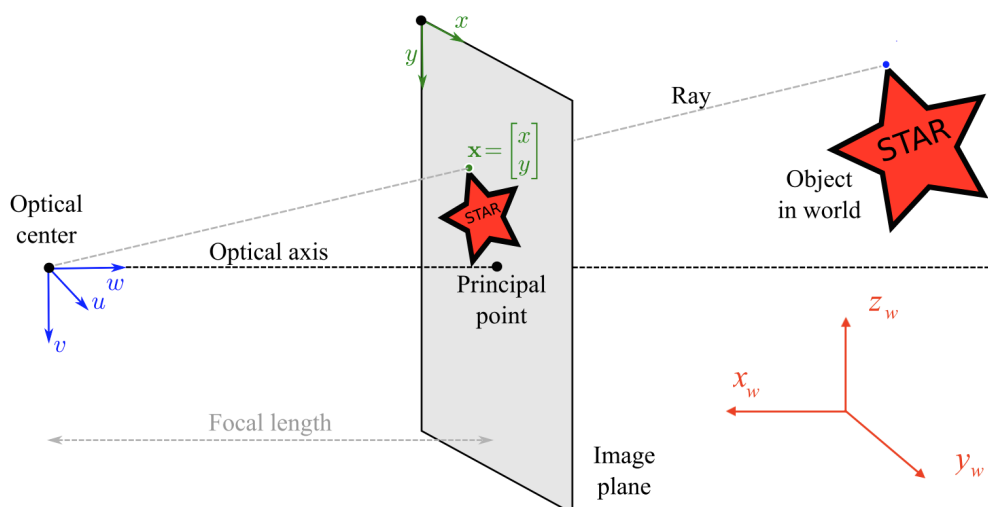


Figure 2.10: Representation of an object being photographed using the pinhole camera model.

2.5.2 Intrinsic Parameters

The final model is represented by the camera matrix K which encodes the intrinsic parameters:

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where f is the focal length and (c_x, c_y) is the principal point location. These values are expressed in image pixels. For practical purposes, this matrix can be obtained for a specific camera by a process known as camera calibration [8]. This process involves the use of external geometric patterns and taking many pictures at several camera orientations. In some cases it is necessary to model the distortion of camera lens [9]. When camera calibration is not possible, for modern smartphones the intrinsic camera parameters can be obtained via the operative system API [10][11].

2.5.3 Extrinsic Parameters

The extrinsic parameters, which encode the camera location and orientation (*a.k.a.* camera pose), are represented by the *3-element* location vector t_{vec} and the *3-element* rotation vector r_{vec} . This representation reflects the *6-DOF* of the camera in the real world. Estimating the extrinsic parameters is always a *sine qua non* condition in order to obtain real-size and human-like avatars from pictures. This can be done through calibration on site (a process called camera pose estimation) using planar patterns located on the ground, patterns printed on special clothes worn by the subject that is being scanned or using polyhedral arrangements of patterns beforehand. These methods try to solve the so-called Perspective-n-point (*PnP*) Problem [12], which consists on estimating the camera pose estimation from projective observations of known model points. In less controlled scenarios, for example scanning at home during the eTryOn project, it is difficult to fulfil the conditions to solve this problem. In order to overcome this difficulty, we proposed a simple method to estimate camera pose using the attributes of the person that is being scanned and the accelerometers of the smartphone.

2.5.4 Case for eTryOn: Estimation of Extrinsic Parameters

For the case of eTryOn, we have a scenario where the camera and inertial measurement unit of a mobile phone are used to estimate the extrinsic parameters. Two pictures of the user are taken: one in front A-pose and another one in side I-pose.

Camera Orientation

In order to estimate the camera orientation, we rely on the readings of the device's accelerometers. From these data we estimate the *euler* angles and compute the equivalent rotation vector r_{vec} .

Being \mathbf{a} the vector with the accelerometers readings:

$$\mathbf{a} = [a_u \quad a_v \quad a_w]^\top$$

we obtain the *euler* angles *pitch* and *roll* as follows:

$$pitch = \frac{\arctan(a_w)}{\sqrt{a_u^2 + a_v^2}}$$

$$roll = \arctan\left(-\frac{a_u}{a_v}\right)$$

Because the *euler yaw* angle cannot be measured and we expect the person to stand parallel to the camera in *A* pose and perpendicular in *I* pose, we assumed its value to be zero. For more clarity, see Figure 2.11.

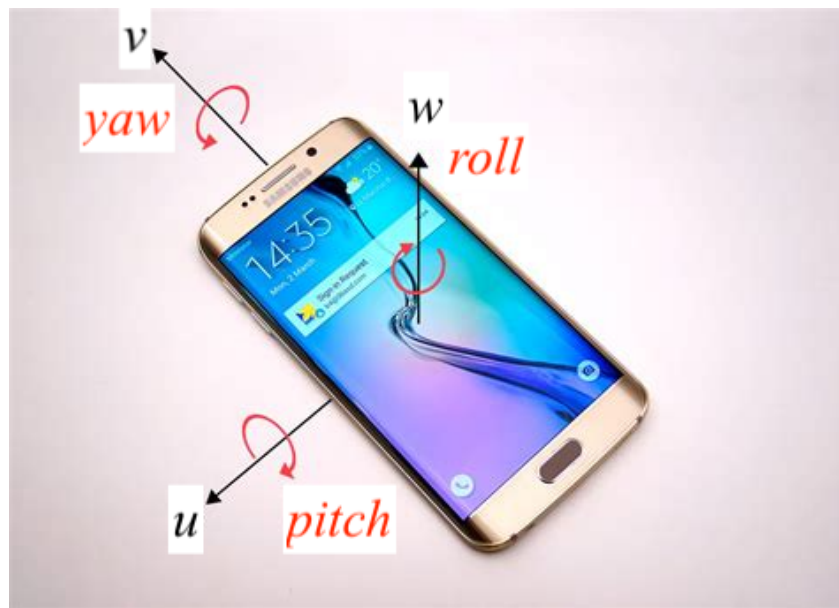


Figure 2.11: Illustration of the *euler* angles calculated from the mobile phone accelerometer readings.

Being $\mathbf{R}_{axis}(\alpha)$ a matrix that defines a rotation around a particular *axis* of angle α , the general rotation matrix that define the camera orientation in the world coordinate system is expressed as:

$$\mathbf{R}_{cam} = \mathbf{R}_{y_w}(pitch) \cdot \mathbf{R}_{x_w}(roll) \cdot \mathbf{R}_{x_w}(-90) \cdot \mathbf{R}_{y_w}(-90)$$

The r_{vec} vector is then expressed as:

$$r_{vec} = Rodrigues(\mathbf{R}_{cam}^T)$$

where $Rodrigues(.)$ is the *Rodrigues' Rotation Formula* [13].

Camera Position

Once we have estimated the camera orientation, we proceed to estimate the camera position. This is done iteratively during the silhouette-based matching process (see Deliverable D1.1 section 5.3 Silhouette-based matching). When knowing the camera pose beforehand, the body shape and pose parameters are optimised to fit the segmentation silhouette to the rendered shape model projection. Since the camera position is not yet known, we include it as an additional input variable to the optimisation. With \mathbf{c} being the camera position in world coordinates, the t_{vec} vector is then expressed as:

$$t_{vec} = -\mathbf{R}_{cam}^T \cdot \mathbf{c}$$

2.5.5 Impact of Estimated Parameters on Accuracy of Avatar and Body Measurements

In order to measure the impact of not having a calibrated camera pose and estimating the extrinsic parameters as explained in Section 2.5.3, we run body matching simulations on a set of 3D meshes. Thus, instead of real pictures we use projections of 3D meshes collected from real people using an in-house 3D scanner. By doing so, we isolate the problem of camera pose estimation and exclude the error added from image acquisition and processing. We therefore compare the result of two sets of body matching simulations: one where the camera pose is known in advance (the values used to obtain the rendered mesh projections), and another one where the camera pose is estimated as explained in Section 2.5.3.

2.5.6 Future Improvements for Estimating Camera Pose

The method described does not account for cases where people have abundant hair. We try to fit the top and bottom of the model projection and the person segmentation, where the body model head does not have hair. We are currently working towards segmenting individual parts of the body in the image. We have already reported in Deliverable D1.1 the need for segmenting the arms in side view (see Deliverable D1.1 section 6.3 Upper Limb Measurement Refinement). For improving the estimation of camera pose, we need to apply hair/face segmentation. In Figure 2.12 we report what the improvements would be by adding this feature.

Notice that although there is a noticeable improvement in the output avatar (e.g. arms size), the result is not yet visually *perfect*. This is due to the misalignment of arms in the original matching result, as shown in Figure 2.13.



Figure 2.12: Matched avatar for model A.. Left: current method, Right: Improvement by segmenting head and hair.

