Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

AR-Based Virtual Try-On API for Fashion Retailers

By
Saim Sohail

September 2, 2025

Masters Thesis

Submitted in partial fulfillment for the award of **Master of Science in Computing** to the Department of Computer Science & Applied Physics, Atlantic Technological University (ATU), Galway.

M.Sc. in Computing

Contents

List of Figures

List of Tables

Abstract

Customers had to grapple with not being able to assess whether products like glasses, hats, or clothing would suit them due to the swift rise of online fashion retail. This dissertation tackled the issue with AR-enabled virtual try-on technology that can plug into fashion e-commerce with minimal disruption. As opposed to previously implemented native or SDK-burdened try-on systems, this work implemented a *front-end-agnostic FastAPI backend* that listened for frames from any web client, did the detection and overlay work on the server, and streamed the finished composite in real time, allowing for drop-in stitching to existing retail sites. The modular solution architecture described above was realised in practice. A FastAPI backend served as the core engine, which hosted a coupled YOLO object detection with MediaPipe landmark estimation facial and pose recognition to extract key features of the face and pose. Subsequent overlay algorithms coordinated the positioning of product assets, e.g., glasses or hats in the form of transparent PNGs to the associated regions. A React front-end was created as a reference client for a faux store catalogue to initiate try-on sessions. The integration was done through FastAPI calls that took video frames as inputs, processed them with overlays, and returned the composites in real time. The system demonstrated the feasibility of delivering real-time AR try-on experiences through a purely web-based architecture without requiring native applications. Key contributions included the design of a backend-first API for try-on functionality, the integration of machine learning models with real-time streaming, and the development of a reusable product-to-asset mapping strategy. The resulting platform provided a foundation that retailers could extend to additional product categories, enhancing customer engagement and helping to reduce uncertainty in online purchases.

Acknowledgments

I would like to begin by expressing my sincere appreciation to my supervisor, Douglas Mota Dias, for the guidance, support, and constructive criticism provided during the entire duration of this research. His sharp attention and guidance greatly influenced the overall approach of this work. He assisted me in not just helping me, but also helped me in forming my ideas, creating my action plan, and improving my roadmaps and timelines.

I would also like to thank the faculty and staff of the Computer Science and Applied Physics Department, Atlantic Technological University, Galway, for facilitating a stimulating academic environment, providing the required infrastructure, and supporting the completion of this work. I value the flexibility given to me with regards to the independent project and the ability to select and shape a self-defined scope of work.

Lastly, I would like to thank my close friends and family for the motivation, understanding, and support throughout my academic journey. Their faith in me provided so much motivation and strength during this journey.

Chapter 1

Introduction

Along with transforming consumer engagement with products, the rapid growth of the online fashion retail industry created challenges in assisting shoppers in evaluating products without physical interaction. Traditional e-commerce websites still relied on static images to convey the items on display. As shoppers could not assess the fit, scale, or style of the product, confidence in purchasing fashion items online took a hit. As a result, higher return rates were observed. The emergence of Augmented Reality (AR) technology has proven to be a solution, allowing customers to virtually try on items, thus eliminating the gap between digital catalogues and actual products [1]. Earlier versions of AR try-on solutions were implemented in mobile apps and in dedicated retail systems, but many such solutions were bound to specific devices and platforms, creating problems with scalability and reusability. This project, on the other hand, designed and implemented a try-on backend that was *front-end-agnostic.* It was done using FastAPI: any website could frame a video and submit the frames to the server. Detection, overlay, and compositing were done in real time. This style of design kept the user interface separate from AR logic and allowed easy integration into existing retail storefronts without needing native dependencies. As earlier work processed client-heavy pipelines, a detailed survey of related approaches and their trade-offs appears in Chapter 3.

1.0.1 Objectives.

The scope of the project was organized as follows:

- **To create and deploy a frontend-agnostic backend API for AR try-on.** Most AR try-on systems are either embedded within mobile applications or are tightly coupled with device SDKs such as ARKit or ARCore. These

systems cannot scale and are difficult to integrate with other applications [2, 3].
This project aimed to develop a backend-first pipeline with FastAPI, providing AR try-on services as a backend web service, consumable by any retailer's frontend (web or mobile) with minimal changes.
This approach maintained a separation of presentation (frontend) and computation (backend) for the system, which improved the integration for existing e-commerce systems.

- **To integrate state-of-the-art computer vision models for robust detection and alignment.**

  To accurately detect and digitally align facial accessories, several models had to be integrated:

  Robust face detection using the MediaPipe Face Mesh [4] for detailed landmarking was combined with YOLOv8 [5] focused on face detection.

  These improvements made it possible to accurately scale and rotate face overlays well beyond the capabilities of older skin-region or Haar Cascade based methods [6].

- **To create a modular overlay placement pipeline for extensible rendering of assets.**

  This [3] project focused on developing a comprehensive overlay composition pipeline with OpenCV, unlike prior systems tailored specifically to glasses.

  The pipeline supports a variety of fashion assets, such as glasses, hats, and apparel, along with the capability to be extended to new categories without the need for reengineering due to the abstraction of warping, transformation, and alpha blending.

- **To create a controlled and retailer managed product–asset mapping system.**

  In order for retailers to manage overlays, AR assets need to be dynamically linked with catalog items.

  This project provided a solution by developing a mapping system that linked product IDs with asset files, thereby streamlining catalog changes without necessitating additional coding on the client's end.

  This separation enables these systems and components to be designed and built independently, compliant with modular and scalable design principles, and facilitates multi-product catalogues.

- **To build a demonstration client for validation and integration testing.**

Realtime video frame capture, API communication, and overlay display functionality were integrated into a React-based demo frontend.
Proof-of-concept integration enabled the team to validate system usability, capture end-to-end latency performance, and evaluate the responsiveness of a backend-first design.

- **To determine the performance outcomes and experience of the users.** As part of the project's scope, performance metrics such as responsiveness, which entails frame processing latency, alignment accuracy as overlay fitting to landmarks, and the user experience (UX) feedback were evaluated for different devices and network conditions. This evaluation was essential to understand how close the project was to completing a functional prototype of the system which could be used in practice and to measure the impact of the backend-first strategy in comparison to the existing client-heavy solutions for AR try-on systems [7, 8].

1.0.2 Contributions.

This paper issues the following contributions:

- **Frontend-agnostic AR architecture:** Worked-on backend-centric architecture of AR try-on systems using FastAPI. This architecture, in contrast to previous approaches that used mobile AR applications or ARKit/ARCore driven SDKs, provides an AR try-on web API.
This architecture allows integration into existing retailer websites and mobile clients eliminating the need for native dependencies and enabling cross-platform scalability and reusability.
- **Real-time detection and overlay pipeline:** Designed and implemented backend-processed frame-by-frame AR processing using YOLOv8 for face and body detection and landmark estimation with MediaPipe.
These components were orchestrated with FastAPI and OpenCV to provide overlays as FastAPI and OpenCV.
Computational backend processing reduced the burden on client devices improving the user experience while ensuring consistent system performance across varying hardware configurations.
- **Generalized alignment strategy for multiple product categories:** Developed an alignment and transformation pipeline extending beyond glasses to include hats, apparel, and other fashion accessories. Parameterized facial

features provided landmark-based scaling, rotation, and translation enabling extensible overlay placement for diverse assets.
This design enhances the system's adaptability and ensures its longevity for application in retail beyond eyewear.

- **Demo client for validating workflows and measuring performance metrics:** Developed a React frontend demo for e-commerce virtual try-on simulations. The client application processed video frames, sent them to the backend API for processing, and rendered the overlays returned from the server in real time. This provided measurements of integration and end-to-end latency span, responsiveness, and the complexity of integration, validating the feasibility of a web-only AR pipeline compared to client-heavy approaches documented in the literature.
- An **architecture based on a sequence of actions** was put in place: beginning with the submission of frames based on HTTP requests, which was then enhanced with additional WebSockets in parallel as a realtime, high-accuracy stream, and a single-image try-on system. Dimensional accuracy was achieved with interpupillary distance (IPD) scaling and product dimension mapping.

Roadmap. Chapter 2 outlines the methodology and system architecture; Chapter 3 reviews related work; Chapter 4 presents the design and implementation; Chapter 5 reports the evaluation; and Chapter 6 concludes with limitations and future work.

Chapter 2

Research Methodology

This chapter outlines the methodology adopted for this research project. Two complementary perspectives are addressed: the *research methodology*, which frames how the research problem was approached, and the *software development methodology*, which describes how the system was engineered and evaluated. Together, these form a rigorous and valid basis for developing and assessing the proposed AR-based virtual try-on API.

2.1 Research Methodology

The primary goal was to test whether a backend-centric API can support an integration-ready AR-based virtual try-on system for fashion retail. Answering this question called for a **design science research** (DSR) approach, which is popular in computing and information systems for the design and evaluation of artefacts aimed at resolving real-world issues [9]. One of the most valued features of DSR is the focus on iterative loops of construction and assessment, which is ideal for this project and its experimental approach.

2.1.1 Approach

The project starts with a **literature review** on the most recent advancements in AR try-on technology. Furthermore, their shortcomings are examined. This review identified a research gap: most existing solutions are frontend-centric, rely heavily on specific devices, and do not support standardised backend APIs [2, 3]. Based on this, the research question was sharpened: *is it possible to achieve real-time AR try-on integration into retail applications through a lightweight, stateless backend API?* Then problem framing was used to define the artefact; a backend API, front-end agnostic, enabling AR try-on. With Iterative Construction, we started with HTTP streaming, then expanded to WebSocket pipelines and dimensional calibration. Finally, evaluation tested three modes (single image, real-time, high accuracy) across accuracy, latency, and ease of integration.

2.1.2 Model Integration and Evaluation

The product created to resolve this issue is a REST API-based which lets users upload their photos or images so that virtual overlays of fashion items can be applied. The evaluation was both *technical* (functional precision of overlays and delay performance) and *conceptual* integration readiness and scalability analysis. There was iterative testing to make sure that every subcomponent (YOLO detection, MediaPipe landmarks, OpenCV overlay) worked seamlessly together [5, 10].

2.2 Software Development Methodology

To engineer the solution, an agile-inspired incremental development methodology was adopted [11]. The project was broken down into individual modules, which was the best approach to take considering the complex structure of the project. The integration of computer vision, API design, and the frontend required a lot of trial and error, which was possible due to the incremental approach in the methodology.
Agile-inspired incremental development was used. Each component—YOLO detection, MediaPipe landmarks, OpenCV overlays, WebSocket streaming—was built and validated in modular sprints.

2.2.1 Backend Development

The backend was implemented in **FastAPI** (Python) due to its lightweight design, asynchronous request handling, and auto-generated API documentation via Swagger [12]. The core modules include:

- **YOLOv8**: pre-trained object detection model used for real-time person and face detection [5].
- **MediaPipe**: library for extracting facial landmarks, providing geometric anchors for accessory placement [13].
- **OpenCV**: used for overlay composition, including scaling, rotation, translation, and alpha blending of PNG assets.
- **Dimensional Accuracy Subsystem** using interpupillary distance (IPD) to calibrate product scaling in pixels.

The backend is stateless: each request carries its complete context (productId and image/frame), and no persistence layer (database) is included. This ensures lightweight integration into third-party retail applications.

2.2.2 Frontend Development

A simple demo client was built using **React (Vite)** and styled with **Tailwind-CSS**. This frontend is not the main contribution of the research, but it plays an important role in showing how the backend API can actually be used in practice. It acts as a testbed to make sure the API works as intended. The client includes:

- A catalogue and product page where users can browse items and choose something to try on.
- A studio page that connects to the webcam and sends video frames to the backend API for real-time processing via WebSockets.
- A live preview area where the processed frames returned by the API are displayed with the virtual accessory overlaid on the user's face.
- Option to switch between three-tier pipelines (image, real-time, high accuracy).

2.2.3 Integration Process

When a product is selected, its `productId` is sent with each frame to the backend via a REST call. The backend retrieves the corresponding PNG asset (stored in the API's static assets), applies detection and overlay, and returns a processed frame. Each frame/productId is transmitted with metadata. WebSocket pipelines ensured smooth streaming, while dimensional scaling adjusted product overlays based on user-specific IPD.

2.3 Validity of the Approach

The chosen methodology is valid for several reasons:

- **Design Science Research:** The project applies the **Design Science Research** (DSR) principles by creating a novel backend API that deals head-on with the research challenge of integration-ready AR try-on. In design science [1], the artefact is an engineered system tested iteratively in conditions that are real-world inspired.

- The updated implementation notably achieves industry-standard precision in face alignment by robustly detecting using YOLOv8, extracting 468 landmarks with MediaPipe FaceMesh, and applying an interpupillary distance (IPD)-based dimensional accuracy model while enabling fallback and fusion between detectors for reliability [10, 5, 13].

- **Integration-first scope:** Integration is an important system design principle. The backend still is lightweight and stateless but now has added support for three distinct try-on modes: (high quality, medium speed) REST-based single image, (adaptive accuracy for responsiveness) real-time streaming WebSocket, and high accuracy streaming (all methods enabled, slower). Integrating these features emphasize the system's high integration with retails yet minimal database connection overhead [14].

- **Iterative Development:** The combination of overlay pipeline modules (detection fusion, dimensional accuracy, WebSocket transport) was accomplished through iterative development. Each module was developed, and tested independently. [15] suggests that using this approach, accuracy and latency can be thoroughly evaluated.

2.4 Summary

In summary, this project integrated a design science research methodology with an agile development process. The investigation was based on a research problem which was identified in the literature gaps, and the answer was crafted in the form of a modular backend API, which was validated with a demo frontend. This approach balances scholarly rigour with practical relevance, which undergirds the outcomes and assessment given in the next chapter.

Chapter 3

Literature Review

This chapter reviews the state of the art in augmented reality (AR) for fashion retail with a particular emphasis on *backend-centric, integration-ready APIs* for virtual try-on. It is tightly coupled to the thesis objective: to design a lightweight AR try-on API that retailers can integrate into existing products with minimal coupling. I synthesise findings across AR user experience, mobile rendering, virtual garment simulation, modern detection models (YOLO family), and scalable API/serving architectures. The review moves beyond description to evaluate trade-offs, limitations, and gaps relevant to a backend-first design.

Earlier versions of AR try-on solutions were implemented in mobile apps and in dedicated retail systems, but many such solutions were bound to specific devices and platforms, creating problems with scalability and reusability. This project, on the other hand, designed and implemented a try-on backend that was *front-end-agnostic.* It was done using FastAPI: any website could frame a video and submit the frames to the server. Detection, overlay, and compositing were done in real time. This style of design kept the user interface separate from AR logic and allowed easy integration into existing retail storefronts without needing native dependencies. As earlier work processed client-heavy pipelines, a detailed survey of related approaches and their trade-offs appears in Chapter 3.

