GITGOOD

Help gitGood, gitBetter

SRS ARCHITECTURE DOCUMENT

DEMO 4

WEATHER TO WEAR

EPI-USE

KYLE LIEBENBERG DIYA BUDHIA ALISHA PERUMAL	IBRAHIM SAID TAYLOR SERGEL

Table of Contents

INTRODUCTION

Weather to Wear is a mobile application that aims to simplify weather forecasts into a personalized wardrobe consultant. By analysing real-time weather forecasts with the combination of a user's personalised styling preferences, it uses AI-driven outfit suggestions to provide appropriate fashion recommendations from a user's personal clothing collection. The application seeks to streamline the daily challenge of choosing outfits that are weather-appropriate, ensuring users step out confidently prepared for any weather condition while expressing their unique fashion sense.

VISION AND OBJECTIVES

Weather to Wear aims to transform and simplify the daily act of choosing outfits into a personalized, engaging experience by using weather data, AI-driven outfit recommendations, and social interactions. This enables users to dress confidently and stylishly for any weather condition. The primary objective is to develop a system that integrates weather forecasts with users wardrobe to suggest outfits that are tailored to weather conditions and personal style preferences. The objective is to deliver a system that personalizes outfit recommendations, enhances user engagement and utilizes innovative visualization through the means of virtual try-ons.

BUSINESS NEEDS

Weather to Wear offers a unique, time-saving solution for outfit-planning. By combining weather-driven AI recommendations, AR visualizations, and social collaboration, Weather to Wear will stand out in the fashion-tech industry. It delivers an innovative, impactful solution that supports a growing user base with a scalable PWA, ensuring that data is handled in a secure manner and communication is encrypted for user privacy.

PROJECT SCOPE:

The project scope for Weather to Wear outlines the key features and deliverables for the mobile application. The project scope, based on the requirements are:

- Clothing Catalog System:
  - Develop a user-managed wardrobe database with detailed clothing profiles and options to add, edit, or remove items.
- Weather API Integration:
  - Implement a flexible model to integrate multiple weather APIs for reliable, real-time forecast data.
- Intelligent Recommendation Engine:
  - Create an AI-driven system using neural networks to suggest personalized, weather-appropriate outfits with a user rating system for learning preferences.
- Social Platform:
  - Build a real-time feed for sharing outfits, enabling comments and collaborative feedback, with content filtering for safety.
- Construct a Virtual Closet Management System:
  - Develop tools to organize a user's wardrobe by categories such as occasion or season, enabling easy navigation and selection.
- Create an Outfit Emergency Response Platform:
  - Design a system to deliver quick outfit recommendations in response to sudden weather changes, ensuring users can adapt their clothing choices efficiently.
  - Provide practical adaptation instructions for modifying current outfits to suit unexpected weather conditions, minimizing disruption.
- Predictive Wardrobe Planning:
  - Develop a system that generates outfit plans for upcoming trips or events by integrating precise weather forecasts for the destination and dates, enabling users to pack and prepare efficiently.
- Visual Calendar:
  - Create a visual calendar displaying complete outfit recommendations for the upcoming week, with adaptive modifications that update automatically as weather forecasts change, ensuring users are always prepared for current conditions.

USER STORIES

AUTHORIZATION

1. As a new user, I want to sign up with my details so that I can create an account and start using the system.
2. As a registered user, I want to log in so that I can access my closet, outfits, and recommendations securely.

CLOSET MANAGEMENT

1. As a user, I want to add my clothing items to the closet catalog so that the system can recommend appropriate outfits based on my wardrobe.
2. As a user, I want to add detailed tags to clothing items, such as color and temperature suitability
3. As a user, I want to upload single or multiple clothing items so that I can build a virtual version of my wardrobe.
4. As a user, I want to categorize my closet items (e.g., tops, pants, shoes) so that I can easily find them when creating outfits.
5. As a user, I want to edit details of clothing items from my virtual closet so that my closet catalog stays up-to-date with my current physical wardrobe
6. As a user, I want to remove clothing items from my virtual closet so that my closet catalog stays up-to-date with my current physical wardrobe.

WEATHER API SYSTEM

1. As a user, I want the system to pull real-time weather data from the weather API so that I receive accurate outfit recommendations
2. As a user, I expect constant polling of real-time weather data from multiple weather APIs so that I receive accurate outfit recommendations even when one API fails
3. As a user, I want outfit suggestions based on the day's weather forecast and my personal preferences.
4. As a user, I want to rate recommended outfits so that the system learns my style preferences over time.
5. As a user, I want to search for city matches so that I can check weather before traveling.

SOCIAL MEDIA FEED

1. As a user, I want to share my outfit combinations on a real-time feed
2. As a user, I want to comment on friends' outfit posts so that I can engage in collaborative style discussions
3. As a user, I want inappropriate content to be filtered from the social feed so that the platform remains safe and enjoyable
4. As a user, I want to create, edit, and delete posts so that I can share my outfits with others.
5. As a user, I want to like or unlike posts so that I can show appreciation for outfits I enjoy.
6. As a user, I want to follow or unfollow people so that my feed is tailored to who I want to see.
7. As a user, I want to receive and manage notifications (e.g., new follower, comment, like) so that I stay updated.
8. As a user, I want to accept or reject follow requests so that I control who sees my posts.
9. As a user, I want to search for other users so that I can connect with people with similar styles.