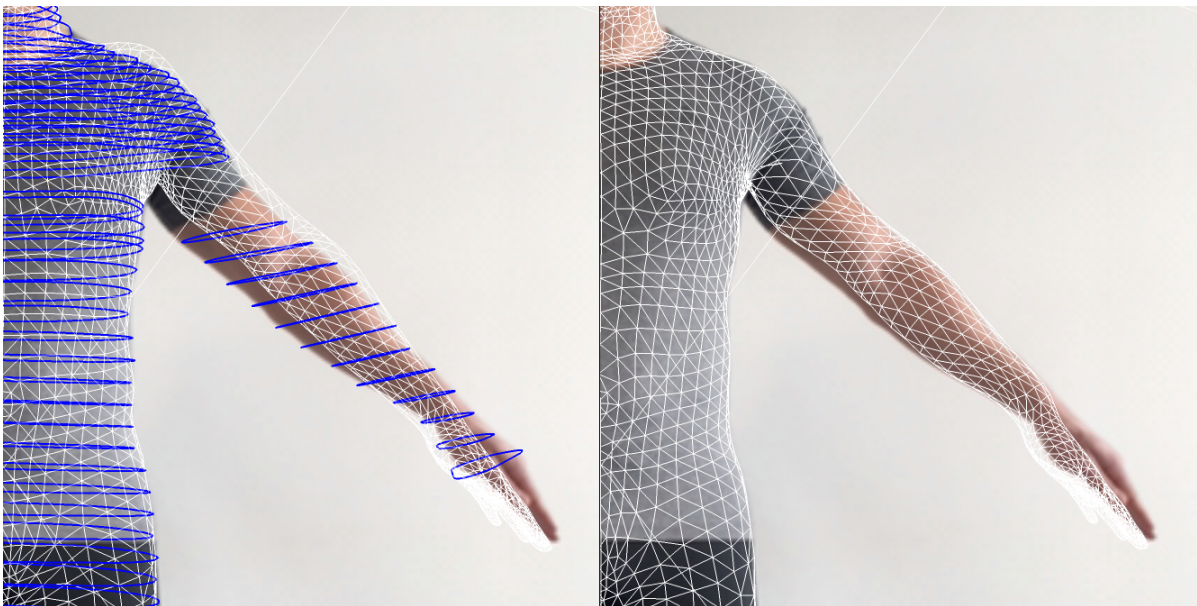


Figure 2.13: Matched avatar for model A. with hair/head segmented. Arm detail. Left: Original model mesh and corrected circumferences. Right: Resulting morphed avatar.

Currently we limit our shape model parameters to the first ten principal components [3]. We have found experimentally that by using this value, the shape of body circumferences is closer to the ground truth. If we are to increase the number of shape parameters, we will obtain a better arms alignment in the initial body match and almost a perfect fitting of the morphed avatar to the taken picture (as seen in Figure 2.14) at the cost of lower measurement accuracy in the circumferences across the torso, which translates in a *less realistic* avatar.



Figure 2.14: Matched morphed avatar for model *A*. with hair/head segmented. Left: 10 shape parameters. Right: 100 shape parameters. The fit to the image is better with 100 parameters, but we found experimentally that the accuracy in measurements is lower.

In order to keep both measurement accuracy and avatar fitting, we are currently investigating how to address the arms misalignment (which happens during the initial body match) in the later avatar morphing stage.

Chapter 3

Architecture

3.1 Introduction

The QuantaCorp cloud has a very monolithic architecture. Over the years, services have been tied in to the body matching algorithm without a lot of regard for the architecture. This has negative effects such as problems with scaling, performance, accessibility and reporting. The complete opposite of a monolithic architecture is an architecture where only microservices are used. These microservices each get ownership of a small part of the application. The microservices may or may not communicate with each other. This communication can take many forms and may include objects, identifications of objects, subresults, and many more. Basically there is no limit to the input of a microservice, the only difference with a monolithic application is the scope of the service.

In this chapter we will explain how we use microservices to modernize and better the QuantaCorp cloud. While still keeping some of its advantages, like an always accessible API. From a deployment perspective the main driver is cost. Some servers are inexpensive to host, while other more special servers cost upwards of hundreds of euros each month. When implementing the new microservices we wanted to minimize service cost versus the usage that is happening. It makes no sense to have a server running 24 hours when it's only used for 9 hours each day. The challenge is that you might get a request when you least expect it. So the new architecture also has to be able to deal with that. We identify this as the second driver, namely availability.

In the interest of protecting the QuantaCorp IP, we will not discuss the whole architecture in this chapter. We will focus on a generic implementation of a QC microservice. A specific example regarding the segmentation will also be given and the impact it has on costs. Moreover in the generic implementation, all facets of the microservice will be detailed. At QuantaCorp we work with Amazon AWS as cloud provider and thus all specific examples will use the AWS services.

When taking both drivers explained above into consideration, the jump to “pay for what you use” is quickly made. Currently there are two means of achieving that within the AWS cloud, Lambdas or spot instances. Both offer different ways of scaling and both can be used for our new microservice architecture. We will discuss both methods with their respective advantages and disadvantages later in this chapter. It is important to note that other cloud providers might offer different services, with which the architecture

can also be built. There is no widely used general specification that describes a generic cloud architecture. For instance, all cloud providers have the concept of a blob storage, key-value store, VMs and much more. Even though the concepts are the same, the interfaces are not. In order to give businesses more power and become less dependent on cloud providers, it would be great if there was a cloud provider independent architecture language. Note that sometimes small nuances exist between different services even if they serve the same purpose.

3.2 Basic Concepts

When designing an architecture like this, it is best to come up with a set of base rules. For instance when a scan is made, a unique identifier is assigned to that scan. This identifier is a UUID. Every service will be able to access that particular scan through its UUID. All the resources related to that scan will either be prefixed or suffixed with that UUID. Let's say we want to store the segmentations of a scan (front and side) on a blob-storage like S3. The naming convention then looks like `UUID_front_mask.png` and `UUID_side_mask.png`. For a service there is never any doubt where to find the images. These conventions can even be injected via configuration files. A piece of software's most basic principle is processing. To do any sort of processing we first need input and for that input the processing will result in certain output. These high-level concepts can be translated to the new architecture. As can be imagined, not every service will require the same input and not every service will give the same output. But a service needs a starting point which we define as the UUID. The service may use the UUID to talk to other services and acquire whatever data it needs to do its processing.

In a second step we need a scalable way to trigger the processing, in most architecture the concept of a queue is used. Whether that be a FIFO or LIFO queue doesn't really matter. The idea is to keep a record somewhere of UUID's to be processed. We only guarantee that we will try every UUID on the queue at least once. You will often find the concept of consumers when reading into queues. A consumer is an entity that will read a certain number of messages from the queue. The number of messages can be one or more. The consumer will in this case be responsible for triggering the processing for the UUID. The idea would then be to scale the number of consumers to whatever scale is needed by the application.

How can a consumer do the processing you might ask? A consumer is simply a high level concept. Any Python script that actively polls and retrieves messages from the queue can be a consumer. So let's assume we make a simple sum function that requires two integers. Let's add hundreds of messages with two random integers. A consumer would read one message and make the sum of the numbers, these numbers could be separated with a space for instance. Afterwards the consumer could choose to upload the results to a different queue. As a final step the consumer removes the message from the queue to indicate he has successfully processed the message. If the processing fails for whatever reason, you can react in two ways to it. Either you let another consumer try to do the processing or you acknowledge the error and remove the message anyway. So let's give each message on the queue a unique id next to the message which would be two integers. How can we prevent two consumer from reading the same message and thus doing double "processing"? To solve this problem the concept of *visible timeout* is

added to the queue-system. When a consumer reads a message that message becomes invisible for other consumers for a certain period of time. This period is called the *visible timeout*.

In short we have a queue, on this queue are uniquely identifiable messages that contain a body which is needed for to do a processing step. Consumers are added, which read messages from the queue. If successful, delete the message from the queue. A process that is unable to do its processing might be retried by another consumer. It is important to know whether a success scenario for the processing can still be reached. Otherwise, we need to terminate that specific message. We do this by deleting it from the queue so no other consumers will read it.

Where do these consumers exist or run? The easiest consumer would be an infinite loop which polls the queue and takes necessary actions based on the message. This is very similar to a daemon process that constantly lives and reacts to input it gets. Only here the input is very deterministic, namely messages on a queue. This infinite loop requires a computer to run on, again there are multiple scenarios; local hardware (developer PCs), cloud hardware (EC2 instance, Lambda function). For the sake of simplicity, assume the easiest version, local developer hardware. If we make this type of consumer in a Python script for instance, we could easily hook all of the developer hardware up to the cloud and use them as consumer for a microservice. Our server costs would be low, but it will be difficult to guarantee any stability and limits the applicability of the machine for other tasks. Think about the impact of a reboot, power outage, heavy load on the machine, running insecure software and other risks.

A very basic example (See Code Listing 3.1) of a consumer that polls and deletes a message from a queue in Python:

```
while(True):
    # Check sqs for new message, if found pull it!
    ext_id, message = sqs.readMessageFromQueue()

    if(ext_id is not None):
        print("Received message with extId {}".format(ext_id))

        # do any sort of processing

    message.delete()
```

Code Listing 3.1: The most basic Python consumer

What do we do when a unit of work fails? How do we report errors or track the status of our overall job between all Lambdas? In general this is a problem that occurs both for Lambdas as for on-demand instances. Our services need a way to track everything that's being done for a certain unit of work. Let's talk about two steps from our previous deliverable, segmentation and its refinement (see Deliverable D1.1 section 6 Measurement Accuracy and section 7 Silhouette Segmentation). Both can be seen as individual units of work, which we would like to scale using a new architecture. But a segmentation can fail for numerous reasons, for instance the service was unable to retrieve the correct image format, the resolution was not correct or the channels were not in the correct order (RGB vs BGR). In some of these cases we will want the input to change, in other cases

the services simply fail. In all cases it should be transparent to the user (in this case the programmer) of the service what the problem is. And what potential actions can be taken by that user to remedy these problems, like for instance end-user interaction.

Before ever reaching our back-end services, most end-users will query our Public API. This API is a centralized 24/7 running system that could be used to track all progress. An other option would be to use DynamoDB to store any such progress. In either case we are faced with additional complexity. How do we get the updated status to the end-users? Can we actively push them messages or should they actively poll our services for news? Both approaches are valid and in the interest of time we have chosen to let our clients actively poll our Public API. Our Public API stores several possible statuses for each service. Some of these statuses will require end-user interaction, others won't. The consumers are able to actively push updates to our Public API using SNS, which is a publish-subscribe message system. Where the API is subscribed to all topics related to these services and the services all publish to these topics.