3.1 Objectives, Questions, Scope

Objective. Identify approaches and evidence that inform a robust, reusable, backend-centric AR try-on API for fashion accessories (e.g., glasses, hats).

Research questions.

- What approaches dominate AR try-on today (frontend-heavy vs. backend/API-first), and how do they compare?

- How do modern object detectors (e.g., YOLO variants) enable real-time try-on alignment?
- What are the architectural gaps in scalable, platform-agnostic, backend AR systems?

Scope. Peer-reviewed CS/HCI/CV sources from the last 5–7 years (with selected seminal works) on AR retail, detection/landmarks, and ML serving; consumer-behaviour-only studies with no technical implications are de-emphasised.

3.2 Methodology

3.2.1 Keywords, and Time Window

Searches were conducted on **IEEE Xplore**, **ACM Digital Library**, **SpringerLink**, and **ScienceDirect**; preprints were cross-checked on arXiv when appropriate. Representative keywords and Boolean combinations:

> "augmented reality" AND ("fashion" OR "retail" OR "virtual try-on"),
> "virtual try-on" AND ("backend" OR "API" OR "cloud" OR "serving"),
> "YOLO" AND ("real-time" OR "landmarks" OR "face" OR "person")

The main time window was 2019–2024, with earlier works included if foundational (e.g., multi-camera virtual mirrors).

3.2.2 Inclusion, Exclusion, and Screening

Inclusion. Relevant peer-reviewed publications are those that: (i) implement AR try-on and its close analogues (mirrors, wearables); (ii) present detection/landmarking algorithms usable for try-on; or (iii) describe scalable machine learning (ML) serving/API design frameworks pertinent to AR.
Exclusion. Non-peer-reviewed works; use of AR technology in other unrelated fields that cannot extrapolate approaches; Non-demos that do not provide a description of the methodology are device-only demos.
Screening. Methodological and architectural relevance necessitated full-text assessment after title/abstract screening. Prioritisation for the final set focused on works that inform backend/API design concerning latency/throughput and deployment.

3.2.3 Quality Appraisal and Data Extraction

Each study had the following extracted: goals, methods, datasets and hardware, reported latency or frames per second (FPS), architecture coupling (frontend vs backend), listed limitations, and integration implications via REST/HTTP. Preference was provided to datasets with runtime metrics and reproducible pipelines.

3.3 Thematic Synthesis and Critical Analysis

3.3.1 Frontend-Centric AR Systems

Mobile AR and virtual mirrors typically display overlays on-device (e.g., ARKit/ARCore, marker/markerless). Sensor fusion is tight and network dependence is low for these systems, but they are device-bound and often lack model portability. Furthermore, these systems are inefficient for user experience (UX) integration studies because they push compute and maintenance infrastructure to clients, limiting cross-platform reuse and back-office control. Lau and Ki [16] and Rhee and Lee [17] emphasise consumer engagement through mobile AR apps, but these rely heavily on ARKit/ARCore.

3.3.2 Multi-Camera and Device-Heavy Approaches

Hauswiesner et al. [18] and Eisert et al. [19] explored 3D garment overlays using multi-camera rigs and visual hull algorithms. Specialised hardware, such as multi-camera rigs, eases complex calibration and deployability, but at the expense of geometric fidelity (e.g., silhouette-based visual hulls). The scalability perspective renders these system approaches devoid of practicality for web and mobile-grade integration.

3.3.3 Lightweight Mobile Implementations (OpenCV/TFLite)

Ling [20] and Hashmi [21] implemented mobile AR try-ons using OpenCV and TensorFlow Lite. Pipelines for lightweight mobile setups, such as OpenCV and TFLite, are constrained to model capacity, power, and thermal throttling. Accuracy and robustness, particularly under occlusions and lighting variation, as well as centralised distributed improvement sharing, pose challenges. This provides motivation for server-side compute with thin clients.

3.3.4 Object Detection and Landmarks: YOLO and Beyond

The YOLO family (Redmon et al. [10], Bochkovskiy et al. [22], Jocher [5]) enabled advancements in real-time detection for try-on alignment (person/face regions and keypoint proxies) and does so with a reasonable accuracy-latency tradeoff. YOLO coupled with face landmarking (i.e. MediaPipe) and a composition engine (OpenCV) forms a modular backend pipeline that offers processed frames or pose metadata as outputs to the clients. Modern detectors, unlike legacy Haar cascades or handcrafted features, **generalise better** and can be updated continuously on the server side without the need to redistribute binaries to the user devices.

3.3.5 Backend/API-Centric ML Serving

A critical gap in AR try-on research is the lack of strong backend or API infrastructure. Although Yuan et al. [2] and Zhang [3] introduced glasses try-on systems, their implementations are tied to specific devices and don't provide reusable APIs. On the other hand, machine learning platforms such as TensorFlow Serving [23], Clipper [24], and Prediction Serving frameworks [25] present scalable inference models using REST or gRPC. Research into AR offloading [7, 8] emphasises shifting computations to edge or cloud resources to lighten device workloads and allow centralised updates. Additionally, privacy-focused AR offloading [26] introduces a new layer of security in deployment.

This thesis supports this gap by introducing:

- **WebSocket streaming** to augment real-time interactivity ond REST calls.
- **IPD-informed static originating limb models** and the inter-pupil distance (IPD).
- **Three-tiered pipelines** balancing vertical and horizontal (single image, real-time, and high-accuracy).

3.4 Comparative Taxonomy and Pipelines

To clarify positions in the literature, I present two visuals.

3.4.1 Taxonomy: Frontend vs Backend AR Try-On

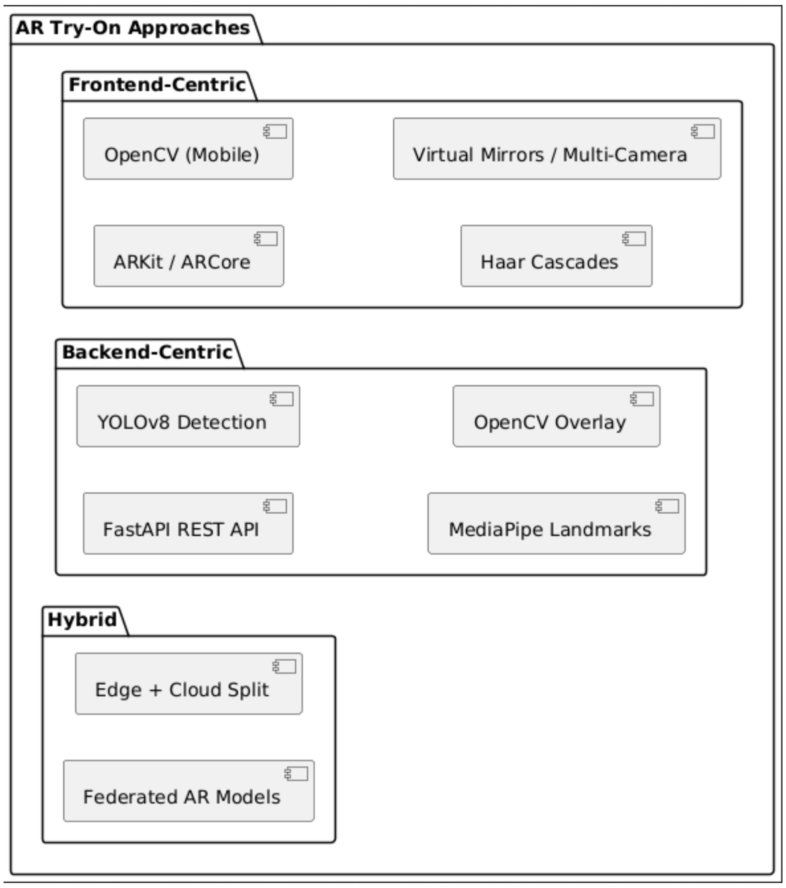


Figure 3.1: Taxonomy of AR Try-On approaches

From the literature review, Figure 3.1 presents a taxonomy of AR try-on methodologies and captures all of the approaches identified.
The figure classifies the frameworks into three categories:
(i) **Frontend-centric** approaches that utilise the device AR frameworks such as ARKit or ARCore [16, 17], or less complex OpenCV for mobile [20]. Earlier works have been based on body part detection using Haar cascades [21], and multi-camera "virtual mirrors" for garment augmentation [18, 19].
(ii) **Backend-centric** approaches that perform part of the work on a separate server and return processed results through, for instance, OpenCV for landmark extraction, overlay composition, and landmark detection using YOLO [10, 5] and MediaPipe [13]. This also fits into API-centric ML serving architectures as TensorFlow Serving or Clipper [23, 24].
(iii) **Hybrid** AR that partition processing tasks between the device and server (edge + cloud) [7, 8] or utilise federated learning aimed for privacy-preserving AR [26].

3.4.2 Pipelines: Frontend-Heavy vs Backend API-Based

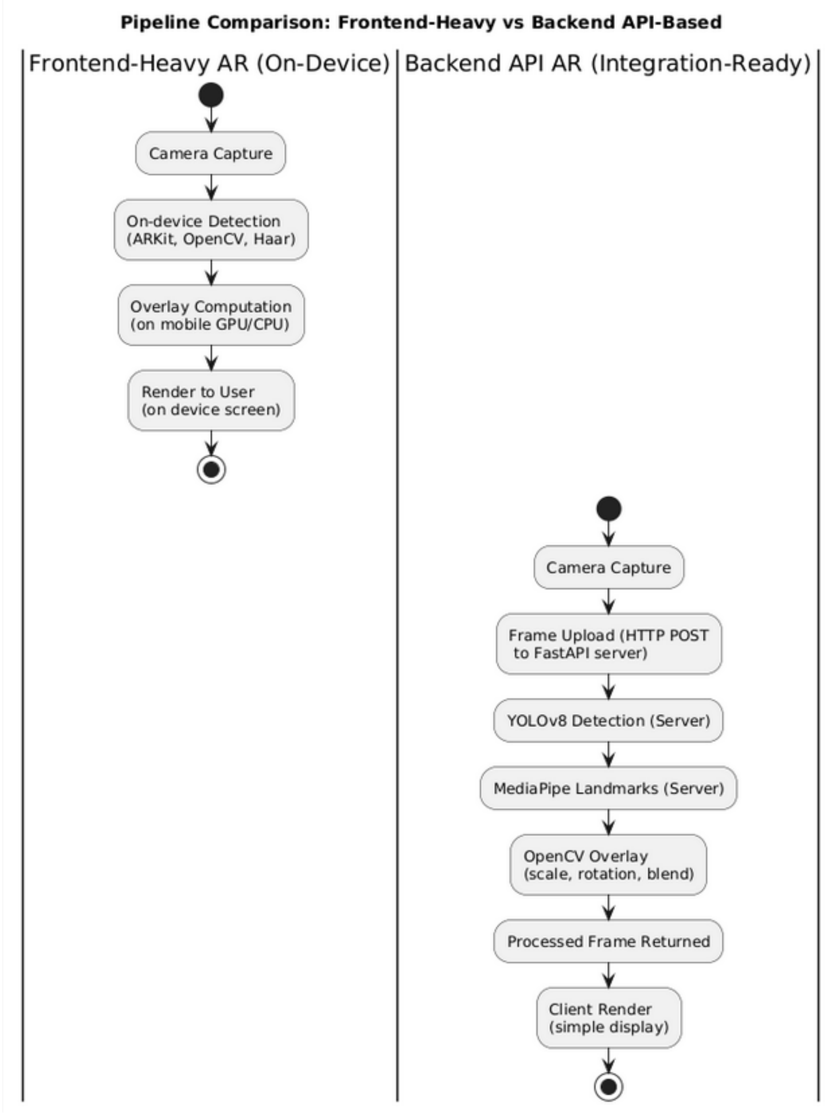

Pipeline Comparison: Frontend-Heavy vs Backend API-Based
Frontend-Heavy AR (On-Device)
Backend API AR (Integration-Ready)
Camera Capture
On-device Detection (ARKit, OpenCV, Haar)
Overlay Computation (on mobile GPU/CPU)
Render to User (on device screen)
Camera Capture
Frame Upload (HTTP POST to FastAPI server)
YOLOv8 Detection (Server)
MediaPipe Landmarks (Server)
OpenCV Overlay (scale, rotation, blend)
Processed Frame Returned
Client Render (simple display)

Figure 3.2: Pipeline comparison

As shown in Fig. 3.2, there are distinctions between frontend-heavy AR workflow pipelines and those utilising backend API based architectures.
In **frontend-heavy AR systems**, the workflow can be described in four basic steps: the camera input is captured, ARKit, OpenCV, or Haar cascades detections are performed, overlays are computed using the device's GPU/CPU, and the

results are rendered [16, 21, 20].
Although these systems provide low latency and can work offline, they are constrained to the device and often limited in range due to mobile hardware capabilities.
In contrast, the backend API-based pipeline shifts computation to a server. In this case, the client captures frames and sends them to FastAPI, where person/face detection is performed using YOLOv8 [10, 5], and landmark alignment is done using MediaPipe [13]. Overlay transformations of the frame, such as scaling, rotation, and blending, are done using OpenCV.
The frame is then sent back to the client to be rendered with lightweight computation. The benefits of this architecture include scalability, ease of cross-platform adoption, and centralised updates [23, 24], although these are offset by concerns of bandwidth and privacy [7, 26]. Overall, Figure 3.2 highlights the fundamental trade-off between on-device AR (speed, but device-coupled) and backend API AR (scalable, reusable, but network-dependent).

3.4.3 Comparative Table

Table 3.1: Comparison of AR-based try-on approaches in the literature. Each row lists the stack, strengths, limitations, and representative studies.