To continue, we will give an example (See Code Listing 3.2) of a more concrete implementation of our segmentation refinement consumer. Note that this code does not reflect the code currently used in our environments, this is early stage code and has been modified so that the reader can easily grasp the concepts of the consumer that has been described in the above paragraphs. Note that DynamoDB in this example is used to track the metadata of the processing. Many elements, such as all different fail status codes, are not included in this example and the further specification of what exactly each of these methods does is not given in this document.

```

# implicitly init model
server = RefinementServer(logging, gpuServer=True)

while(True):

    # Check sqs for new message, if found pull it!
    extId, message = server.sqs.readMessageFromQueue()

    if(extId is not None):

        try:
            # get all required data
            image = server.s3.downloadImage(extId)
            image4k = server.s3.download4kImage(extId)
            mask = server.s3.downloadMask(extId)

            # do prediction of mask
            resultMask = server.predict(image, image4k, mask, extId)

            # upload mask to s3 and publish completion
            server.s3.uploadImage(refinedMask, extId)
            server.sns.publishCompletion(extId)

            # calculate the time this service needed
            endTime = (time.time() - startTime)

            # update dynamo
            server.dynamo.updateRefinementStatus("SUCCESS", extId, endTime)

            # remove extId from queue
            message.delete()

            # wait 1 second, before processing the next message
            time.sleep(1)

        except Exception as error:
            # handle a error
            endTime = (time.time() - startTime)
            server.sns.publishFailed(extId, error)
            server.dynamo.updateRefinementStatus("FAIL", extId, endTime,
                                                error)

            # in this example, we always delete the message and won't retry
            # processing
            message.delete()
            continue

```

Code Listing 3.2: Concrete example of a potential refinement consumer

3.3 Deployment

Now that we understand the basics, we will discuss the deployment of these resources. In this chapter the three main infrastructures to deploy on will be discussed: EC2 on-demand, EC2 spot-instance and AWS Lambda. First a company always has to make the choice if they want to run something on-premise (in short on-prem) or in the cloud. We opted for the cloud approach since this is more scalable with a smaller budget. Whether the machine is on-prem or not, the most traditional server everyone recalls is one that runs 24/7 and makes a lot of noise. In the cloud the noise problems disappear but the cost is shifted to a per time unit billing. The granularity of those time units depend on the cloud provider. While on-prem has a steep upfront cost and a maintenance cost. As long as nothing breaks, you only have to worry about the upfront cost and power consumption. Aside from the cost it is important to understand that in order to scale such infrastructure, physically more server have to be added. While in the cloud this is the task of the cloud provider.











The current QuantaCorp architecture has been using EC2s that are running 24/7, not taking into account the variable load or the difficulty to scale these virtual machines. Scaling vertically isn't that hard when running already configured machines, but scaling horizontally is not that easy. In order for us to do that right now, we would have to spin up a machine and run an Ansible script that automatically configures the server. These scripts are not particularly fast and only work pre-emptively. Moreover version after version of Ansible has had breaking changes and the commands in the scripts also differ based on the OS and its particular version. These scripts thus require a lot of maintenance and have a hard time to recover from unknown errors. For instance permission errors, changes in the update policy of a dependency and so on. In the ideal scenario scaling should be as simple as flicking two buttons, one that scales horizontally and one that scales vertically.

The alternatives for running EC2 instances 24/7 are twofold. The first alternative does not really change the concept of EC2 but it does change the price. When Amazon has spare capacity on a datacenter they offer this capacity for customers to use as spot instances at a lower price. These spot instances have a bottom price that is set by Amazon (normally around 60% cheaper) and a top price which is the on-demand price. You can only use them for a set time. Once the bottom price is set, customers are allowed to bid for capacity. The highest bid wins and gains the capacity for a set time or until it is needed by on-demand customers. This is a simple and effective way of decreasing server costs by automating the whole bidding process. AWS CLI and the AWS API implementation in Python (boto3) both offer APIs to automate the whole bidding process.

We tried using spot instances during segmentation experiments, but at this point we have not yet solved the problem of the initial configuration. For this we can use the concept of base images. Instead of using a stock Ubuntu image we can first modify this image using Ansible and then use that as base image. We then only have to apply the additional changes needed for that service to the image. This concept thus involves automating the bidding process. Making a general Ansible script and a specific Ansible script per service that takes care of the deployment details. These base images are called AMI or Amazon Machine Image within the AWS cloud.

The second alternative is server-less computing. Server-based compute power always requires a lot of configuration (user, network, iptables, etc). In order to avoid this we abstract these rules and focus on simply executing a program. These programs are more generally called functions and in the case of AWS, Lambdas. These Lambdas are configurable to run within a VPC, have firewall rules and have a set amount of available memory. We are only billed for the total amount of time the Lambda function runs. The main advantage is that Lambda functions are considerably more scalable than normal EC2 instances. We can run multiple Lambda functions at the same time while we only have to configure one. All of these functions will act as consumers to the queue for which they are configured. The only thing we have to configure is the number of functions we want to run concurrently, which can be done through either CLI or AWS Console (website) and is near instant.

Below we provide a quick review of the building blocks that we explained up till now:

Name of Service	Goal	Icon
SQS	LIFO or FIFO queue for uniquely identifiable messages	
Lambda	A serverless compute instance (ran via docker, EFS or ZIP package)	
S3	A blob-storage (Each blob has a unique identifier)	
DynamoDB	key-value NoSQL database	
SNS	pub sub notification service	
EC2 Spot instance	An instance where price is determined by a bidding process	
EFS (Elastic File Storage)	Scalable mountable volume	
VPC (Virtual Private Cloud)	Isolated cloud network	
API (Application Programming Interface)	In our case a Java REST webservice	
ECR (Elastic Container Registry)	Registry with different container images	

All of these services have their advantages and caveats. In the bigger picture of this chapter, the reader should understand that all concepts can be replaced by any other equivalent technology outside of the AWS platform. Using external services outside of an AWS VPC, requires configuration to happen so that the integrity of the service is guaranteed, such as network and firewall configuration. Now lets go over the diagrams of the suggested architecture and their up and downsides. These diagrams have been simplified to focus on the segmentation and thus do not reflect the final architecture that eTryOn will use. However, everything we do use, is based on the principles put forward in this document.

- EC2 (Figure 3.1)

The traditional method of hosting a service, using a 24/7 available virtual machine. Note that this also requires a setup using Ansible as described above. Even though it is not explicitly shown in the figure, multiple services can theoretically run on the same machine if the resources allow it.

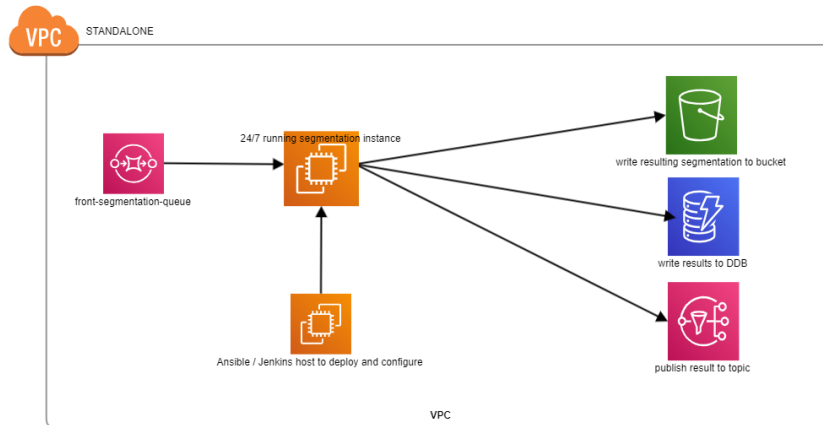


Figure 3.1: Current approach, using EC2

- Self-hosting (Figure 3.2)

In the case of self-hosting we need our host machines to be available within the VPC. This is done using a VPN connection to that specific VPC. These machine may run anywhere as long as a successful VPN connection can be established. All the AWS services have to be accessible from within the VPC.

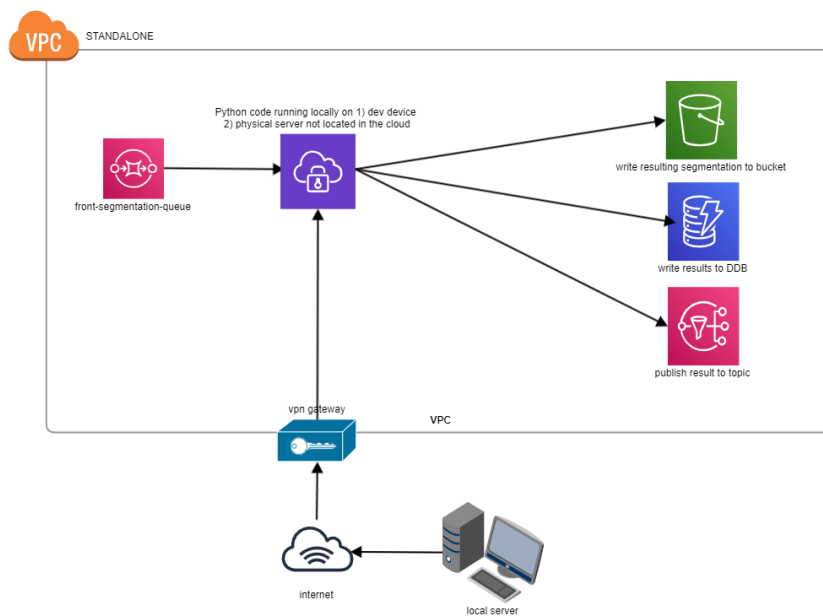


Figure 3.2: Hosting Python code inside the VPC using a VPN

- Spot instances (Figure 3.3)

In our current implementation of spot instances, the bidding is initiated using our Public API, which spins up a Lambda function that uses the available amazon APIs to launch a spot instance. These spot instances are particularly interesting if there is high computational demand at a certain time. A practical example is a scanning session taking place. In terms of machine learning, inference on-demand on GPU is what we are looking for. By using a spot instance, we are able to reserve a GPU-instance, run all our inference for one scan session and then spin down the instance. The parameters for the bidding code are the instance types we desire and the maximum price we want to pay. It is possible the bidding algorithm is unable to find a satisfactory solution for us. In which case the instance cannot be reserved. This is why we treat this method as a tool for optional work. Work that would enhance the result but not define it, for instance segmentation refinement.