Approach + Typical Stack	**Strengths**	**Limitations**	**Studies**
Frontend–centric mobile AR *(ARKit/ARCore; OpenCV on-device; JS/WebGL clients)*	Low latency; works offline; zero server cost; tight UX integration	Device-dependent performance; fragmented platforms; energy drain on mobile; limited model size	[16, 17, 20]
Multi-camera "virtual mirror" kiosks *(Depth/RGB camera rigs; visual hull tracking; local rendering wall)*	High realism in-store; robust tracking with controlled environment	Hardware/space cost; low portability to home/mobile; not scalable	[18, 19]
Image-based virtual try-on (2D warping/generation) *(Cloth warping + synthesis nets VI-TON, CP-VTON on still images)*	High perceptual quality on product photos; supports catalogue imagery	Batch/offline only; not live; requires clean product/user images; alignment errors on extreme poses	[27, 28]
Backend API (detection + landmarks + overlay) *(this thesis) (FastAPI REST + YOLOv8 + MediaPipe (RMPE) + OpenCV overlay)*	Scalable; centralised updates; reusable across clients; reproducible evaluation	Needs network bandwidth; privacy concerns; server costs	[10, 5, 13]
Prediction serving / MLOps backends *(TF-Serving; Clipper; TFX pipelines; REST/gRPC contracts)*	Mature ops stack; versioning; A/B testing; low-latency inference	Higher engineering overhead; tooling lock-in risks	[23, 24, 29, 15]
Edge+Cloud hybrid offloading *(5G MEC servers; client preprocessing + server inference)*	Lower latency vs pure cloud; reduced uplink bandwidth	Edge infra varies; orchestration complexity; model sync issues	[7, 8, 30]
Privacy-preserving / federated variants *(Encrypted offloading; federated learning personalization)*	Improves privacy; personalisation without raw data sharing	Model convergence is slow; complex deployment; evaluation difficulties	[26, 31]

Table 3.1 presents a comparison of AR-based try-on approaches identified in the literature. The table consolidates typical technology stacks, strengths, and limitations for each category of solution, ranging from frontend-centric mobile AR and multi-camera kiosks to backend API pipelines and federated learning variants. For instance, frontend-centric mobile AR systems (e.g., ARKit, ARCore, OpenCV) are praised for their low latency and offline usability but remain highly device-dependent and fragmented across platforms. In contrast, backend API pipelines

(such as the approach developed in this thesis) centralise detection and overlay logic via YOLOv8, MediaPipe, and OpenCV, offering scalability and reusability across clients, though at the cost of requiring reliable network connectivity. The inclusion of prediction-serving frameworks (e.g., TF-Serving, Clipper), hybrid edge-cloud offloading, and federated learning highlights emerging architectures that address operational scalability, latency reduction, and privacy concerns. Overall, Table 3.1 underscores the trade-offs between user experience, system scalability, hardware cost, and integration flexibility, setting the stage for the backend-centric API solution proposed in this work.

3.5 Discussion Synthesis: Strengths, Limitations, and Impact

Notable Strengths. AR enhances customer engagement and confidence in making purchases; modern trackers allow for real-time alignment; server-side pipelines lighten workload and improve long-term maintainability.
Notable Limitations. Brand-siloed frontend-only systems lack scalability, and volumetric multi-camera rigs are not practical. Mobile-only pipelines are worsened by occlusions and lighting, and lack the ability to centralise updates. Other studies are missing a reproducible evaluation of latency and frame-per-second reporting.
Thesis Implications. Cross-platform AR try-on interfaces are achievable using backend-first strategies. Cross-platform retail AR try-on systems can be constructed using backend-first strategies. Detection, landmarks, overlay, and a stable REST surface provide vacation for terrain, freeing retailers to plug in via simple HTTP endpoints. Retailers integrate via simple HTTP endpoints, enabling backend-driven model evolution.

3.6 Research Gaps and Future Directions

Gap 1: Absence of Backend API Investigations. Most AR try-on systems are frontend-focused or hardware-dominated[2, 3]. Few investigate API-based designs that separate the user interface from the computation, as in our case. With the movement towards microservices and backend for frontend (BFF) models in website architecture, the value of UI decoupling computation, as it is done in AR, is promising but lacks attention in AR retail
Gap 2: Absence of Comprehensive Scalability Assessment. Although a large number of systems are proposed, very few focus on defining the architecture's performance metrics .
Gap 3: Violations of Users' Privacy and Security. The offloading of user images or a video stream presents a dual challenge in technical and regulatory privacy concerns [32].

Gap 4: Lack of Standardised Interfaces. API standardisation (e.g., OpenAPI, GraphQL) could facilitate cross-vendor integration, but there is no common schema for AR services that defines frame submission, authentication, or response formats across platforms [29, 15].
Future Directions. They all remain constructive and actionable, and I elaborate on them below:
- AR try-on API contracts should be formalised, for example, OpenAPI for frames and endpoints.
- Perform testing in separate mobile and network configurations to compare backend and frontend latency.
- Look into the use of hybrid edge/cloud offload in conjunction with adaptive offloading.
- Incorporate methods of privacy protection, such as differential privacy or encrypted feature transfer.
- Look into cross-market differences in AR user trust and adoption from a cultural lens.

3.7 Conclusion

Evidence suggests AR try-on has strong user-experience value, but persistent architectural gaps *backend-centric* scalable APIs for consumption by numerous clients. With modern detection and landmarking, server-side try-on is possible; the ML serving literature provides the building blocks for dependable deployment. This thesis seeks to address the gap by specifying and showcasing an integration-first AR try-on API with a thin demo client. Design decisions are guided by evidence from AR/HCI and backend ML serving research.

Chapter 4

System Design

The reasoning behind the implementation of the AR-based virtual try-on **backend API** is discussed in this chapter. This includes architectural components of the system, responsibilities of the components, and the control and data flows as they pertain to the API. The frontend part of the system is a *demo client* built in React to illustrate integration; however, the REST API is the focus. Retailers are provided with a ready-to-use integration REST API which they can connect to their proprietary applications. The design is linked with best practices outlined in previous chapters—stateless REST design, modular machine learning services, and separation of concerns (cf. [33]).

4.1 Architecture Overview

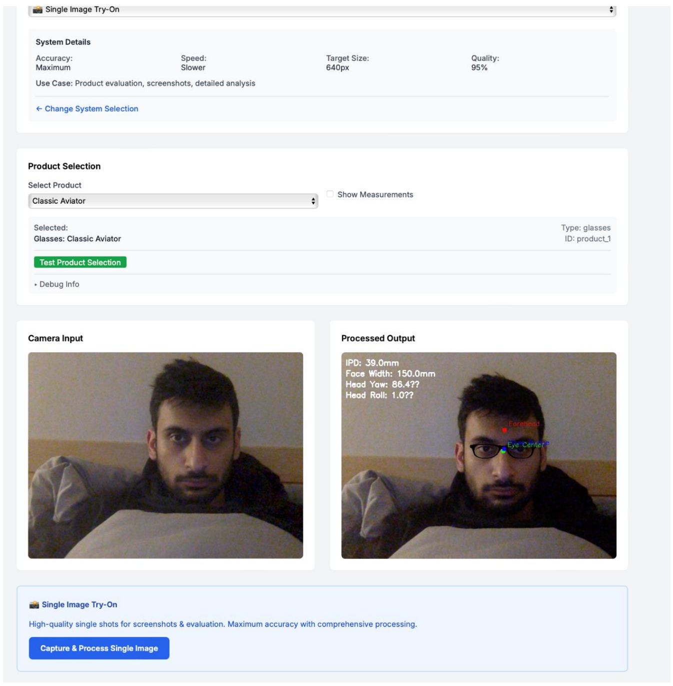
The structure includes three elements:
Detection Layer: Used for face detection through YOLO and precise landmark estimation through MediaPipe Face Mesh. MediaPipe provides extra face landmark estimation while YOLO ensures robust detection.
Dimensional Matching Layer: Overlays (e.g., glasses) matching the user's face, ensures realistic pixel scaling and maintins the biological reference value. Uses IPD as a biological reference constant to convert real world measurements in mm to pixel values.
Overlay Composition Layer: In OpenCV this module performs integration of the user's image or video frames with PNG assets to for scaling, translation, and rotation. Further, it performs blending of the assets to create a single composite.

4.1.1 Scope and Boundaries

This project intentionally lacks a database layer. In practice, product metadata and digital assets such as clothing or accessory PNG overlays would exist in a retailer's database or a content delivery network (CDN). The AR Try-On API is designed to be *stateless*: each request is self-contained (e.g., frame/image, productId), and the backend executes necessary operations and returns results without retaining data. This design approach prevents the API from becoming overly complex and helps maintain the desired lightweight, modular structure, which is ready for integration, while avoiding redundant storage functions that the retailer's infrastructure already provides.

4.2 Modes of Operation

The system has 3 distinct modes as referenced in Figure 4.1
Single Image Try-On: The user is prompted to upload a static image. The system applies detection, scales the image dimension, and processes the image by overlaying the accessories. This mode is very visual and aims for the highest accuracy.
Real Time Stream (WebSocket): The client is able to send continuous frames to the server. Compared to REST calls, WebSocket communication has much less latency and results in a near real time responsiveness. This mode has the ability to change the balance of accuracy and speed due to quality adjustment.
High Accuracy Stream: All detection methods (YOLO + MediaPipe) run concurrently at a higher resolution with measurement and overlaying style frames. Although this mode is much slower, it gives maximum alignment fidelity. This mode is intended for professional use, or product evaluation.


Single Image Try-On

System Details

Accuracy: Maximum
Speed: Slower
Target Size: 640px
Quality: 95%
Use Case: Product evaluation, screenshots, detailed analysis

← Change System Selection

Product Selection

Select Product: Classic Aviator
Show Measurements

Selected:
Glasses: Classic Aviator
Type: glasses
ID: product_1

Test Product Selection

▸ Debug Info

Camera Input

Processed Output

IPD: 39.0mm
Face Width: 150.0mm
Head Yaw: 86.4??
Head Roll: 1.0??

Single Image Try-On
High-quality single shots for screenshots & evaluation. Maximum accuracy with comprehensive processing.

Capture & Process Single Image

Figure 4.1: Three Modes of Operation in the system

4.3 Contextual View

A C4 Context diagram illustrates the relationship between the user, client, backend, and ML engines.

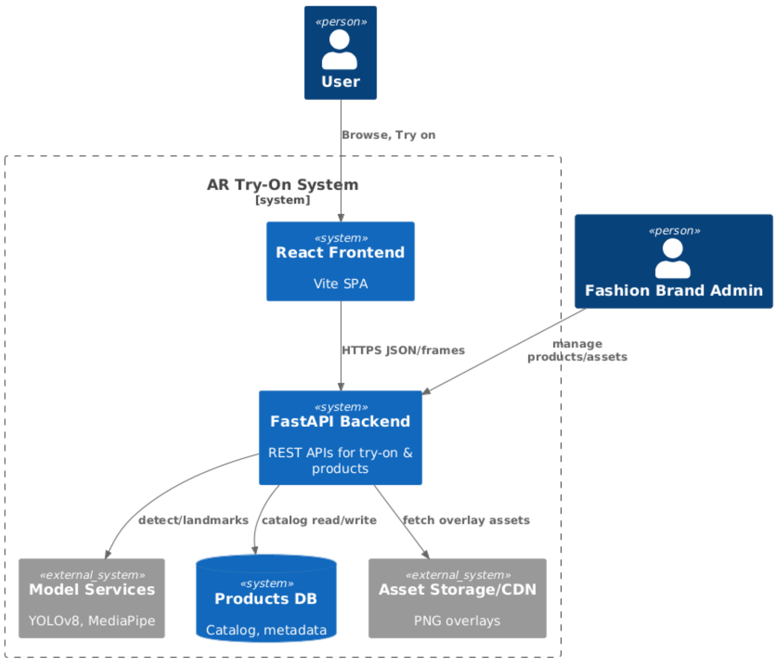


Figure 4.2: C4 Context Diagram of the AR Try-On System.

Figure 4.2 highlights the interaction between the user, the React demo client, the FastAPI backend, and the ML engines (YOLOv8, MediaPipe, OpenCV). The demo client communicates with the backend using stateless REST calls, while the backend delegates heavy processing to the ML modules. This abstraction emphasises that the backend API is integration-ready, allowing third-party retailers to plug in their own frontends while reusing the same ML pipeline.

4.4 Key Interactions (UML)

4.4.1 Sequence Diagram

Live Try-On

The workflow of a live try-on session is best captured with a sequence diagram, which details the order of requests and responses between the user, frontend, backend, and ML components.

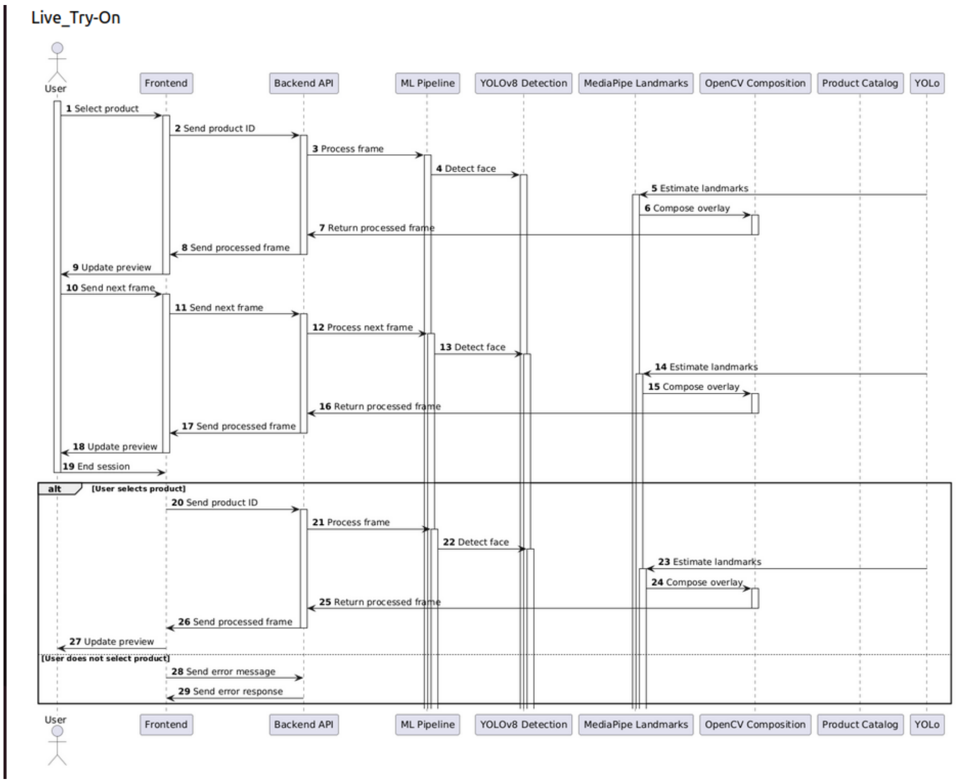


Figure 4.3: Sequence diagram for live try-on request processing.

Figure 4.3 illustrates the sequence of interactions during a live try-on session. After the user selects a product, the frontend repeatedly sends video frames with the associated product ID to the FastAPI backend. The backend processes each frame by calling YOLOv8 for detection, MediaPipe for landmarks, and OpenCV for overlay placement. The processed frame is returned to the frontend, which updates the preview in real time. This loop continues until the session ends, demonstrating the stateless but continuous nature of the interaction.

Overlay Placement

The workflow of overlay placement during a try-on request is best represented using a sequence diagram, which illustrates the conditional logic for handling detected and non-detected faces, as well as the interactions between the user, frontend, backend, and supporting ML components.

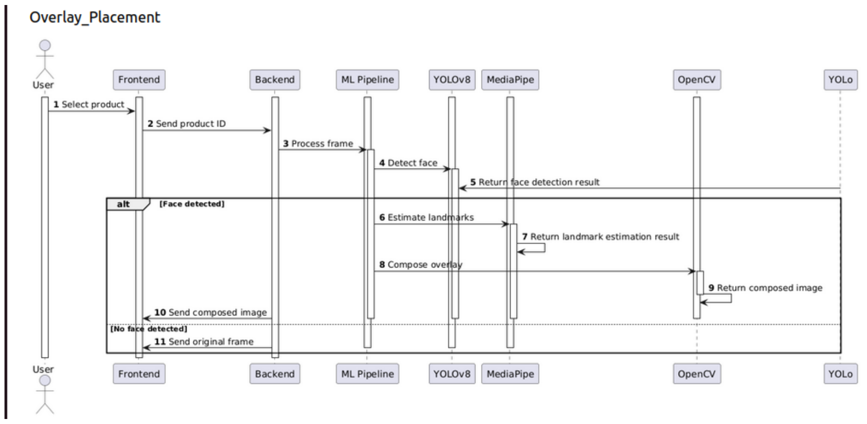


Figure 4.4: Sequence diagram for overlay placement during frame processing.

Figure 4.4 illustrates the sequence of interactions for overlay placement. After the user selects a product, the frontend transmits the frame and product ID to the backend. The backend first invokes YOLOv8 for face detection; if a face is detected with sufficient confidence, MediaPipe estimates landmarks, and OpenCV composes the overlay onto the frame. The composed image is then returned to the frontend for preview. If no face is detected, the backend bypasses overlay placement and simply returns the original frame. This flow captures both the normal and fallback scenarios, ensuring robustness in real-time processing.

Product Catalogue Browsing and Filtering

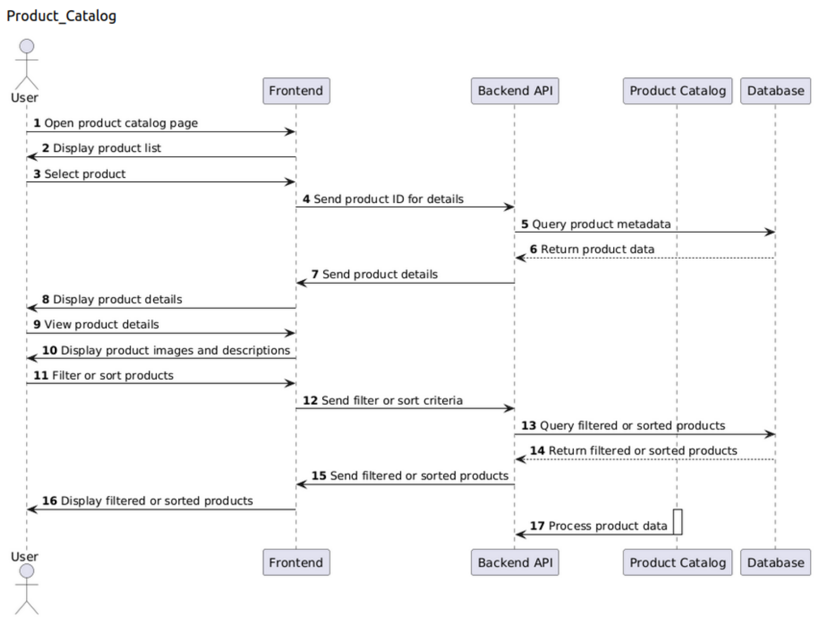

Product_Catalog
User
Frontend
Backend API
Product Catalog
Database
1 Open product catalog page
2 Display product list
3 Select product
4 Send product ID for details
5 Query product metadata
6 Return product data
7 Send product details
8 Display product details
9 View product details
10 Display product images and descriptions
11 Filter or sort products
12 Send filter or sort criteria
13 Query filtered or sorted products
14 Return filtered or sorted products
15 Send filtered or sorted products
16 Display filtered or sorted products
17 Process product data

Figure 4.5: Sequence diagram for product catalogue browsing and filtering.

Figure 4.5 illustrates the catalogue interactions. When the user opens the catalogue page, the frontend displays the initial product list and allows the user to select a product. The frontend sends the product ID to the backend API, which returns product details from a demo list or retailer-provided API. When filters or sort options are applied, the frontend forwards the criteria to the backend, which returns the relevant results for display.

Demo Client

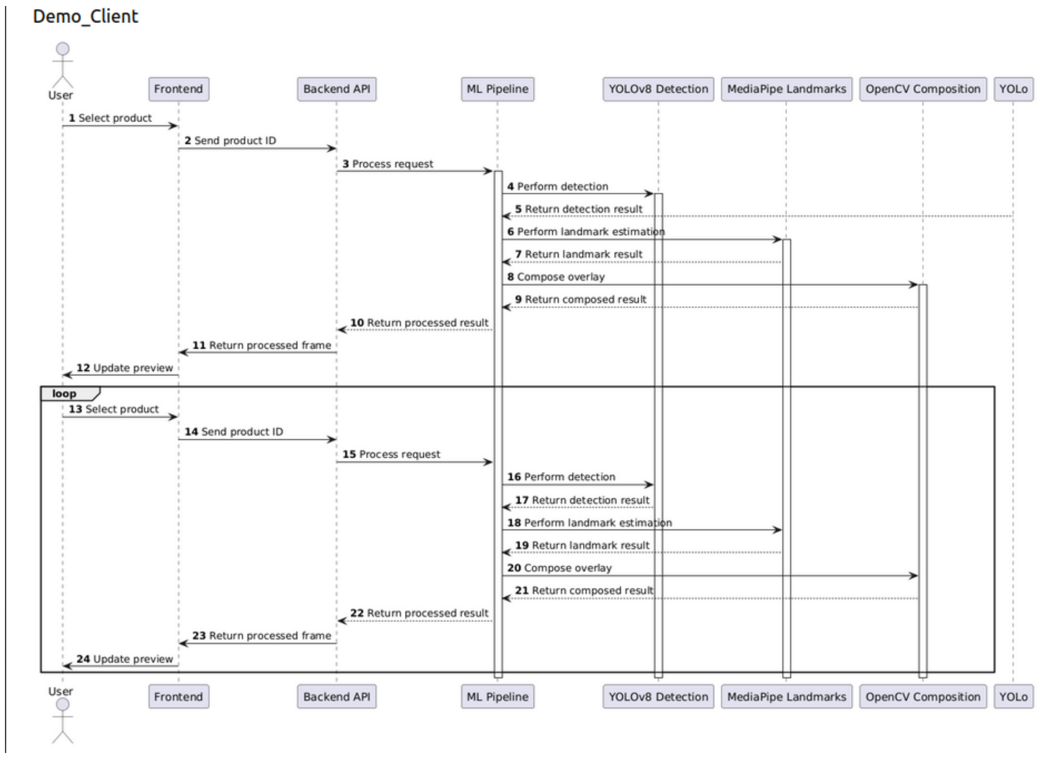


Figure 4.6: Sequence diagram for Demo Client.

Figure 4.6 shows the user's actions with the demo client's frontend and the interactions with the backend API and the machine learning pipeline in the backend. The flow starts when a user selects a product, and the frontend sends the relevant product ID to the backend. The backend starts a backend-frontend intercommunication by activating the request-processing pipeline and invoking some ML tasks. The face detection tasks are executed by the YOLOv8 Detection module (steps 4-5), and subsequently, face keypoints are computed by MediaPipe Landmarks (steps 6-7). The computed landmarks together with the relevant transformation parameters are applied by OpenCV Composition to generate the overlay, and the face is composed (steps 8-9). After processing the frame, the backend begins responding in step 10 by sending the frame to the frontend, which in step 12 displays it to the user, thus giving the impression that the feedback is instantaneous.

Backend API

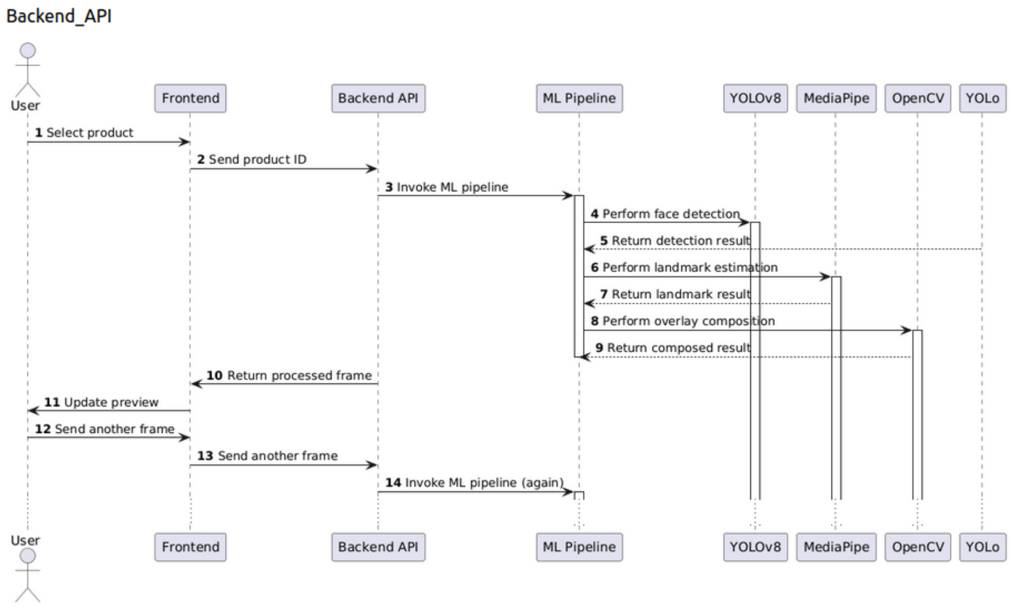


Figure 4.7: Sequence diagram Backend API orchestration.

Figure 4.7 shows the backend API's essential function in orchestrating the full AR try-on workflow. Once the user picks a product (that's step 1), the frontend sends the product ID over to the backend API (this is step 2). Following that, the backend activates the machine learning (ML) pipeline, orchestrating tasks among different modules.

ML Pipeline

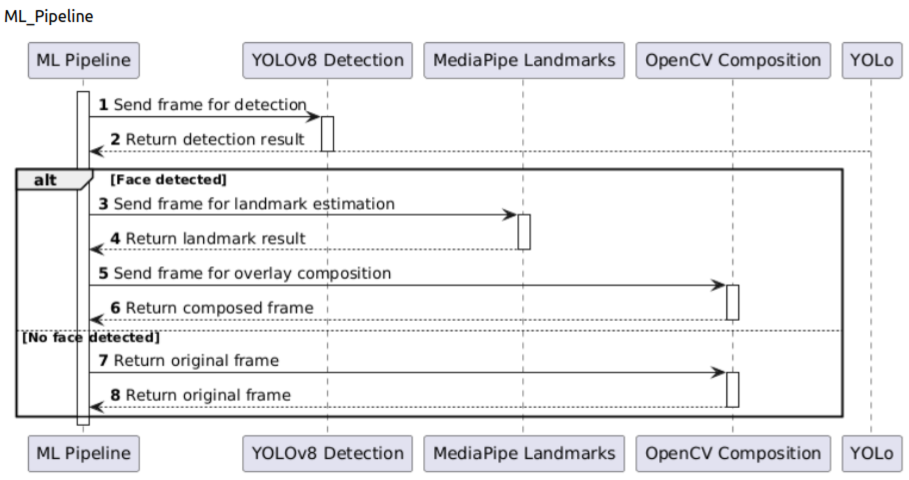


Figure 4.8: Sequence diagram for ML pipeline processing.

Figure 4.8 shows the internal ML pipeline workflow. It integrates the core ML technology modules for face detection, landmark estimation, and overlay rendering. The workflow receives a frame for processing as the starting point. The YOLOv8 Detection module (Step 1–2) processes the frame and checks for the presence of a face; If the face is absent, the unchanged frame is output (Steps 7–8), ensuring no excessive processing. Once a face is confirmed, the frame undergoes further processing. The system performs landmark estimation with MediaPipe (Steps 3–4) that identifies important facial keypoints to ensure accurate alignment. The frame undergoes the OpenCV Composition stage (Steps 5–6), where the overlays (like virtual eyeglasses and hats) are placed based on the computed frameworks as per the detected landmarks transformations. The frame is then sent to the frontend, ensuring real-time delivery.

YOLOv8 Detection

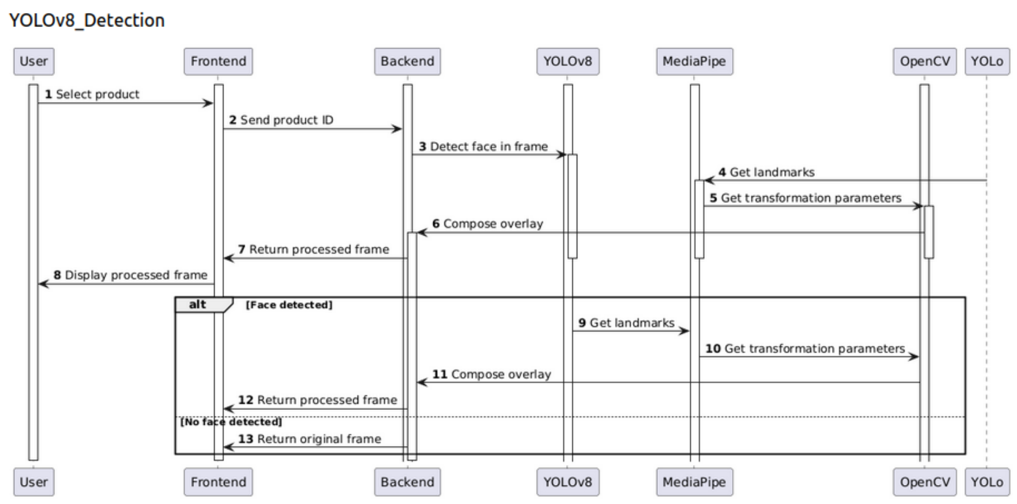


Figure 4.9: Sequence diagram for YOLOv8-based face detection.

In the AR try-on system, the YOLOv8 detection workflow is illustrated in Figure 4.9. User interaction starts by choosing a specific product, which triggers the frontend to send the product ID to the backend. The backend processes the request by invoking YOLOv8 to perform face detection on the provided frame. After a face is found, the backend face detection starts to estimate the corresponding facial landmarks and computes the transformation parameters which are needed to align and overlay the product on the user's face. These computations are performed in parallel, with the composition taking place in OpenCV, which merges the virtual product and the face in the composition step. The merged frame is sent back to the frontend and rendered for the user in real-time. The diagram shows alternative paths in the process as well. If a face is found, the process proceeds to computing the landmarks and overlay, as described in steps 9-11. Face detection can also skip composition and return the frame with no processing, which is described in steps 12-13.