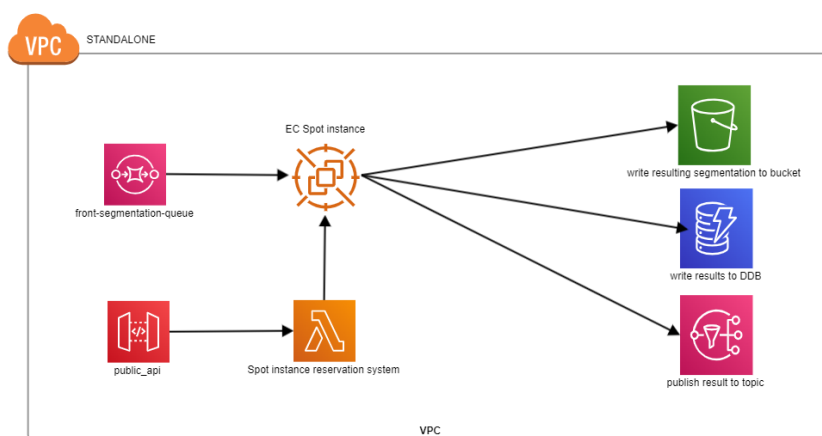


Figure 3.3: Using spot instances and a spot instance broke to launch services

- Lambdas using EFS (Figure 3.4)

In order for us to run segmentation on a Lambda function, certain dependencies need to be taken care of. Examples are Tensorflow, image processing libraries and others. Lambdas use by default a very limited operating system provided by Amazon. This operating system might not have all the necessary binaries for us to do our work. The only thing we are guaranteed when using a Lambda is that the runtime is available. For runtime there are several choices like Node.js, Python, Ruby, Go, Java, C#, etc. In order for our code to have access to the dependencies it needs, we mount an extra volume to each Lambda that is spinned up. This volume contains everything that is necessary like for instance the segmentation models and Python-level packages. It is possible to define environment variables to inject these dependencies. This volume is called an EFS (Elastic File System) and is by design scalable. Note that this solution cannot be used for system-level dependencies like virtual screens. The only real guarantee is that the runtime will be available.

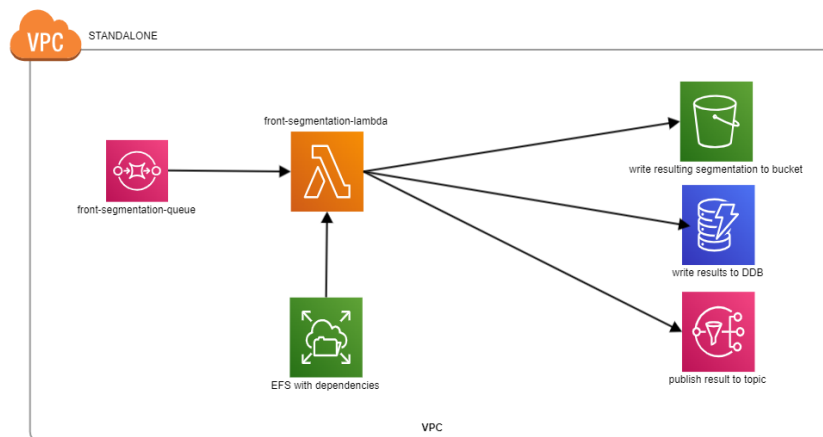


Figure 3.4: Lambda with EFS to provide dependencies

- Lambdas using ECR (Figure 3.5)

If we want to retain full control over the OS and specifically the dependencies that are used to execute our code we need to launch the Lambda using a Docker image. Here we will first make a Docker file containing the definition of the container. This definition includes the OS version and specific OS-level dependencies that are needed.

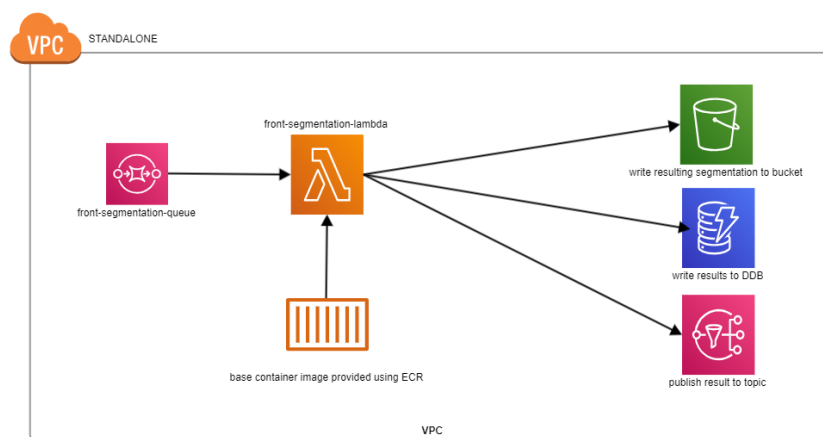


Figure 3.5: Lambda with ECR to provide base Docker images

- A mix of the above technologies (Figure 3.6)

Obviously all the suggested architectures have up and downsides hence we can combine multiple architectures together. We might have our segmentation and matching backed up with ECR images and use on spot instances for the optional segmentation refinement. All these functions communicate their status back to the services. The Public API has access to these resources in order to update the consumers of the API. We can show the status of processing in our mobile

application for example. User action can be demanded based on what is written to DynamoDB.

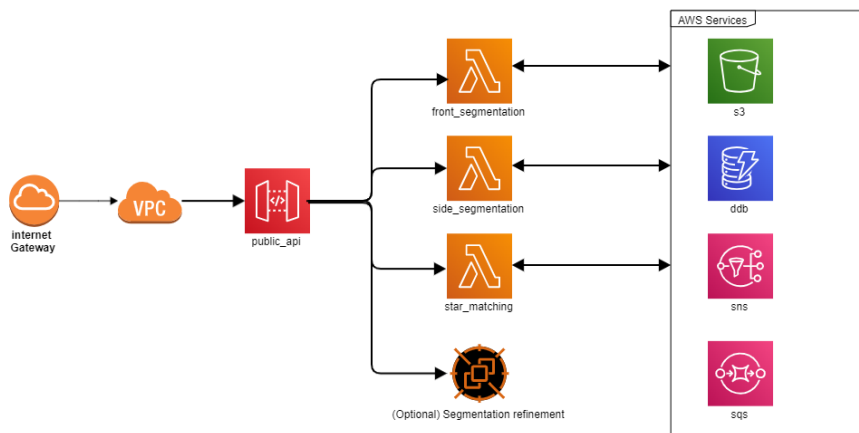


Figure 3.6: A mix of all possible architectures.

3.3.1 Costs

In this subsection, we show the impact of these new architectures on the costs. We will not be disclosing any actual costs related to QuantaCorp’s operations, instead a theoretical comparison will be made. Certain service costs like SQS, SNS, etc will not be taking into account for the sake of simplicity. Assume we will have to process scans for a whole work-week of 5 days. During these span of these 5 days 1500 people will be scanned during working hours. We will now focus on the machine cost for segmentation only. All prices will be based on what is currently displayed on the AWS pricing website¹, for the Europe(Ireland) region.

We start with the scenario where we use 24/7 running EC2 machines to do segmentation. In order for inference of the segmentation neural network to be very fast we will use compute optimized instances like c5.large or a c5.xlarge. Furthermore, we optimize these instances by finding out what the available CPU instruction set is and compiling a Tensorflow binary specifically for that instruction set. In Table 3.1 the hourly rate for these machines is shown. Given the assumption above, to run these machines for 5 days we could have a total cost of respectively 11.52 \$ and 23.04 \$. Although the costs are steep, we have a guaranteed availability for these instances and we can segment around the clock.

Name	On-demand rate per hour	vCPU	Memory
c5.large	0.096	2	4
c5.xlarge	0.192	4	8

Table 3.1: Costs for two types of C5 instances.

¹<https://aws.amazon.com/ec2/pricing/on-demand/>

In the second scenario we will be using spot instance that are only live during the actual scan sessions. In AWS it is possible to see the average hourly costs for these instances and their variation. Please take into account that spot instances have some downsides in terms of availability and start-up time. They may be interrupted at any time if the capacity is needed by AWS. In Figure 3.7 and 3.8 the price evolution for these instance types is shown in our zone. The average spot instance price for these type, is very stable. We will take this average price as starting point, resulting in a server cost of 4.02 \$ and 9.312 \$. This is a significant improvement over the on-demand price, but at the cost of availability.



Figure 3.7: Spot price evolution in the past 3 months for c5.large.