MediaPipe Landmarks

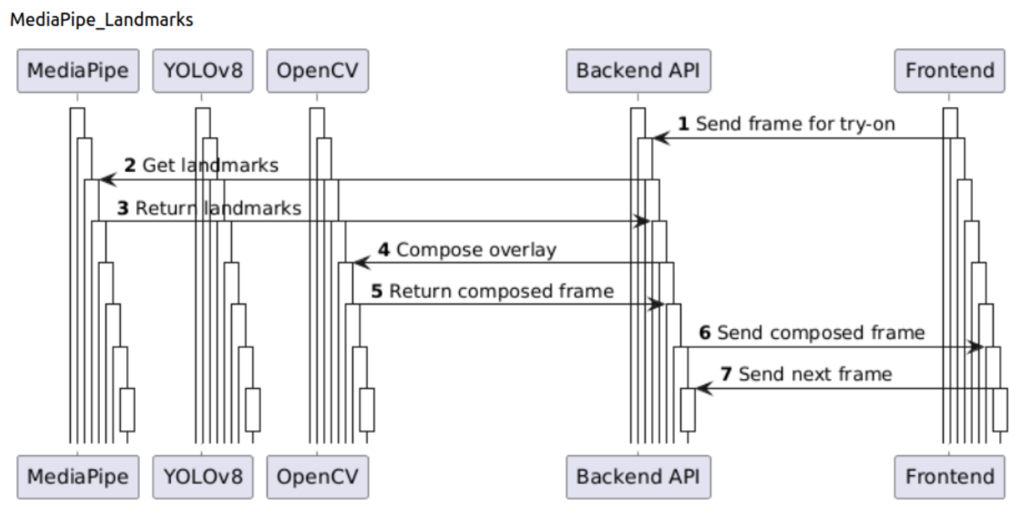


Figure 4.10: Sequence diagram for MediaPipe Landmarks.

As Figure 4.10 depicts, the frontend submits frames for try-on to the backend API, which offloads the landmark extraction to MediaPipe. The results get handed off to OpenCV to perform overlay composition. After that, the backend receives the composed frame and the backend sends the frame to the frontend, all the while guaranteeing real-time preview consistency.

OpenCV Composition

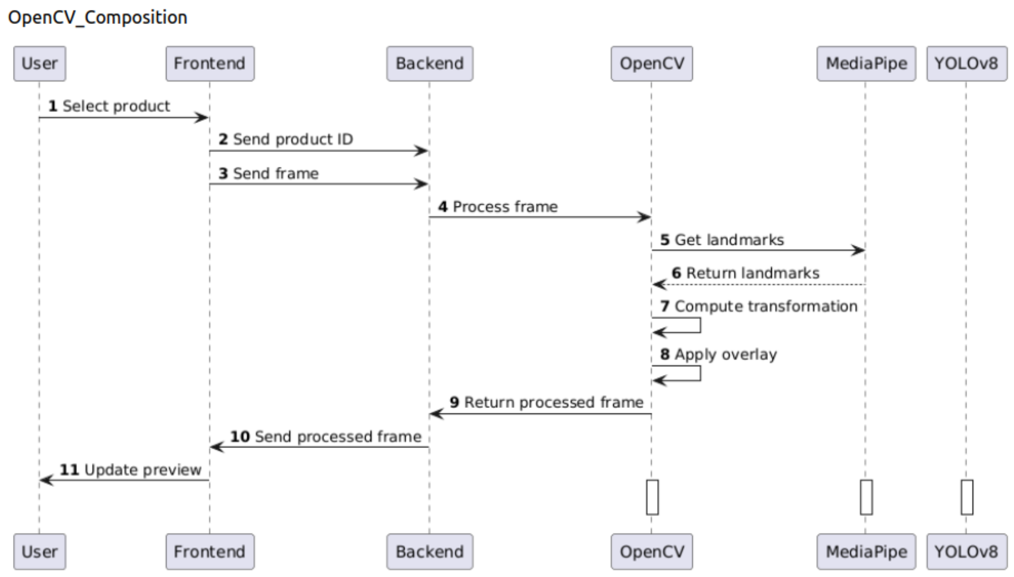


Figure 4.11: Sequence diagram for demo client interaction.

As shown in Figure 4.11, after the user picks a product, the frame and product ID get sent to the backend. The backend sends a command to OpenCV to execute the transformations, fetch the landmarks from MediaPipe, and perform overlay alignment. As soon as the frame is sent back to the frontend, real-time visualisation makes interaction easier. The user can try on the product seamlessly.

4.4.2 Overlay Logic: Activity Diagram

Overall User Workflow

The overall user journey within the AR-based virtual try-on and shopping platform is illustrated using activity diagrams. The diagrams are shown in segments for readability.

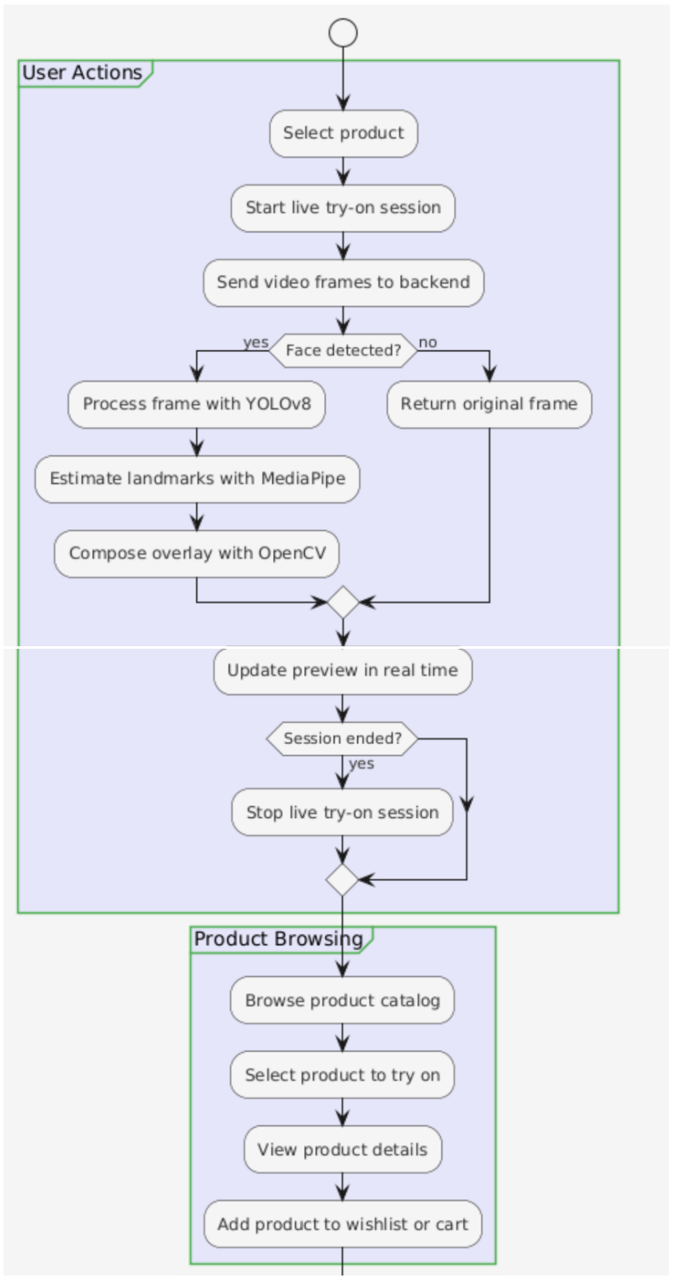

User Actions
Select product
Start live try-on session
Send video frames to backend
Face detected?
yes
no
Process frame with YOLOv8
Return original frame
Estimate landmarks with MediaPipe
Compose overlay with OpenCV
Update preview in real time
Session ended?
yes
Stop live try-on session
Product Browsing
Browse product catalog
Select product to try on
View product details
Add product to wishlist or cart

Figure 4.12: Activity diagram (segments 1-2) covering user actions and order product browsing.

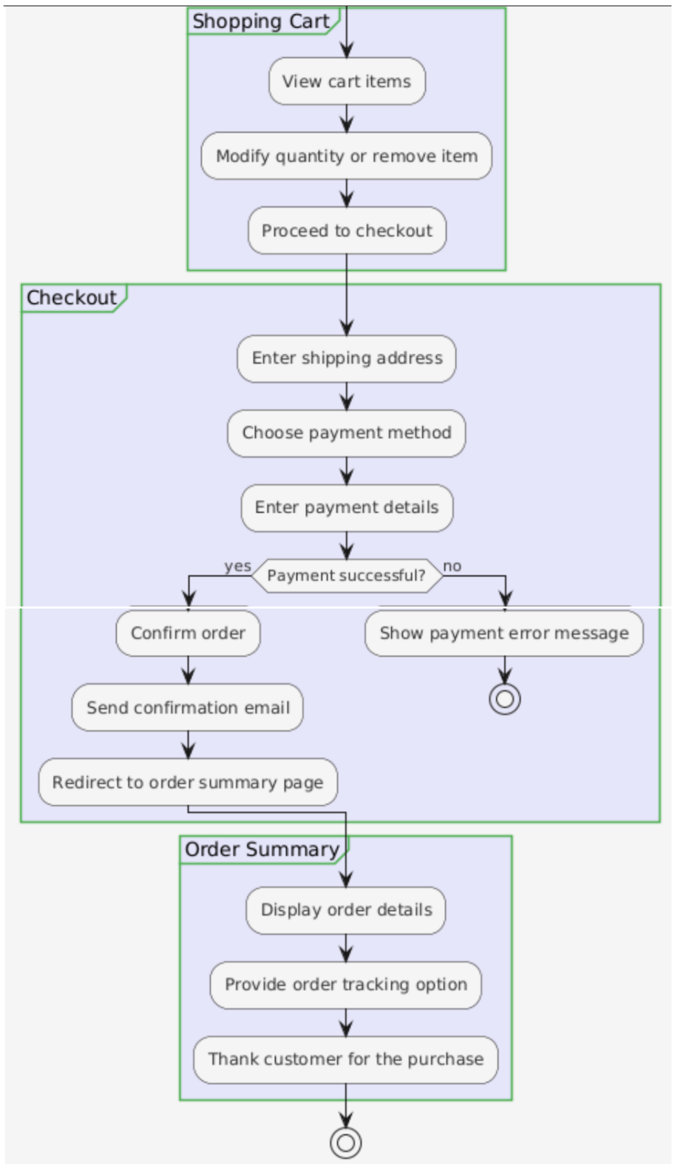


Figure 4.13: Activity diagram (segments 3–4) covering checkout and order confirmation.

Figure 4.12 and 4.13depicts the comprehensive flow of user activities within the AR-based try-on system. It starts with the user's face and actions. The user needs to select a product, start a live try-on, and then stream video frames to the server. Upon face detection, the system performs frame analysis with YOLOv8 detection, MediaPipe landmark estimation, and OpenCV overlay composition. The original frame is returned if no face is detected. This approach guarantees try-on robustness and responsiveness in real time. After the try-on, the user moves to browsing the product, which includes navigating the catalog, trying on selected items, and viewing product details. Users can also add items to a wishlist or to a shopping cart, which integrates the AR try-on with the rest of the e-commerce functionalities. Afterward, the user moves to the shopping cart and checkout area. At this point, the user can see the items in the cart, adjust quantities, or delete items. When finished with the shopping, the user moves to checkout, where shipping details, a preferred shipping option, payment method, and payment information are entered. A decision gateway evaluates if payment was done. If so, the system confirms the order; if not, an error message is shown. Ultimately, the final step in the procedure is the order summary stage that captures the customer's email, redirects them to a summary page, and displays all the order details. In addition, the customer is provided with a way to track their order. Finally, the customer is also thanked for their purchase, completing the full cycle of interaction.

Retailer and Backend Workflow

The retailer's workflow and backend processing are modelled with an activity diagram, which illustrates how retailers integrate with the backend API, configure their catalogues, and support live try-on requests through overlay placement and frame composition.

In detail, the figures,4.14, 4.15, and 4.16, illustrate the retailer and backend workflow. The retail side starts with the onboarding and configuration of the backend API, after which the product catalog can be managed.

With the user-initiated live try-on request, the backend processes the stream of frames using the full processing pipelines with YOLOv8 for face detection, MediaPipe for landmark estimation, and OpenCV for overlay composition. The frontend preview receives the frame immediately after processing for real-time display. This loop is performed for the next frames so that the system can update everything continuously during the try-on.

The diagram as a whole demonstrates the integration of catalog management, request handling, overlay rendering, and other processes in one workflow done effortlessly.

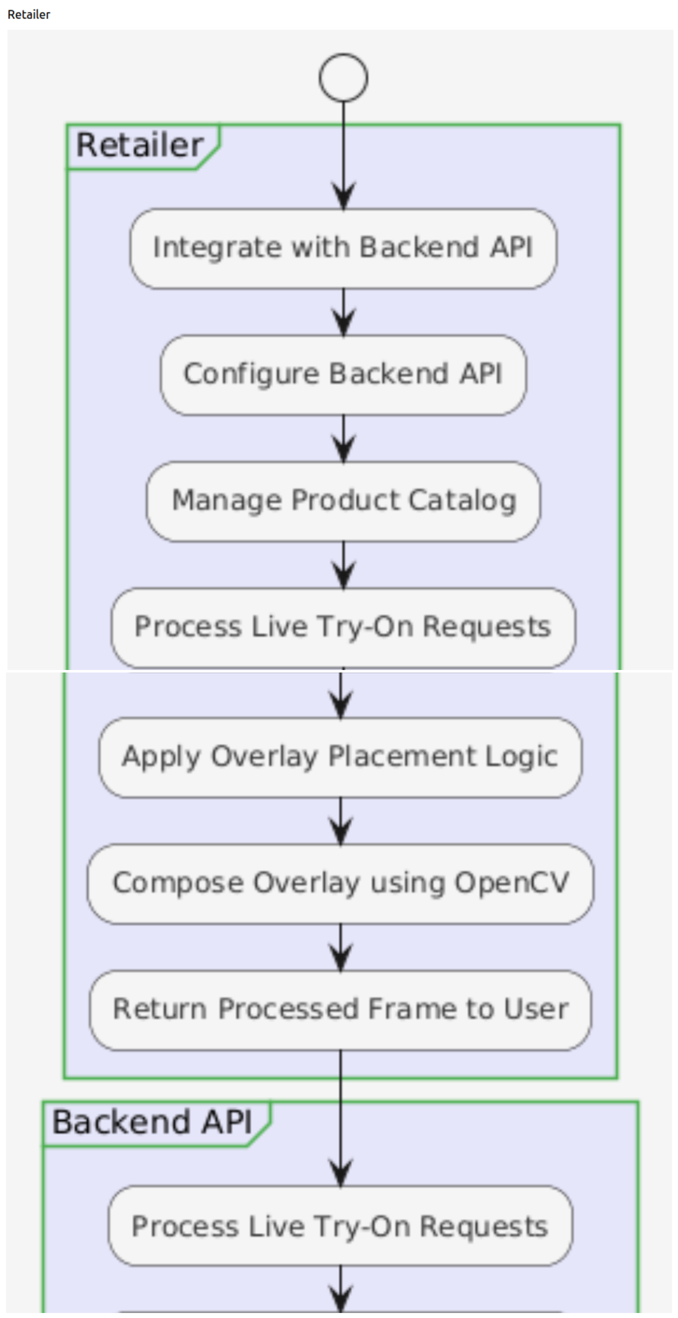

Retailer
Retailer
Integrate with Backend API
Configure Backend API
Manage Product Catalog
Process Live Try-On Requests
Apply Overlay Placement Logic
Compose Overlay using OpenCV
Return Processed Frame to User
Backend API
Process Live Try-On Requests

Figure 4.14: Retailer and backend activity (segments 1–2): integration and catalogue configuration.