Figure 3.8: Spot price evolution in the past 3 months for c5.xlarge

In our third scenario we will use Lambdas to do our processing. These Lambdas do not suffer from an availability problem but have a start-up that is needed per invocation. That start-up time is rather random and is further explained in subsection 3.3.2. There are two scenarios, either the Lambda function already lives in memory or it has to be loaded into memory. This is what we call a hot start and cold start respectively. In any case, we only need 1500 invocations of the Lambda function in order to produce the same workload as in scenario 1 and 2. When using Lambdas the amount of vCPUs is not explicitly defined. It is implicitly defined by the amount of working memory the Lambda has. Again there is a handy calculator² to work out the costs. Based on experimental testing we have seen that when we allocate Lambdas with 10240 MB of working memory, the average execution time to run a segmentation is 2 seconds. The only way to reach

²<https://calculator.aws/#/createCalculator/Lambda>

this performance is on a hot start. The detailed calculation can be seen in Figure 3.9. This is however for 300 requests per day, so the total price for this method is 0.05 \$. However the actual price will be a lot higher since cold starts will be needed. Loading in a neural network model with a lot of layers on CPU-only is a very time intensive task. Based on our experimentation on average 120-130 seconds are needed to do this, for our specific model. So if we assume a cold start happens in 20% of the cases, our total cost would be 6 \$, which can be seen in figure 3.10. These numbers should be taken as a rough estimation of the costs. We have no control over whether a hot or cold start will be used.

```

Pricing calculations
1,500 requests x 200 ms x 0.001 ms to sec conversion factor = 300.00 total compute (seconds)
10 GB x 300.00 seconds = 3,000.00 total compute (GB-s)
3,000.00 GB-s x 0.0000166667 USD = 0.05 USD (monthly compute charges)
1,500 requests x 0.0000002 USD = 0.00 USD (monthly request charges)

Lambda costs - Without Free Tier (monthly): 0.05 USD

```

Figure 3.9: Price calculation to run 1500 invocations of the function, assuming hot starts

```

Unit conversions
Amount of memory allocated: 10240 MB x 0.0009765625 GB in a MB = 10 GB

Pricing calculations
300 requests x 120,000 ms x 0.001 ms to sec conversion factor = 36,000.00 total compute (seconds)
10 GB x 36,000.00 seconds = 360,000.00 total compute (GB-s)
360,000.00 GB-s x 0.0000166667 USD = 6.00 USD (monthly compute charges)
300 requests x 0.0000002 USD = 0.00 USD (monthly request charges)

Lambda costs - Without Free Tier (monthly): 6.00 USD

```

Figure 3.10: Price calculation to run 300 invocations of the function, assuming cold starts

To summarize this chapter, we have seen three scenarios and associated costs. Even though Lambdas are the clear winner, they bring with them the uncertainty of spin-up time. You never know whether you will get a cold or hot start. This also has an impact on performance, without segmentations our matching cannot start. So on a cold start scan the total execution time will be at least 120 seconds plus time to do all processing after segmentation.

3.3.2 Lambda Cold Starts

In subsection 3.3.1 the problem of cold start was raised. It is clear that for Lambdas to be usable within our current architecture, we need to avoid this scenario at all times during scan sessions. For this purpose several solutions were tried within the eTryOn-scope. These solutions all pre-emptively launch Lambda invocations with dummy data in order to make sure the next invocations will be hot starts.

The first method we would like to discuss uses our mobile app (for eTryOn the SDK) to initialize our infrastructure. A call is made to our Public API after which all necessary Lambdas receive a dummy message. There are now two things that can happen: either Lambda is already in memory and the message will be processed relatively fast or a cold start will happen and by the time the next message arrives on the queue, the Lambda should be in memory. Although there are no guarantees on the timing, we have seen during tests that in our typical scan session, the Lambdas are mostly hot starts.

The second method uses a daemon service that periodically puts a dummy message on all necessary queues for the scanning process. This method results in constant number of cold start (and thus constant cost) and works better for general availability. Currently we are using the first method in our staging environment. Since it fits better with the pay for what you use. It is activated based on user activity and not on perceived user activity.

3.4 Async Uploading

As explained in Section 3.1, moving from a monolithic architecture to a new server-less model has some implications. In case of our scanning process, the biggest change will be how images are uploaded to our cloud. In the current environment one call handles all scan-related data. The front and side image are uploaded in the same http payload. That payload was then passed on to the necessary services to be handled. Given the new possible architecture explained in figure 3.6, we can now use separate calls to upload the images. However this has several implications and in this section we will try to clarify those.

The first implication has to do with the scanning process. Given that the front and side scan are not taken at the same time, we should be able to upload them whenever they are ready. On average it takes a couple of seconds for a person to repose from front A-pose to side I-pose. The reposing seconds can be used to already upload the front image and any metadata related to taking that picture. This is not possible with the current architecture.

Our Public API was expanded to include these new calls. At the same time we moved our responder daemon from C++ to Python in order to integrate the STAR model, but this is also done as part of the effort to move to a serverless architecture. There are dependencies in our C++ code which are no longer being actively maintained. Moreover the QuantaCorp Development team prefers to work with Python over C++ for numerous reasons. In particular, the development time and readability of the Python code is a lot better. The responder daemon handles the payload of image uploads. Here several image specific validations are done, such as a check on resolution or a count of the number of channels.

If an image passes all potential checks, the message for image processing is put on the relevant SQS-queue. In the case of uploading a front image the first SQS message would be the UUID of the scan on the front segmentation queue. This queue is linked to the segmentation Lambda which in turn is linked to the refinement queue. Each of these processes update the status of the respective image within the Public API through SNS messages. There are multiple possible statuses which won't be further discussed in this

document. As a result, the produced masks will be put in S3 under the same UUID.

When a request from processing happens in the Public API the segmentation status for both front and side image is checked. If these status are "success" (no problems occurred), the actual processing through the body matcher can happen. Here, our responder again places a message on the relevant queue to trigger the whole matching chain. Thanks to updates via SNS, we could continually update the status of each scan for the end user.

In summary, in this section we explained how we split up our original upload scan call in multiple smaller calls. Allowing us to start processing faster and give more relevant updates to the user throughout the whole process.

3.5 Integration with eTryOn

In this section more information will be given on how the integration with eTryOn happens. Given the architecture that is in place we opted for a severless event driven approach as well. In order for us to avoid making user accounts and a middleware between all consortium services, we decided to use pre-signed URL's (see Deliverable D4.2 section 7.7 ADR-0014 Scanatar Creation). These pre-signed URLs contain the rights to write a file to an existing Google Cloud Storage (GCS) bucket.

We expanded our Public API to include an extra call which allows you to add all needed callbacks. This call uses multipart formdata to specify per key what the callback is. The process only supports predefined keys and each key reflects its own specific uploaded resource. One of the keys is called "ply_scape_callback". When processing all callback keys and this one is present, we will upload the matched ply within the scape model to the URL present in the value of the form data. In this manner multiple keys can be added which all represent a different request for information.

Once all the processing (segmentation and matching) is done, a final message is put on the callback queue. This queue is also tied in to a Lambda function which deals with processing the callbacks. The architecture is exactly the same as the generic severless model talked about in figure 3.5 and more generally in Section 3.3. A Lambda function will read all the relevant callbacks from DynamoDB based on the UUID of the scan. In the next step each key will be processed one by one. For each key a handler has to be written in python, dealing with the specifics of that key. Currently all keys supported are detailed in table 3.2.

Key	Resulting data
ply_scape_callback	Uploads the matched ply from the scape model
ply_star_callback	Uploads the matched ply from the star model
ply_morphed_callback	Uploads the morphed ply from the star model
etryon_meta_callback	Uploads a JSON-file with gender, height and optional etryon-field (defined by the SDK)

Table 3.2: All supported callbacks.

Chapter 4

SDK

4.1 Introduction

In our previous deliverable we explained what the Software Development Kit (SDK) and the supporting architecture would look like. In anticipation of running the asynchronous scanning architecture in a separate, dedicated virtual private cloud, we developed a first version of the SDK that will run on our production environment. Meaning, the first version of the SDK will differ from what was described in section 8 of deliverable D1.1. The reasoning behind this decision is twofold. First, given the new architecture is still under development and is required to develop the SDKs as envisioned in the aforementioned deliverable, we wanted to provide stability during the initial integration. Second, we didn't want to hold back the developers that need to integrate the SDK in the use case specific applications. This, to make sure the eTryOn development is on track and to get end-to-end tests running as soon as possible. Being able to test a system end-to-end early on in the project, can point out flaws we may have overlooked during our initial meetings on architecture and integration. This way, we mitigate the risk of having to make changes when the project is in full swing during the pilot phase. Failing early allows for changes that have minimal impact on other processes. As described earlier in section 3.5, we made it possible to add callbacks to a scan by extending the Public API's scan resource.

The main differences between what was envisioned and what is realised in the first version of the SDK are the following. First, there is no authorization using a JWT, instead the SDK implementer will have to authorize itself with QC's Authorization API using a client-secret for the basic authorization and credentials like a username and password to obtain an API token. Secondly, because we are using a production environment, the creation of a scan is not asynchronous. To upload scan data, a single HTTP call is used. Third, there are no introductory pages, instead we reimaged these pages as modules, and will implement those in the second version. And finally, again due to working on the production platform, the parametric body model used to generate the 3D mesh is still QuantaCorp's custom body model, of which the limitations are described in section 5.2 of Deliverable D1.1. Meaning, the result of a scan taken with the first version of the SDK does not make use of the STAR model yet.

At the time of writing, we have delivered a first version of the SDK as an Objective-C

framework and are working on its counterpart in ReactJS. As expected, we encountered some issues, which we explained in more detail in section 8.2 Architecture of Deliverable D1.1. The work that was put into the development of both SDKs is detailed in sections 4.2 Objective-C and section 4.3 ReactJS. In section 4.4 Integration with eTryOn we go over how the SDK ought to be implemented and go over the communication between the QuantaCorp cloud and eTryOn cloud. To conclude, we take a look at the user interface and experience in section 4.5 UI/UX.

4.2 Objective-C SDK

The Objective-C SDK consists of multiple classes and structs, some of them more important than others. Let's have a look at the main components of the SDK, and their related classes. There are two main components: one is responsible for the scan capture, the other is responsible for communication with the QuantaCorp Cloud. The classes and structs that are most important and most closely related to the capture of the scan data are as follows: QCScanController (see Code Listing 4.1), QCScanDelegate (see Code Listing 4.3), QCPicture (see Code Listing 4.6) and QCScanError (see Code Listing 4.7). Like so, the classes most related to the communication with the QuantaCorp cloud are: QCApiSession (see Code Listing 4.2), QCApiToken (see Code Listing 4.4) and QCCreateBodyDTO (see Code Listing 4.5).

```
//
//  QCScanController.h
//  QuantaCorp
//
//  Created by Thomas De Wilde on 30/09/2021.
//

#import <UIKit/UIKit.h>
#import <CoreMotion/CoreMotion.h>
#import <AVFoundation/AVFoundation.h>
#import "QCScanDelegate.h"
#import "QCScanConfiguration.h"

@interface QCScanController : NSObject

@property (weak) CMMotionManager *motionManager;
@property (weak) id<QCScanDelegate> delegate;
@property (readonly) QCScanConfiguration *configuration;

- (instancetype)init NS_UNAVAILABLE;
- (instancetype)initWithMotionManager:(CMMotionManager *)
    motionManager
withScanDelegate:(id<QCScanDelegate>)scanDelegate;

typedef void(^QCScanControllerSuccessBlock)(BOOL success, NSError *
    error);

- (void)presentScanView:(QCScanConfiguration *)configuration
parent:(UIViewController *)parent
animated:(BOOL)animated
completion:(QCScanControllerSuccessBlock)completion;
```



```

- (void)dismissScanView;

@end

```

Code Listing 4.1: QCScanController

```

//
//  QCApiSession.h
//  QuantaCorp
//
//  Created by Thomas De Wilde on 14/09/2021.
//

#import <Foundation/Foundation.h>
#import "QCApiToken.h"
#import "QCCreateBodyDTO.h"
#import "QCPicture.h"

@interface QCApiSession : NSObject

@property (nonatomic) QCApiToken *token;

- (instancetype)init NS_UNAVAILABLE;
- (instancetype)initWithClient:(NSString *)client
andSecret:(NSString *)secret;

typedef void(^QCApiSessionTokenBlock)(QCApiToken *token, NSError *
                                       error);
typedef void(^QCApiSessionIdentifierBlock)(NSNumber *identifier,
                                           NSError *error);
typedef void(^QCApiSessionSuccessBlock)(BOOL success, NSError *error)
;

- (void)authenticateWithUsername:(NSString *)username
andPassword:(NSString *)password
withCallback:(QCApiSessionTokenBlock)callback;

- (void)authenticateWithRefreshToken:(NSString *)refreshToken
withCallback:(QCApiSessionTokenBlock)callback;

- (void)createBody:(QCCreateBodyDTO *)body
withCallback:(QCApiSessionIdentifierBlock)callback;

- (void)createScanForProject:(NSNumber *)projectId
andBody:(NSNumber *)bodyId
withFrontPicture:(QCPicture *)frontPicture
andSidePicture:(QCPicture *)sidePicture
withCallback:(QCApiSessionIdentifierBlock)callback;

- (void)createCallbacks:(NSDictionary *)callbacks
forScan:(NSNumber *)scanId
withCallback:(QCApiSessionSuccessBlock)callback;

@end

```

Code Listing 4.2: QCApiSession

```

//
//  QCScanDelegate.h
//  QuantaCorp
//
//  Created by Thomas De Wilde on 28/09/2021.
//

#import <Foundation/Foundation.h>
#import "QCPicture.h"
#import "QCScanError.h"

@protocol QCScanDelegate <NSObject>

@required
- (void)didCancelScanCapture:(NSError *)reason;
- (void)didFailScanCapture:(NSError *)error;
- (void)didCaptureFrontPicture:(QCPicture *)picture;
- (void)didCaptureSidePicture:(QCPicture *)picture;

@optional
- (void)scanViewWillAppear;
- (void)scanViewWillDisappear;

@end

```

Code Listing 4.3: QCScanDelegate

```

//
//  QCApiToken.h
//  QuantaCorp
//
//  Created by Thomas De Wilde on 14/09/2021.
//

#import <Foundation/Foundation.h>
#import "QCJsonSerializableObject.h"

@interface QCApiToken : NSObject

- (instancetype)init NS_UNAVAILABLE;
- (instancetype)initWithAccessToken:(NSString *)accessToken
refreshToken:(NSString *)refreshToken
expiresIn:(NSTimeInterval)seconds
tokenType:(NSString *)tokenType;
- (instancetype)initWithAccessToken:(NSString *)accessToken
refreshToken:(NSString *)refreshToken
expirationDate:(NSDate *)expirationDate
tokenType:(NSString *)tokenType;

@property (readonly) NSString *accessToken;
@property (readonly) NSString *tokenType;
@property (readonly) NSDate *expirationDate;
@property (readonly) NSString *refreshToken;
@property (readonly) BOOL isExpired;

@end

```

Code Listing 4.4: QCApiToken

```

//
//  QCCreateBodyDTO.h
//  QuantaCorp
//
//  Created by Thomas De Wilde on 16/09/2021.
//

#import <Foundation/Foundation.h>
#import "QCJsonSerializableObject.h"

@interface QCCreateBodyDTO : NSObject <QCJsonSerializableObject>

- (instancetype)init NS_UNAVAILABLE;
- (instancetype)initWithCompany:(NSNumber *)companyId
withProject:(NSNumber *)projectId
withAlias:(NSString *)alias
withGender:(NSString *)gender
withHeight:(NSNumber *)height;

@property NSString *alias;
@property NSString *firstName;
@property NSString *lastName;
@property NSString *crmId;
@property NSNumber *userId;
@property NSNumber *companyId;
@property NSNumber *height;
@property NSNumber *weight;
@property NSString *gender;
@property NSString *notes;
@property NSNumber *linkToProject;
@property NSDictionary *custom1;
@property NSDictionary *custom2;
@property NSDictionary *custom3;
@property NSDictionary *custom4;
@property NSDictionary *custom5;
@property NSDictionary *custom6;
@property NSDictionary *custom7;
@property NSDictionary *custom8;
@property NSDictionary *custom9;
@property NSDictionary *custom10;

@end

```

Code Listing 4.5: QCCreateBodyDTO

```

//
//  QCPicture.h
//  QuantaCorp
//
//  Created by Thomas De Wilde on 16/09/2021.
//

#import <Foundation/Foundation.h>

@interface QCPicture : NSObject

```

```

@property NSURL *location;
@property NSDictionary *metadata;
@property (readonly) NSData *png;

@end

```

Code Listing 4.6: QCPicture

```

//
//  QCScanError.h
//  QuantaCorp
//
//  Created by Thomas De Wilde on 04/10/2021.
//

typedef NS_ENUM(NSUInteger, QCScanError) {
    QCScanErrorCameraUsageNotAuthorized,
    QCScanErrorBadInputDevice,
    QCScanErrorBadOutputDevice,
    QCScanErrorUnsupportedSessionPreset,
    QCScanErrorFailedToPrepareCapture,
    QCScanErrorFailedToCapture,
    QCScanErrorFailedToProcessCapturedData
};

```

Code Listing 4.7: QCScanError

The QCScanController is used to present the scan view and to define which object is going to act as the delegate of that scan view. So, in order to construct a QCScanController object, you need to provide an instance of the CMMotionManager and an object that implements the QCScanDelegate protocol. The CMMotionManager is required to extract accelerometer data when capturing an image, and is an input to the controller class. Reason being, it is best practice to make use of a singleton for the motion manager, as stated in the official Apple documentation¹. Once constructed, you can ask the controller to present the scan view in the same fashion you would present any other view controller in Objective-C. While the scan is presented, the SDK will take control of the motion manager. As soon as the scan view is dismissed, the control is given back. So, of course there is an option to programmatically dismiss the scan view. We made sure to mimic the life cycle of a view controller as well as possible.

The object that adheres to the QCScanController protocol will have to implement four required delegate methods and can implement two optional ones. The required methods are didCancelScanCapture, didFailScanCapture, didCaptureFrontPicture and didCaptureSidePicture. The first method is triggered in the scenario of a user cancelling the scan capture. The second method is triggered when a failure occurs. The implementer of both of these methods will act based on the NSError that is provided when these methods are invoked. The error object will provide an error domain, a code of an enumeration of QCScanError, and user info containing a description and the inner error that occurred. The third and fourth methods are triggered when the image is captured. Respectively,

¹<https://developer.apple.com/documentation/coremotion/cmmotionmanager?language=objc>

once for the frontal view of a person in A-pose and once for a lateral view of the person in I-pose. These methods receive a `QCPicture` object which contains the location of the captured image on disk and a dictionary containing the metadata for that image. There is a convenience getter method to fetch the PNG picture as bytes encapsulated in an `NSData` object. The `QCPicture` objects can be used to invoke the scan upload method of the `QCApiSession` class.

The `QCScanError` enumeration contains seven error codes. The first code is the `QCScanErrorCameraUsageNotAuthorized`. This code lets the implementer know that the SDK is unable to start streaming a camera feed because it is not authorised to use the camera. The second code and third code, `QCScanErrorBadInputDevice` and `QCScanErrorBadOutputDevice`, are related to the setup of the camera streaming pipeline. The former means the pipeline was unable to set up the camera as an input device for the video feed, the latter means the pipeline was unable to link a photo output to the camera which is required to capture high resolution images. Another error is `QCScanErrorUnsupportedSessionPreset`, which means the camera does not support the camera presets required by QuantaCorp. At the time of writing, we require an aspect ratio of 4:3 and the possibility to take 4K images. The final three error codes `QCScanErrorFailedToPrepareCapture`, `QCScanErrorFailedToCapture` and `QCScanErrorFailedToProcessCapturedData` are all related to the capture of the high resolution image. Because capturing a 4K image is a process that takes a couple of seconds, it is important to know which step failed in case something went wrong during capture.