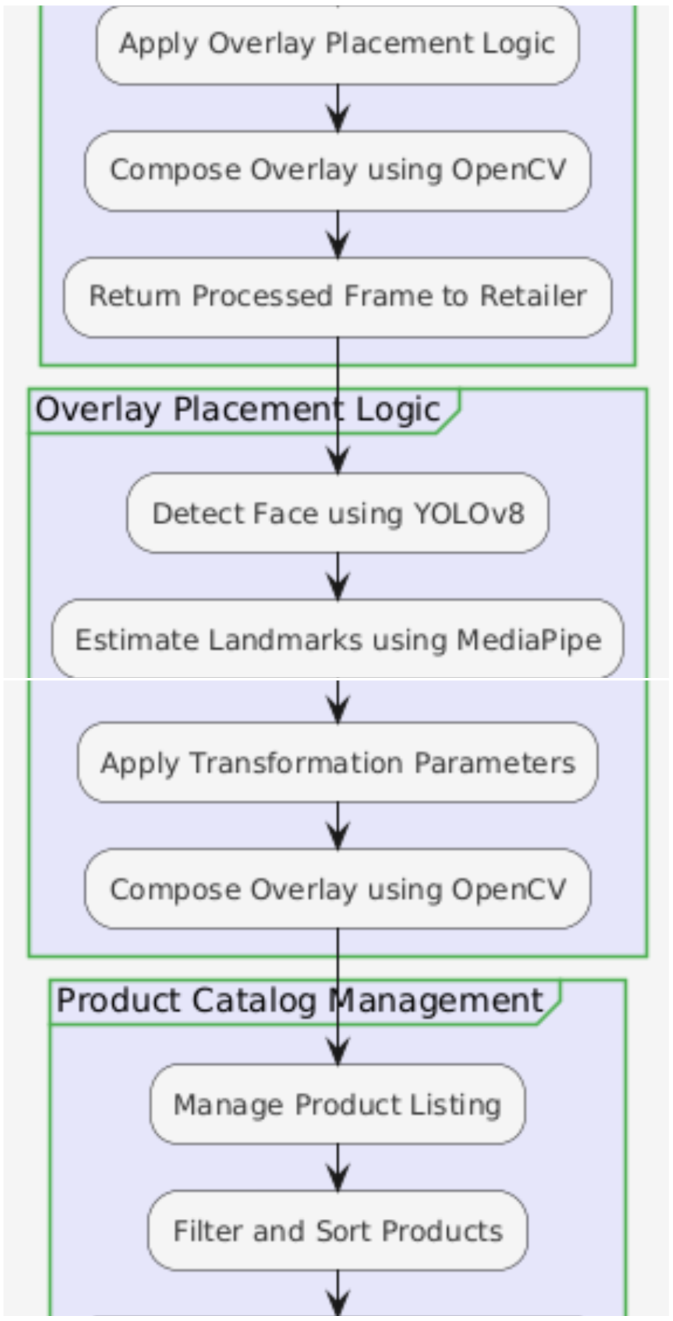

Apply Overlay Placement Logic
Compose Overlay using OpenCV
Return Processed Frame to Retailer
Overlay Placement Logic
Detect Face using YOLOv8
Estimate Landmarks using MediaPipe
Apply Transformation Parameters
Compose Overlay using OpenCV
Product Catalog Management
Manage Product Listing
Filter and Sort Products

Figure 4.15: Retailer and backend activity (segments 3–4): handling try-on requests.

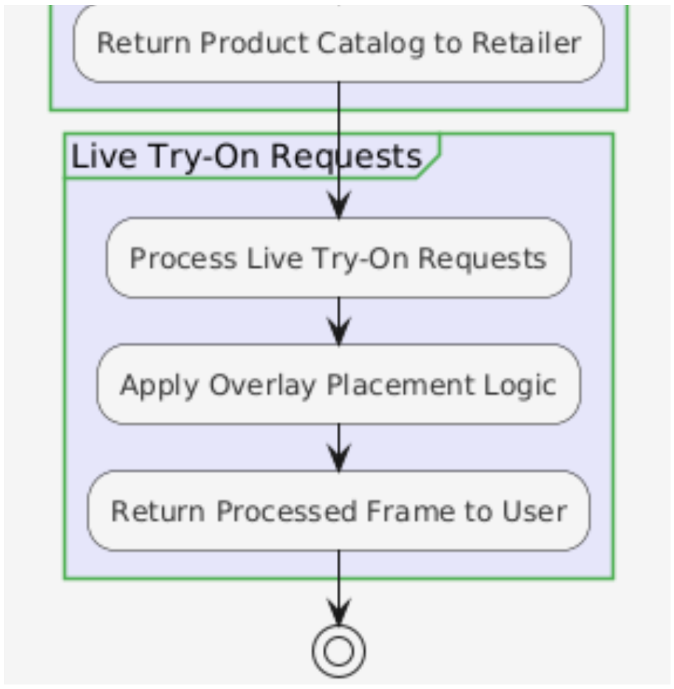

Return Product Catalog to Retailer
Live Try-On Requests
Process Live Try-On Requests
Apply Overlay Placement Logic
Return Processed Frame to User

Figure 4.16: Retailer and backend activity (segment 5): overlay composition and response.

4.4.3 System Use Case

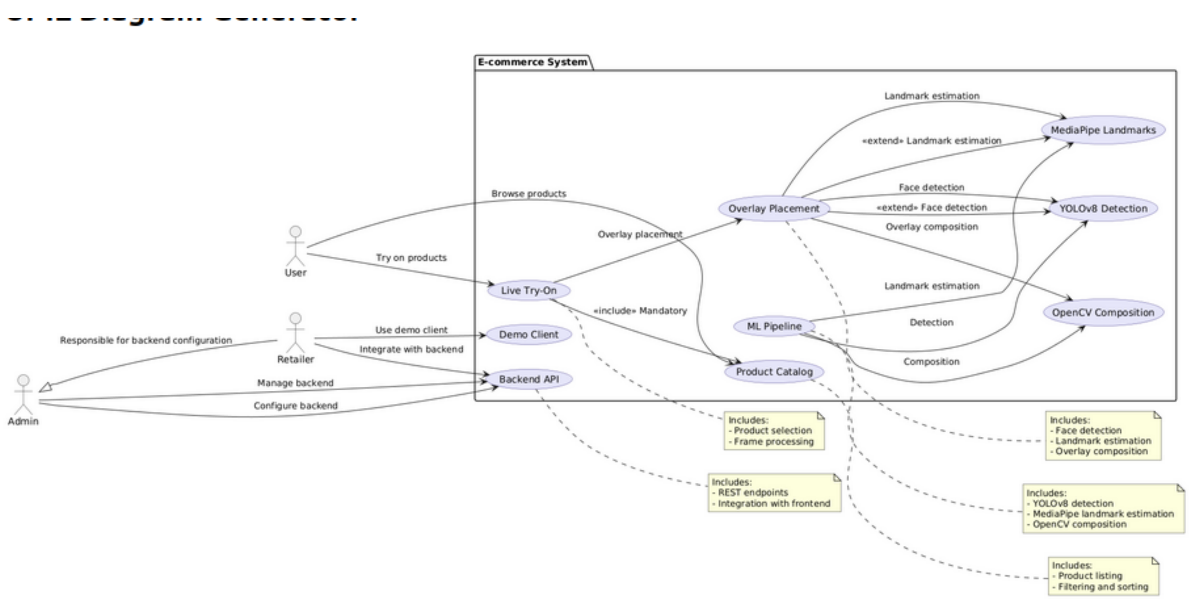


Figure 4.17: UML use case diagram for the AR-based virtual try-on platform.

Figure 4.17 summarises the main roles and system capabilities. Users browse products and initiate *Live Try-On*; retailers integrate with and configure the *Backend API* and manage the *Product Catalogue*. Within the system, *Overlay Placement* composes try-on results by invoking ML subfunctions—face detection (YOLOv8), landmark estimation (MediaPipe), and image composition (OpenCV)—exposed through the *ML Pipeline* and consumed by the *Backend API/Demo Client*.

4.5 API Surface (REST)

The API is purposely minimal for easy adoption by existing apps (web or mobile). Table 4.1 lists the endpoints used by the demo client and intended for integrators. No storage/admin endpoints are provided.

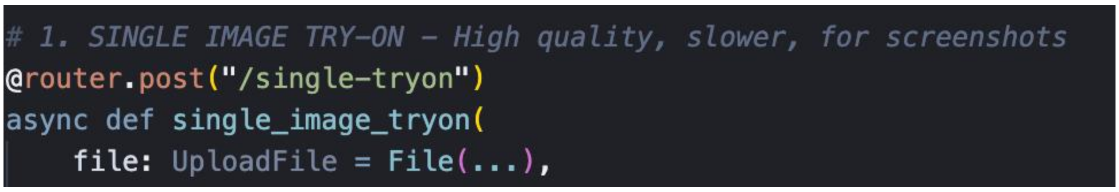
Table 4.1: Core API Endpoints for Different Try-On Modes

Method & Path	**Purpose**	**Notes**
`POST /single-tryon`	Single image processing	High-quality output, optimized for screenshots and evaluation. Uses YOLO + MediaPipe fusion with IPD scaling.
`WS /websocket-tryon`	Real-time streaming	Fast, adaptive quality for continuous live try-on. Balances accuracy and speed dynamically.
`POST /tryon`	High-accuracy processing	Maximum accuracy using full YOLO + MediaPipe + IPD pipeline. Slower; suitable for professional analysis and product imagery.

4.6 Algorithm and Code Snippets

This section provides pseudocode for the updated three–tier pipeline and the new dimensional accuracy module, along with representative FastAPI endpoints (REST and WebSocket). The algorithms reflect the dual detection fusion (MediaPipe + YOLO) and Interpupillary Distance (IPD)–based scaling used to convert real-world product dimensions (mm) to image pixels.

4.6.1 Backend Snippet (FastAPI: single image, REST)

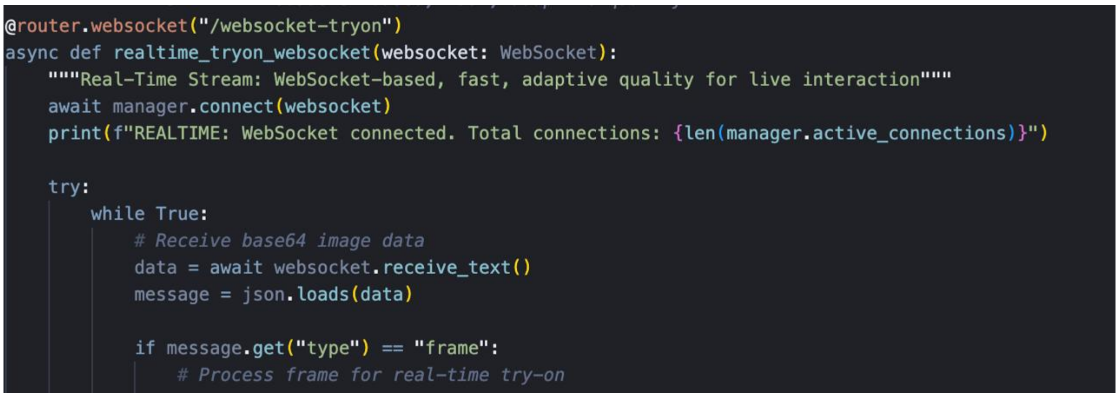


```
# 1. SINGLE IMAGE TRY-ON - High quality, slower, for screenshots
@router.post("/single-tryon")
async def single_image_tryon(
    file: UploadFile = File(...),
```

Figure 4.18: Single Image Backend Snippet

The single image try-on endpoint (Figure 4.18) is intended for use cases that don't have strict time restraints, for these cases, the quality is of aforementioned importance, analyzing screenshots for alignment accuracy is one of the most notable use cases for this endpoint.
This is implemented by using FastAPI's `.post` decorator for the `/single-tryon` endpoint. The user uploads an image using the UploadFile interface, that returns a jpeg or PNG image. Upon receiving the image, the backend runs a detector that captures faces using YOLOv8 and then detects face mesh landmarks using MediaPipe Face Mesh. There is a robust fallback pipeline for the event that the YOLO face detector fails.
Pixles of the landmarks that are detected are scaled to the biological constant (average biological inter pupillary distance is 63 mm) constant to obtain the scaled biological product dimensions (glasses, hats etc) that are chosen. The IPD which is computed is then scaled by a multiplier to account, and compensate, for perspective, and the distance of the user, to the camera, for realism.
OpenCV completes the final augmented image, that is sent to the user in a streaming out JPEG format, by the user face with the product seamlessly mixed, to the user, with the product image. The computed dimensions of the product image, which is resized to match the dimensions, are then translated, and geometrically transformed by the user's face, and then rotated, to the final product.
Each request in this endpoint is aligned most accurately, using IPD scaling and dual YOLO + MediaPipe validation for alignment, validation. The speed of processing for alignment requests individually is much worse than when aligned using the streaming pipeline, as requests are processed in a which is then aligned to the streaming pipeline.
For those best fit when responsiveness is not as pivotal as visual quality.

4.6.2 Backend Snippet (FastAPI: real-time, WebSocket)



```python
@router.websocket("/websocket-tryon")
async def realtime_tryon_websocket(websocket: WebSocket):
    """Real-Time Stream: WebSocket-based, fast, adaptive quality for live interaction"""
    await manager.connect(websocket)
    print(f"REALTIME: WebSocket connected. Total connections: {len(manager.active_connections)}")

    try:
        while True:
            # Receive base64 image data
            data = await websocket.receive_text()
            message = json.loads(data)

            if message.get("type") == "frame":
                # Process frame for real-time try-on
```

Figure 4.19: Backend implementation of the WebSocket-based real-time try-on service

The real-time try-on for WebSocket usage is shown in Figure 4.19. which allows for endless live communications with changing qualities. Also, in contrast to the single-image REST endpoint, this endpoint allows for a continuous bi-directional connection between a client and a backend.