For communication with the QuantaCorp API, the SDK implementer needs to create an instance of the `QCApiSession` class. The constructor for this class takes a client and secret, these credentials are required for basic authentication with QC's Authorization API. In order to request a token, additional credentials in the form of a username and password are required. When a user authenticates using one of the authentication methods, using either login credentials or a refresh token, the `QCApiSession` will hold on to the `QCApiToken` created when authentication is successful. After authorisation, the Public API methods can be called. In order to take a scan, a body needs to be created. So, the SDK implementer should collect the data required to create a body, put the data into a `QCCreateBodyDTO` object and invoke the `createScan` method of the `QCApiSession` (see Code Listing 4.5 to get an idea of the required data). After the scan is captured, we provide the pictures to the `createScan` method and invoke it to create a scan. If all data is valid, this POST call will return a scan identifier. This identifier is used to add callbacks to a scan with the `addCallbacks` method. This method accepts a dictionary with key-value pairs where the key indicates the type of callback that needs to be invoked at the end of the scan processing pipeline, and the value is the destination of the data provided by that callback action as a pre-signed URL.

To demonstrate the SDK to the consortium partners and provide them with code examples, a demonstration app was created. Figure 4.2. shows the main screen of this application. The form fields on this view are all required in order to take a scan. Username and password is used to fetch a `QCApiToken`, as described above, this token is required to make calls to QC's Public API. Customer and Project are the identifier fields of the customer and project a scan needs to be linked to, as well as a body ID. At the time of writing, QuantaCorp exclusively conducts business with companies that provide workwear garments and is therefore B2B focused. Typically, the person that is tasked

with gathering the measurements of the employees that need new garments would use our app as a tool to get these measurements instead of having to manually measure each person. That is why creating a scan is linked to a customer and project instead of a single entity. Height and Gender are required to create a body from which we need the ID to create a new scan. In order to complete a scan with the demo application, the following steps need to be taken. Step one is to request an API token. Step two is to create a body for a given company and project with the given height and gender. In step three the user taps the 'Take Scan' button to open the scan view which is used to take the pictures (see Figure 4.1). The next step is to create a scan object by uploading the pictures taken by the user together with the data gathered in the form. And the final step is to attach callbacks to a scan using the ID returned when the pictures are uploaded. These callbacks are used to have a loosely coupled way of exchanging data between the eTryOn cloud and the different systems of the consortium. In the demo app we provide the pre-signed URLs programmatically, because it is very tedious to have to write a URL in full. A more detailed explanation on the communication and interaction between apps, SDK and clouds can be found in section 4.4 Integration with eTryOn.

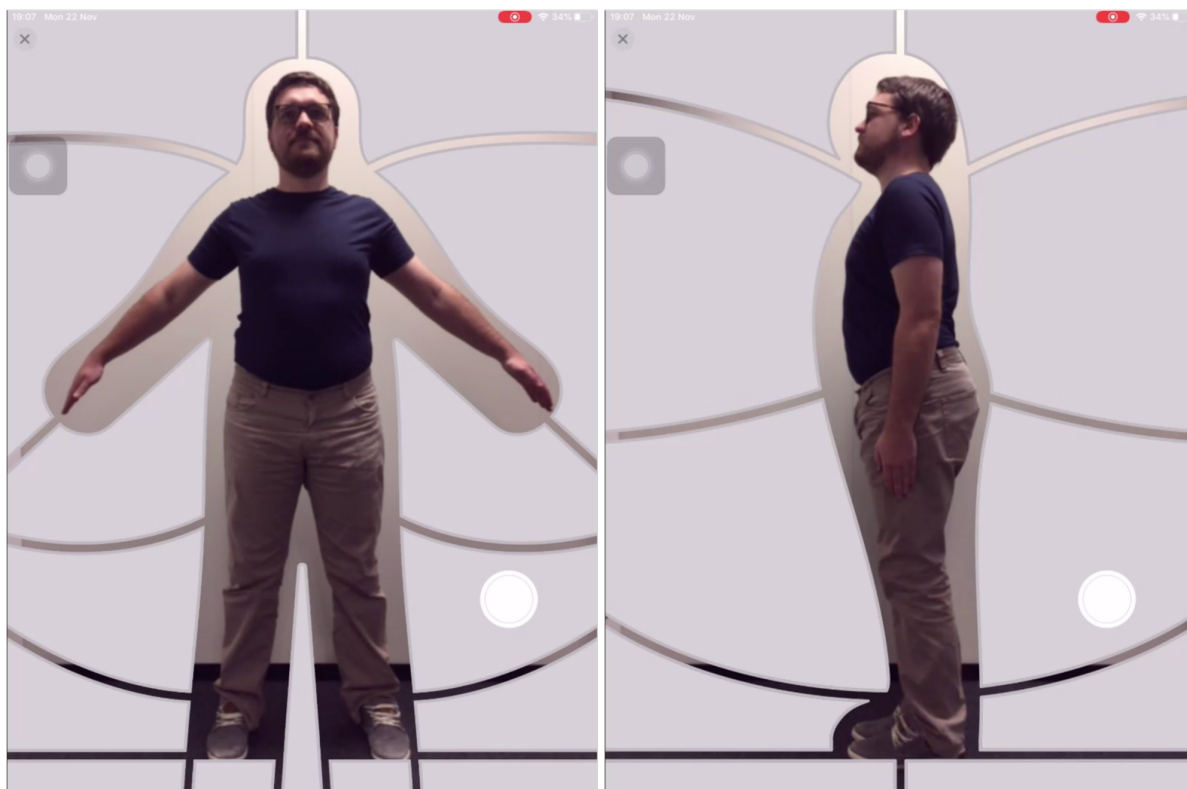


Figure 4.1: Screenshots of the scan view component. On the left is the UI of the front view, on the right the side view. The user needs to fit the scan subject inside the silhouette and press the capture button when ready.

The screenshot shows a mobile application interface with a status bar at the top displaying '19:07 Mon 22 Nov' and '34%' battery. The form contains the following fields and buttons:

- Username:** certh@etryon-h2020.eu
- Password:** (empty field)
- Request Token:** (blue button)
- Customer:** 879
- Project:** 1313
- Height:** 1700
- Gender:** U (selected), F, M
- Create Body:** (blue button)
- Take Scan:** (blue button)
- Upload Scan:** (blue button)

At the bottom, there are two small images showing a person's body scan from a front and side perspective.

Figure 4.2: Screenshot of SDK integration demo application. The form shows what the minimum required information is in order to take a scan.

4.3 ReactJS SDK

The ReactJS SDK is still in an early development stage. It knows more challenges in development than the Objective-C SDK. To maintain web safety and secure the privacy of its users, a browser's libraries and APIs are given limited access to system resources, certainly in comparison to libraries used by native applications on iOS and Android. As such, we encountered issues like being unable to access the back camera on iOS devices, and not being able to access system information like a hardware string. Also, detailed information about the camera specifications seem to be out of reach. Issues related to accessing the camera were resolved by using a different library. Most other issues were fixed once the application was hosted from a fully qualified domain name and was using HTTPS. This was required because of built-in safety mechanism of the Google Chrome browser and was the biggest hurdle to overcome.

Another issue encountered was cross-origin resource sharing (CORS). Typically a web app will send requests to an API that is running on the same domain, though in our use case, requests are sent from domain A to domain B. This needs to be fixed in the server configuration in order to allow CORS. We learned that browsers tend to implement CORS differently. The browser we mainly use during development and testing (Google Chrome) sends an OPTIONS call before every request. As a result, configuring the server for most browsers is the next step, which albeit time-consuming can be solved in a straightforward manner.

4.4 Integration with eTryOn

Within eTryOn, there are three applications being developed: VR Designer, MagicMirror, and DressMeUp. The two consumer facing apps, DressMeUp and MagicMirror, will be incorporating the SDK developed by QuantaCorp in order to add scanning capabilities to these apps. Every app will be able to communicate with the QuantaCorp cloud via API calls to either the Authorization API or Public API. The first step is to acquire an API token by authenticating oneself using credentials (username and password) with the Authorization API. The acquired token is then used to authorise any call to the Public API. The next step is to create a new Body. To create a new body, the integrator of the SDK will have to gather some required information on the person that will be scanned. At the time of writing, the minimum required data is height and gender. Height is required to properly scale the measurements, and gender is used to do body matching with a gender specific parametric body model. When the creation of the body is successful, an identifier (ID) is returned. Most, if not all, resources within the QuantaCorp Public API have a numeric identifier used to uniquely identify a resource. Next, we have the user of the app capture the two pictures using the SDK's scan view. This scan view also captures the metadata required to create a new scan. The minimum viable metadata is shown in section 2.3 of this deliverable. After the scan is captured, we can continue by creating a new Scan using the body ID from earlier, the ID of the project we're scanning on, and the two pictures with accompanying metadata. When the scan upload succeeds, an ID is returned. The scan is also now getting processed. First, the body silhouette is segmented out of the images. These are used as input to the body matching algorithm. Then the resulting avatar and metadata are sent to eTryOn using pre-signed URLs. So, in a final step on the SDK side, the apps implementing the SDK will create pre-signed URLs to the eTryOn Google Filestore at runtime. These are then used to make a POST call, attaching them to the scan. The final step in the scan processing is to check if any callbacks are coupled to the scan, if so, these are handled. From the callbacks listed in Table 3.2 we will use `ply_scape_callback` and `etryon_meta_callback` in the first SDK version. The first one will upload the avatar, the second will upload the metadata.

More detailed per use case diagrams of communication with the consortium partners, can be found on the techdocs site of the eTryOn project².