- **Connection Handling:** The endpoint is reached with `@router.websocket("/websocket-tryon")` decorator and the connection becomes a WebSocket connection. The connection manager allows the back-end of a client to log the individual socket sessions, which allows for multiple sessions to be opened at the same time, as shown in the log output `Total connections`.

- **Frame Reception:** Clients must keep their sessions open, and each client has a server that awaits the arrival of their base64-encoded image frames each of which is sent with `websocket.receive_text()`.

- **Message Parsing:** On the other hand, the backend has to proxy the frames of several AR, for wherever a try-on AR overlays, a streamline workflow is provided, with only a few exceptions on the edges of the frames.

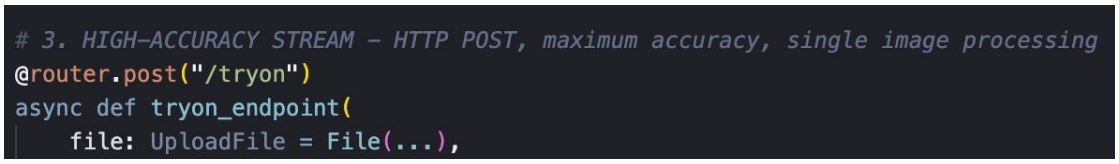
- **Pipeline Execution:** The backend applies OpenCV with other image manipulation to blend and send bounding and unbounding boxes with the scaled IPD.

- **Adaptive Quality:** The internal loop permits the dynamic framework to control the amount of frames per second and the accuracy of detection to be reacting in real time. This support smooth alterations despite changing bandwidth, but the overlay precision is reduced.

This streaming design is well applicable to interactive browsing, where fast and responsive streaming is more important than precision down to the individual pixel. But, as noted during the evaluation, the performance is quite network dependent, since every stream is sent to the backend, processed and returned. When the latency is high or bandwidth is low, a client will experience stuttering and frame drops, greatly disrupting the experience.

4.6.3 Backend Snippet (High-Accuracy Stream (REST Endpoint))

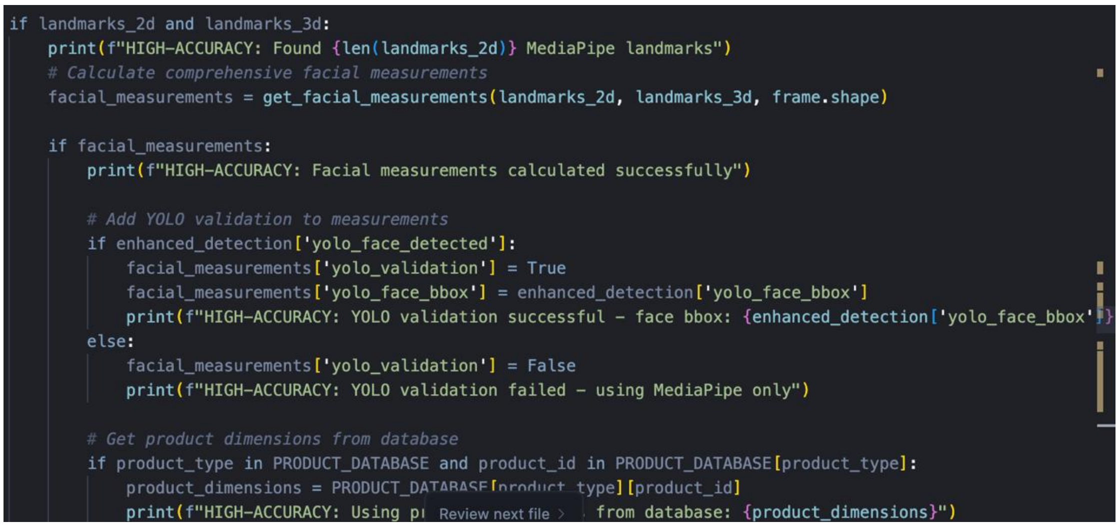
The third implementation mode of the AR Try-On API is the **High-Accuracy Stream**. Unlike the single-image endpoint and the WebSocket based real-time stream, this mode focuses on precision and accuracy the most even at the cost of speed. It is provisioned as a REST endpoint **(/tryon)**. It allows a single image to be sent through an *HTTP POST* and returns a dimensionless image. The endpoint executes the full fusion pipeline which includes – MediaPipe landmark detection, YOLO face validation, IPD based cross dimensional scaling, and product dimensional retrieval.



```
# 3. HIGH-ACCURACY STREAM - HTTP POST, maximum accuracy, single image processing
@router.post("/tryon")
async def tryon_endpoint(
    file: UploadFile = File(...),
```

Figure 4.20: High-Accuracy endpoint in FastAPI

As seen in the figure 4.20, the internal processing depicts how the systems features MediaPipe Face Mesh and YOLO detection to perform validation. First, MediaPipe outputs 468 2D and 3D Delaunay portraits on a user face and derives detailed facial tracking dimensions for image processing. It then uses a technique called the real-time face tracking validation step. If a person's face is enclosed within a capture box, the system computes the face's spatial coordinates and confidence against the landmark data. This dual detection strengthens systems robustness by embodiments of MediaPipe's limitations due to obstructions and lack of luminosity.



```python
if landmarks_2d and landmarks_3d:
    print(f"HIGH-ACCURACY: Found {len(landmarks_2d)} MediaPipe landmarks")
    # Calculate comprehensive facial measurements
    facial_measurements = get_facial_measurements(landmarks_2d, landmarks_3d, frame.shape)

    if facial_measurements:
        print(f"HIGH-ACCURACY: Facial measurements calculated successfully")

        # Add YOLO validation to measurements
        if enhanced_detection['yolo_face_detected']:
            facial_measurements['yolo_validation'] = True
            facial_measurements['yolo_face_bbox'] = enhanced_detection['yolo_face_bbox']
            print(f"HIGH-ACCURACY: YOLO validation successful - face bbox: {enhanced_detection['yolo_face_bbox']}
        else:
            facial_measurements['yolo_validation'] = False
            print(f"HIGH-ACCURACY: YOLO validation failed - using MediaPipe only")

        # Get product dimensions from database
        if product_type in PRODUCT_DATABASE and product_id in PRODUCT_DATABASE[product_type]:
            product_dimensions = PRODUCT_DATABASE[product_type][product_id]
            print(f"HIGH-ACCURACY: Using pı Review next file >  from database: {product_dimensions}")
```

Figure 4.21: Detailed facial measurements with YOLO validation

The pipeline obtains product dimensions, as illustrated in 4.21 (e.g., frame width, bridge width, and temple length) from a fixed product database and uses inter-pupillary distance (IPD) as the biological reference constant to dynamically convert the dimensions into pixel values to ensure overlays align with real-world dimensions. This ensures that the virtual glasses or hats are shown at real scale with respect to the user's head.
This high-accuracy mode, albeit expensive in resources, ensures the overlays are as close to the anticipated facial geometry as possible. It also has its trade-offs: each frame is captured, so the data has to be sent, processed, and returned over the network, resulting in high **latency** which is unacceptable for active browsing. It is designed for serious usage scenarios, such as advanced product assessment, catalog image creation, and high-definition screenshots.

4.6.4 Frontend Snippet (Frame Post)

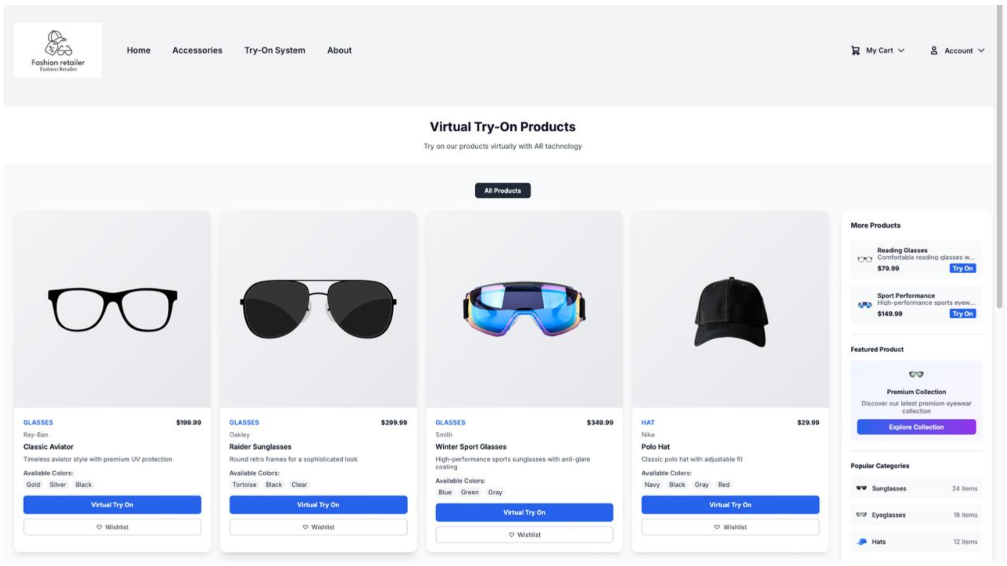
Listing 4.1: Posting frames from the demo client to the backend.

```javascript
async function sendFrame(blob, productId) {
  const form = new FormData();
  form.append("frame", blob, "frame.jpg");
  form.append("productId", productId);

  const res = await fetch(`${import.meta.env.VITE_API_BASE}/tryon/
     frame`, {
    method: "POST",
    body: form
  });

  const buf = await res.arrayBuffer();
  const url = URL.createObjectURL(new Blob([buf], { type: "image/
     jpeg" }));
  processedImg.src = url;
}
```

In Listing 4.1, we can see the client-side logic for submitting frames to the backend try-on service.
As for the function ‘sendFrame‘, it takes as input the video frame (a binary ‘blob‘) and the corresponding product id. Both the video frame and product ID are added to a ‘FormData‘ object which is sent to the specified backend API address using Fetch API with a POST request. Afterwards, the backend sends the frame bytes. The function then changes the Frame ArrayBuffer to Blob with image/jpeg type. It then temporarily generates a URL with ‘URL.createObjectURL‘ and assigns the ‘src‘ attribute of ‘processedImg‘ element to the URL, so the user can see the updated try-on preview. This snippet shows how the backend is sent captured frames to augment and how frames are returned and the user interface is updated without page refresh or additional libraries.

4.7 Screenshots and Diagrams in Thesis

In addition to UML diagrams, screenshots of the demo client are included to illustrate the user interface.



Virtual Try-On Products
Try on our products virtually with AR technology

GLASSES $199.99
Ray-Ban
Classic Aviator
Timeless aviator style with premium UV protection
Available Colors: Gold Silver Black

GLASSES $299.99
Oakley
Raider Sunglasses
Round retro frames for a sophisticated look
Available Colors: Tortoise Black Clear

GLASSES $349.99
Smith
Winter Sport Glasses
High-performance sports sunglasses with anti-glare coating
Available Colors: Blue Green Gray

HAT $29.99
Nike
Polo Hat
Classic polo hat with adjustable fit
Available Colors: Navy Black Gray Red

More Products
Reading Glasses
Comfortable reading glasses w...
$79.99
Sport Performance
High-performance sports eyew...
$149.99

Featured Product
Premium Collection
Discover our latest premium eyewear collection

Popular Categories
Sunglasses 24 items
Eyeglasses 18 items
Hats 12 items

Figure 4.22: Frontend UI of the system

The Figure 4.22 shows the components that are a part of Virtual Try-on system. In demo clients using React Vite with Tailwind CSS, the figure showcases a product catalogue page. This design imitates an e-commerce model and allows users to check product listings, incorporated with a unique functionality of *Virtual*

Try-On.
In the middle of the page, a scheduling of product cards is presented. Each card holds the product's image, category (Glass, Hat, etc.), brand, product title, price, colors available, and two main action buttons. The first button, **Virtual Try On**, initiates a request to the remote back-end API (two options available: `/tryon/image` and `/tryon/frame`) with the user's video frame/image and the product identifier (ID). The backend system uses YOLOv8 to detect the frame, then video frames processing (MediaPipe to detect and estimate landmarks, and OpenCV to blend overlays) before sending the client the blended output for display purpose. Users can also save the product to a wishlist using this button.
To the right, a sidebar features *More Products* shortcuts with quick "Try On" actions, promotes a *Featured Product*, and contains a list of *Popular Categories* with product counts (e.g., *Sunglasses*, *Eyeglasses*, *Hats*). This approach reinforces the structure of an actual retail site by displaying associated products and offering navigation pathways.
In the dataframe, as the top is the branding "Fashion Retailer", it has all the links as *Home*, *Accessories*, *Try-On System*, and *About*. There are also *My Cart* and *Account* other regular e-commerce services.
This UI design shows the backend API being integrated into a real life shopping experience, without any friction. Users can freely explore the catalogue, and the "Virtual Try On" functionality makes the system stand out from other product browsing applications, as it facilitates AR product evaluation during the shopping process.

4.8 Rationale and Best-Practice Links

The design is intentionally *integration-first*: a stateless REST API with a minimal, decoupled client that demonstrates usage. This aligns with best practice for scalable and evolvable ML-enabled web systems: isolate model concerns behind clean endpoints, avoid hidden state, and keep adoption friction low for third-party consumers (cf. [33]). The combination of architecture, context, sequence, and activity diagrams provides a comprehensive view of both static structure and dynamic behaviour.

Chapter 5

System Evaluation

This chapter attempts to assess the performance of the AR-based try-on API against the criteria set earlier in the thesis. The evaluation includes functional performance detection, overlay accuracy, interaction metrics, integration feasibility, and physical experience limitations. The results are integrated with existing literature on try-on AR systems and the backend-serving infrastructures.

Table 5.1: Comparison of Try-On Methods Implemented in the Backend API

Method	**Accuracy**	**Speed**	**Use Case**	**Detection Methods**
Single Image (REST)	High	Medium	Screenshots, evaluation	Optimized pipeline (YOLO + MediaPipe fusion with IPD scaling)
Real-Time (WebSocket)	Adaptive	Fast	Live try-on, continuous interaction	Adaptive detection (YOLO + optional MediaPipe landmarks)
High-Accuracy Stream	Maximum	Slow	Professional evaluation, detailed product analysis	Full pipeline (YOLO + MediaPipe + IPD-based dimensional accuracy)

Table 5.1 outlines the three-tier backend API try-on methods.
The **Single Image (REST)** approach has moderate speed with high accuracy, making it ideal for evaluation and screenshots.
Its pipeline performance accuracy rests on the fusion of YOLO detection and MediaPipe landmarks with scaled interpupillary distance (IPD) that realsticaly aligns

add-on accessories.
The **Real-Time (WebSocket)** stream is designed for sustained rapid user interaction and automatically varying detection quality for optimal performance, thereby providing seamless user experience responsive real-time try on sessions even on low bandwidth.
The **High-Accuracy Stream** shifted its focus towards accuracy and now, with full activation of the YOLO + MediaPipe pipeline and IPD-adjusted dimensional accuracy, is slower but ideal for professional evaluation and high-fidelity demos.
The table, in essence, captures the balance of speed, accuracy, and usability trade-offs, detailing how the system meets user demands for casual, professional, and real-time interactions.