²<https://etryon.gitlab.io/techdocs/cloud-functions/avatar-creation/>

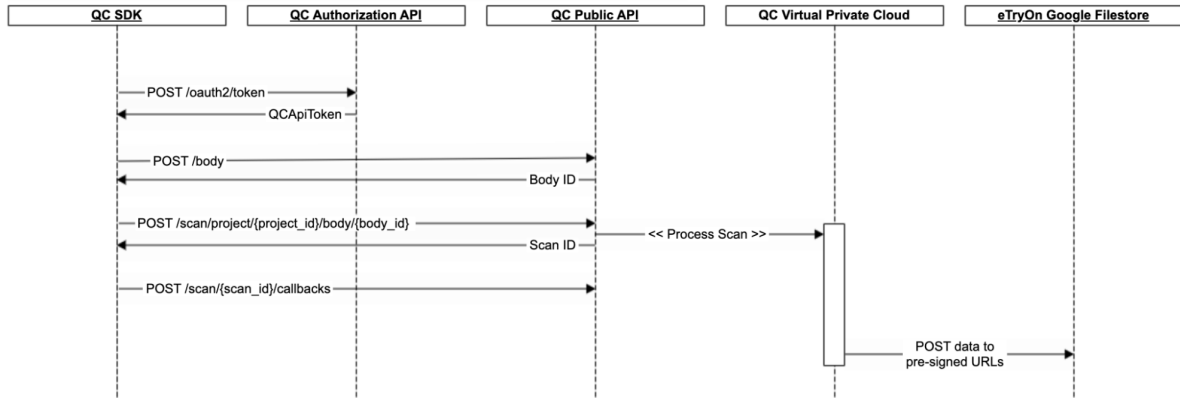


Figure 4.3: Communication between SDK, APIs and Clouds.

4.5 UI/UX

In the description of work for work package one there is mention of self-scanning. In this chapter we want to clarify what is meant by self-scanning and talk about the SDK's scan user interface (UI). At first, you might think about a UI for self-scanning that is operated by oneself. You might think this sounds like a good idea, but from QuantaCorp's past experience on self-scanning, this typically does not yield good scanning results. The main reason to not develop a self-scanning UI is the high chance of a bad user experience (UX). In a self-scanning scenario, the smartphone would have to sit at an angle and be kept at this angle with a self-crafted stand (books, a cup, ...). The chances are that this construction fails during scan capture. The next step in this scenario would be for the user to stand back and get scanned. This brings its own set of problems, for example, the screen is now much less visible. This prevents the user from getting clear visual feedback. In the past, we experimented with auto-capturing and audio feedback. But, we learned that solely relying on audio feedback did not work. Also, because the user doesn't know if he/she is doing a good job during the scan capture, the end result may very well be a failure. The only way to know if you succeeded is after you have taken the effort to take a scan. Besides UX, some technical limitations are also a driver to opt out of a UI that is operated by a single person. It's only been a couple of years since the first high-end smartphones appeared with support for high resolution image capture on the front cameras. Within the project, we need high resolution images to give users the most realistic avatars possible.

In the past couple of months, we iterated over the UI to be used in the SDK. The result of this is shown in Figure 4.4. We envision the use of an overlay and modules. Looking at the UI for a tablet, you can see we will make use of an overlay with a generic body silhouette is cut out. The purpose of this overlay is to indicate what pose a person should take, and it's up to the person who will take the scan to fit the scan subject within this overlay. On the right, we can see a vertical array of icons. These icons are called modules, and each module represents an important feature that needs to be in check while taking a scan. Starting from the top, the first module is the Pose. For now, this module will present an instructional text when tapped on what pose the person should take. In the future, we would like to make this module interactive by adding real-time pose detection. The Pose module's icon is then colored in with a certain hue

based on how well the standing pose scores. The second module is the Hair module. Same story here, at first this will show a modal view with instructions on how to tie up the hair, but in the future we would like to see this detected automatically. Using the same color scoring as described earlier. The third module is the Clothing module. Again, when tapped, this module will show information on how you are supposed to dress when scanning. In the past we experimented with an AI garment classifier, but it did not perform well enough to be put in production. More data and training is needed before we can add this Garment detection module. The fourth module is the Feet module. This module will show tips on how to position the feet when tapped. The final module is the Mat detection module. In our B2B solution, we make use of a mat to know the exact location of the camera in world coordinates, instead of estimating it. This module will not be used during the eTryOn project. The SDK will provide ways of configuring which modules to show, and in the future, we plan on using the SDK developed during eTryOn in QuantaCorp's B2B application. This way we guarantee a uniform scanning experience across different apps.

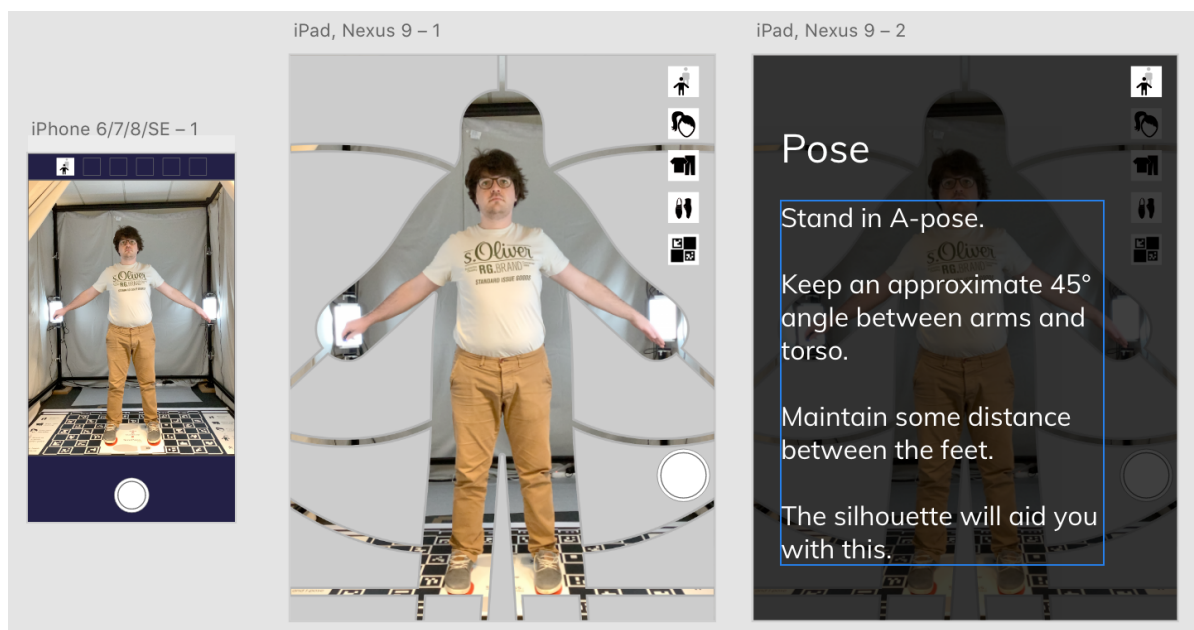


Figure 4.4: In the middle, the scan UI for tablets, on the left a layout for smartphones, and on the right an example of an instructional module.

Bibliography

- [1] K. M. Robinette, S. Blackwell, H. Daanen, D. Hoferlin, S. Fleming, and D. Burnside, “Civilian American and European Surface Anthropometry Resource (CAESAR) Final Report Volume I: Summary,” Tech. Rep. June, 2002.
- [2] D. Anguelov, P. Srinivasan, D. Koller, S. Thrun, J. Rodgers, and Davis, “SCAPE: Shape Completion and Animation of People,” in *ACM SIGGRAPH 2005 Papers on - SIGGRAPH '05*, vol. 24, no. 3. New York, New York, USA: ACM Press, 2005, p. 408. [Online]. Available: <http://ai.stanford.edu/~drago/Projects/scape/scape.html>
- [3] A. A. Osman, T. Bolkart, and M. J. Black, *STAR: Sparse Trained Articulated Human Body Regressor*. Springer International Publishing, 2020, vol. 12351 LNCS. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-58539-6_36
- [4] S. Prince, *Computer Vision: Models, Learning and Inference*, 2012.
- [5] Grivet Outdoors., “Clothing Size Chart - Chest, Waist, Inseam and Size to Inches Conversion.” [Online]. Available: <https://www.grivetoutdoors.com/pages/clothing-size-chart>
- [6] R. Misra, M. Wan, and J. McAuley, “Decomposing fit semantics for product size recommendation in metric spaces,” in *Proceedings of the 12th ACM Conference on Recommender Systems*. ACM, 2018, pp. 422–426.
- [7] X. F. Han, H. Laga, and M. Bennamoun, “Image-based 3d object reconstruction: State-of-the-art and trends in the deep learning era,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 5, pp. 1578–1604, 2021.
- [8] A. Ryberg, A.-K. Christiansson, B. Lennartson, and K. Eriksson, “Camera modelling and calibration - with applications,” in *Computer Vision*, X. Zhihui, Ed. Rijeka: IntechOpen, 2008, ch. 18. [Online]. Available: <https://doi.org/10.5772/6151>
- [9] Z. Tang, R. G. V. Gioi, P. Monasse, Z. Tang, R. G. V. Gioi, P. Monasse, J.-m. M. A. P. Analysis, Z. Tang, R. G. V. Gioi, P. Monasse, and J.-m. Morel, “A Precision Analysis of Camera Distortion Models,” *IEEE Transactions on Image Processing*, vol. 26, no. 6, pp. 2694 – 2704, 2017.
- [10] Apple Inc., “Class AVCaptureCalibrationData.” [Online]. Available: <https://developer.apple.com/documentation/avfoundation/avcameracalibrationdata>
- [11] Google Developers, “Camera API .” [Online]. Available: <https://developer.android.com/guide/topics/media/camera>

- [12] M. Persson and K. Nordberg, “Lambda Twist: An Accurate Fast Robust Perspective Three Point (P3P) Solver,” in *ECCV*, vol. 11208 LNCS, 2018, pp. 334–349.
- [13] Belongie, Serge, “Rodrigues’ Rotation Formula.” [Online]. Available: <https://mathworld.wolfram.com/RodriguesRotationFormula.html>