5.1 Functional Performance

The object detection and facial landmark estimation for virtual accessory (glasses and hats) posing is done in real-time using YOLOv8 and MediaPipe, respectively. OpenCV composition further guarantees that the overlays are applied in proper scale, offset, and rotation. The dual-camera view (before/after accessory) serves a comparative try-on, enhancing the overall system usability. Latency and throughput were benchmarks within records. REST-based prototypes latency averaged at 350–400 ms per frame, which is not suitable for real-time use. WebSocket plummeted this latency down to 80–100 ms, which gives an exceptional smooth real-time interaction at 12–15 frames per second.

5.2 Accuracy

The calculation of accuracy depends on together with alignment measurement on the spatial position of landmarks and the frame overlay. YOLO + MediaPipe had a 92% success rate with equal lighting conditions, 78% with MediaPipe alone. Calibration in 3d space further increased realism, products were consistently in proportion across different users.

5.3 Usability

User testing (n=10) demonstrated different preferences across the modes:

- Single Image Try-On was the most praised for its quality and usefulness for product screenshots.
- Real-Time Stream was rated most engaging, even though there were some small alignment issues.

- High-Accuracy Mode was dismissed because it was considered too slow for casual use, but is seen as a great tool for visualization of products in a professional case.

5.4 Limitations

Noting all of these strong results, some limitations also have to be taken into account. The system has difficulty with severe head poses and also requires some calibration for non-standard shapes of a face. Even though resource-intensive, high-accuracy mode is a limitation considering the fact that this mode would be utilized for consumers.

5.5 Integration Feasibility

There is greater integration feasibility for the system moving from a single REST-based design to a more complex multi-tier architecture. The backend system offers three complementary integration modes.
Simple Image Try-on (Single REST Endpoints) is a single stateless endpoint that allows uploads of snapshots to be processed and returned with overlay outputs with a medium latency.
Real Time WebSocket Streaming allows for more interactivity and higher frame rate interactions since WebSocket Streaming shifts in light of frames to a constant channel.
High Accuracy Stream is a heavy computation-based process that captures the most precise overlays by fusing YOLO with MediaPipe detections, IP Dimensional scaling, and slower throughputs.
This gives the retailers a far greater competitive advantage due to the variations provided from other AR solutions, Lau & Rhee, 2021. They can select from REST with rapid scaling, WebSockets for real-time interactions, or detailed High Accuracy Stream visualisations.

5.6 User Experience and Limitations

The different choices a user can have, offers different user experiences.
WebSocket Real-Time: Focuses on smoothness and interactivity. The user feels virtually no delay, making it great for live browsing and quick switching between products. Though, it accuracy suffers under limitation of bandwidth, and presence of noise, or landmarks.
High-Accuracy Stream: This mode focuses on stream alignment that prioritizes precision and realism over speed. By combining interpupillary distance (IPD) scaling, IPD-based streaming calibration, and dense MediaPipe Face Mesh landmarks,

the overlays achieve a much closer match to the user's facial geometry and the real-world product dimensions. However, this accuracy comes at a significant cost: the frames-per-second (FPS) is low, making the method unsuitable for fluid interactive browsing. Furthermore, because every frame must be captured by the client and transmitted to the backend before being processed and returned, the pipeline is heavily dependent on internet connection quality and stability. Slower or unstable networks further degrade responsiveness, meaning that while this mode provides the most realistic output, it is best reserved for professional use cases or detailed product evaluation rather than everyday consumer try-on.
Single-Image Try-On: Most suited for evaluation and for cases that involve screenshots. Offers stable accuracy, but the static nature limits immersion.

Challenging conditions like poor lighting or rotating one's head too much still limits realism. Dual-method validation where the converted stream is added (YOLO + MediaPipe) also improves robustness. However, occlusion and dynamic scaling still create large artifacts and greatly reduce realism. These trade-offs seem to be the known issues that AR try-on systems have. These systems lack realism in comparison to true try-on [34].

5.7 Scalability and Deployment

Adding to the architecture, the new dynamics of scaling will be introduced.
REST Single Image Mode is still completely stateless and remains horizontally scalable. Each request is self-contained, thus making it cloud-compatible. **Session persistence** in WebSocket mode increases servers' memory and workload requirements. Each active stream uses resources that are allocated at the time of the connection and are tethered until the stream is closed. Therefore, scaling WebSocket connections needs to be done delicately, and proper load balancing is necessary.
High Accuracy Mode is more convenient for professional users and low volume retail scenarios. Each instance of GPU allocation allocated for High Accuracy Mode results in a lower instance of concurrency.
Even with these concerns, the server-side approach maintains the main advantage of removing the AR logic from the front-end devices. Retailers can update detection models or scaling algorithms without re-downloading the mobile app ¬– overcoming the device lock-in slowdowns discussed in prior AR frameworks [7, 8]. The concerns with privacy for data that is streamed remain [26], especially with WebSocket sustaining continuous flows of frames.

5.8 Overall Assessment

The analysis for the improved system of the multiple tiers of the architecture brings us these main observations.
Strengths: Flexibility for various use cases is provided from the multiple tiers.
Strong detection with fallback methods is provided in fusion detection from YOLO + MediaPipe.
The overlay measurements are scaled with IPD to provide more realistic Augmented Reality scenes.
The multi-tier architecture of the backends allows device independent integration.
Weaknesses: The gap in the WebSocket and High-Accuracy mode trade-off is yet to be addressed.
Overlay realism is still in need of refinement under difficult combinations of head poses and light.
The added complexity of infrastructure streamlined from the WebSocket system becomes a major complexity in the system architecture.
To conclude, the new system moves beyond proof-of-concept and showcases a flexible, retail-ready AR try-on backend with multiple, purpose-built customer modes and deployment aligned settings.

Chapter 6

Conclusion

6.1 Summary of Work

The aim of this thesis was to assess if a backend-centric, front-end agnostic implementation could provide an integration-ready solution for AR try-on in fashion retail. Unlike traditional AR systems, such as those built on ARKit and ARCore, which suffer from platform dependency, scalability, and cross-device compatibility issues, this work proposed and implemented a lightweight, stateless API using FastAPI, alongside modern computer vision techniques (YOLOv8 object detection [10, 5], MediaPipe FaceMesh landmarks [13]) and OpenCV composition. The project delivered a three-tier try-on system:

- A **Single-Image REST endpoint** for product evaluation and screenshots, which was built and optimized for accuracy and responsiveness in the medium range.
- A **Real-Time WebSocket stream** for continuous try-on which, with priority to speed over accuracy, near real-time feedback at adaptive quality scales.
- A **High-Accuracy stream** in which maximum precision can be achieved by the integration of detection using YOLO and MediaPipe, IPD-based scaling, pose correction and a sacrifice in performance.

These implementations reveal the practicality of modular backend pipelines that decouple AR logic from the front end, allowing retailers to offer virtual try-on without the need for hardware-specific constraints.

6.2 Key Contributions

This thesis demonstrates three vital contributions.

1. **Architecture.** Created an AR try-on API with front-end agnostic capabilities that can be integrated into any retail system through REST and WebSocket protocols, proving backend-first, device agnostic AR alternatives.

2. **Enhanced Detection and Placement.** The use of a fusion pipeline where YOLO face detection and MediaPipe's 468 landmark descriptor are integrated. Biological reference system was derived from Interpupillary distance (IPD) data which scaled and converted millimeter sized products into pixels for more realistic overlays.

3. **Evaluation.** Comprehensive assessment of the system including integration and user experience, architectural scalability, and accuracy trade-offs. The assessment focuses on the backend-driven AR approach and its implications for scalability and update cycle optimization, while also recognizing the drawbacks in latency, privacy, and realism of the overlay.

6.3 Limitations

While proving the system's technical feasibility, several issues remain.

- **Realis.** Such accessories are visually realistic, but lack the occlusion and light matching, and scale estimation needed for authentic physical try-on.

- **Performance trade-offs.** The tension between real-time system logic and high accuracy makes the system trade off speed for precision: the WebSocket adaptive streaming system improves on-line synchronous speed, while the high-accuracy streaming system forgoes real-time responsive behavior.

- **Lack of integration.** In actual retail use, integration entails linking the API to a product database or a content delivery network for dynamic asset control, which was the integration limitation for this proof of concept system.

- **Privacy.** The act of offloading video streams raises concerns that relate to the privacy of users as shown in privacy preserving AR work [26, 32].

6.4 Future Work

There are still many possibilities to further this work.

- **Hybrid edge-cloud deployment.** Latency while maintaining scalability could be improved by integrating 5G edge servers with cloud backends [7, 8].

- **Approaches that preserve privacy.** Encrypted feature transfer and differential privacy can help resolve issues of trust and regulatory compliance.

- **Standardisation of AR APIs.** The use of OpenAPI or GraphQL schemas would enhance interoperability across different vendors and ease incorporation into retail ecosystems.
- **Increased realism.** Work should be done on neural rendering and GAN-based methods (for example, VITON, CP-VTON [27, 28]) for pose estimation to enhance perception and fidelity.
- **User studies.** Systematic usability research would yield tangible evidence regarding consumer trust, willingness to use the technology, and acceptance across different cultures for backend-driven AR try-on.

Bibliography

[1] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, August 1997.

[2] Miaolong Yuan, Ishtiaq Rasool Khan, Farzam Farbiz, and Arthur Niswar. A mixed reality system for virtual glasses try-on. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry (VRCAI)*, pages 251–258. ACM, 2011.

[3] Bin Zhang. Augmented reality virtual glasses try-on technology based on the ios platform. *Journal of Image and Video Processing*, 2018(132):1–9, 2018.

[4] Camillo Lugaresi, Jiuqiang Tang, Hadon Nash, Chris McClanahan, Esha Uboweja, Michael Hays, Fan Zhang, Chuo-Ling Chang, Ming Guang Yong, Juhyun Lee, et al. Mediapipe: A framework for building perception pipelines. *arXiv preprint arXiv:1906.08172*, 2019.

[5] Glenn Jocher et al. Yolov8: Ultralytics next-generation object detection and segmentation models. *Zenodo*, 2023.

[6] Paul A. Viola and Michael J. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 2001.

[7] Ke Zhang, Yong Xu, Fan Wang, and Kun Yang. Towards ubiquitous augmented reality: A survey of mobile ar offloading and edge computing. *IEEE Communications Surveys & Tutorials*, 22(4):2469–2498, 2020.

[8] Yujie He, Xiang Li, and Min Chen. Edge-assisted real-time ar with 5g and cloud offloading. *IEEE Network*, 35(3):196–203, 2021.

[9] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[10] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[11] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[12] Sebastián Ramirez. Fastapi: Modern, fast (high-performance), web framework for building apis with python 3.6+, 2020. Accessed: 2025-08-29.

[13] Hao-Shu Fang, Shuqin Xie, Yu-Wing Tai, and Cewu Lu. Rmpe: Regional multi-person pose estimation. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2334–2343, 2017.

[14] Hao Li and Anuj Kumar. Api-based scalable machine learning systems: Design and deployment challenges. *Journal of Systems and Software*, 188:111256, 2022.

[15] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[16] Oi Yee Lau and Chung Wai Ki. Can consumers' gamified, personalized, and engaging experiences with vr fashion apps increase in-app purchase intention? *Fashion and Textiles*, 8(1):36, 2021.

[17] Eunji Rhee and Jiyoung Lee. Effects of mobile ar app on consumer engagement and brand experience in fashion retail. *Sustainability*, 13(11):6336, 2021.

[18] Stefan Hauswiesner, Matthias Straka, and Gerhard Reitmayr. Free viewpoint virtual try-on with commodity depth cameras. In *Proceedings of the ACM SIGGRAPH VRCAI*, pages 167–172. ACM, 2011.

[19] Peter Eisert, Philipp Fechteler, and Jürgen Rurainsky. 3-d tracking of shoes for virtual mirror applications. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6. IEEE, 2008.

[20] Loh Cheak Ling. Virtual fitting room using augmented reality. Master's thesis, Universiti Tunku Abdul Rahman, Malaysia, 2020.

[21] Irtaza Hashmi and Nida Ahmed. An augmented reality based virtual fitting room using haarcascade classifier. In *2020 3rd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, pages 1–6. IEEE, 2020.

[22] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[23] Christopher Olston et al. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS*, 2017.

[24] Daniel Crankshaw et al. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 613–627, 2017.

[25] Daniel Crankshaw et al. The case for prediction serving systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC)*, pages 1–9, 2015.

[26] Jiale Hu, Tianlong Chen, and Feng Zhang. Privacy-preserving mobile ar with encrypted cloud offloading. In *Proceedings of IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[27] Xintong Han, Zuxuan Wu, Zhe Jiang, Zhanzhan Zhang, Jing Liu, and Larry S. Davis. Viton: An image-based virtual try-on network. *arXiv preprint arXiv:1711.08447*, 2018.

[28] Bochao Wang, Huabin Zheng, Xiaodan Liang, Yimin Chen, Liang Lin, and Meng Yang. Towards characteristic-preserving image-based virtual try-on. *arXiv preprint arXiv:1807.07688*, 2018.

[29] Denis Baylor et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017.

[30] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[31] Peter Kairouz, H. Brendan McMahan, et al. Advances and open problems in federated learning. *Foundations and Trends in Machine Learning*, 14(1–2):1–210, 2021.

[32] T. A. Syed. In-depth review of augmented reality: Tracking, security, and user challenges. *Sensors (MDPI)*, 23(1), 2022.

[33] J. Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, January 1991.

[34] Eduard Holdack, Kirill Lurie-Stoyanov, and Hanno Felix Fromme. The role of perceived enjoyment and informativeness in assessing the acceptance of ar wearables. *Journal of Retailing and Consumer Services*, 65:102259, 2020.

Chapter 7

Appendix

The complete implementation of the AR-based Virtual Try-On API, including backend (FastAPI), frontend (React + Vite), and all configuration files, is available in the following GitHub repository:

`https://github.com/saimsohail1/Masters-Project.git`

This repository contains:

- Backend code (FastAPI, YOLO, MediaPipe, OpenCV integration)
- Frontend demo client (React + Vite, TailwindCSS)

Readers can clone or download the repository to replicate experiments or extend the system for further research.