EVENT MANAGEMENT

1. As a user, I want tools to organize my wardrobe by season so that I can easily find items for specific events.
2. As a user, I want to categorize outfits by occasion so that I can quickly select appropriate clothing for events like work or parties.
3. As a user, I want outfit plans for upcoming trips based on destination weather forecasts so that I can pack and prepare efficiently.
4. As a user, I want a visual calendar of outfit recommendations for the week so that I can plan my wardrobe with adaptive updates as forecasts change.
5. As a user, I want to be able to create an event to plan future outfits
6. As a user, I want to be able to add details to an event, including location, date and occasion so that outfit suggestions align with the event.
7. As a user, I want to see upcoming events on the calendar page.
8. As a user, I want to be able to see upcoming events on the dashboard/home page for efficiency.

EMERGENCY PUSH NOTIFICATIONS

1. As a user, I want quick outfit recommendations for sudden weather changes so that I can adapt my clothing without stress.
2. As a user, I want instructions for adapting my current outfit to unexpected weather so that I can make practical adjustments on the go.

INSPIRATION PAGE

1. As a user, I want to like outfits or items so that the system can generate personalized outfit inspiration.
2. As a user, I want the system to generate outfit inspiration based on my likes and wardrobe so that I can discover new styles.
3. As a user, I want to view all my inspiration posts in one place so that I can revisit them anytime.
4. As a user, I want to remove inspiration posts I no longer like so that my inspiration feed stays curated to my style.

USER SETTINGS

- As a user, I want to be able to set my clothing preference, such as formal, casual, etc. so that outfit suggestions align with my personal taste.
- As a user, I want to be able to set my color preferences, so that the system can suggest outs that align with my personal tastes.
- As a user, I want to change my username or email so that my account details stay current.
- As a user, I want to update my profile photo so that others recognize me in the social feed.
- As a user, I want to update my privacy settings so that I can control who sees my posts and closet.

OUTFIT RECOMMENDATION SYSTEM

- As a user, I want the system to generate outfits for upcoming events based on event details, personal preferences and the weather forecast.
- As a user, I want layering suggestions for temperature changes throughout the day so that I'm prepared for varying conditions.

TRY-ON AVATAR

- As a user, I want to use augmented reality to virtually try on outfit combinations so that I can see how they look before dressing.
- As a user, I want to save fitted outfits on my avatar so that I can revisit them later.
- As a user, I want to upload a photo of myself so that I can see how outfits look on my body.
- As a user, I want the system to overlay clothing on my uploaded image so that I can virtually try on outfits.
- As a user, I want to delete uploaded photos and try-on results so that my account stays private and uncluttered.

FUNCTIONAL REQUIREMENTS

The Functional Requirements provided below ensure that the Weather To Wear system meets users needs for an efficient, personalized and weather-informed outfit planning system. The requirements are organized by feature area and align with the user stories.

FR-1: Clothing-Catalog Management

- **FR-1.1:** The system shall allow a user to add a wardrobe item by taking a photo and completing a short form (name, category, material, weather-appropriateness, tags).
- **FR-1.2:** The system shall let the user edit or delete any wardrobe item they own.
- **FR-1.3:** The system shall provide search, filter and sort capabilities over the wardrobe (category, colour, material).
- **FR-1.4:** Wardrobe data and images shall be persisted so that a user's catalog is available across devices.
- **FR-1.5:** Each wardrobe item shall support tagging for weather conditions (hot, rainy, sunny) and occasions (work, casual).
- **FR-1.6:** The system shall enforce a maximum file size and accepted image types (JPEG, PNG) during wardrobe uploads.
- **FR-1.7:** The system shall allow batch uploads of wardrobe items.
- **FR-1.8:** The system shall allow users to mark wardrobe items as favourites.

FR-2: Weather-Data Integration

- **FR-2.1:** The system shall retrieve a location-specific weather forecast (min/max temperature, precipitation, wind, humidity, condition) from an external API.
- **FR-2.2:** If the primary provider fails, the system shall automatically fallback to a secondary API without user intervention.
- **FR-2.3:** Weather data shall be refreshed at a configurable interval (default: every 6 hours) so that recommendations stay current.
- **FR-2.4:** The system shall attempt to automatically detect the user's location using IP-based geolocation.
- **FR-2.5:** If automatic detection fails, the system shall allow manual location input from the user.

FR-3: Wardrobe-Recommendation Engine

- **FR-3.1:** Given a forecast and the user's wardrobe, the system shall produce at least one ranked outfit recommendation for a selected day.
  - **FR-3.1.1:** The ranking shall consider temperature range, weather condition, and user preferences.
- **FR-3.2:** The engine shall learn from user feedback captured through a 5 star rating system.
- **FR-3.3:** Recommendations shall include layering advice when forecasts indicate significant temperature variation.
- **FR-3.4:** The engine shall exclude items marked unsuitable for the predicted conditions.
- **FR-3.5:** The system shall avoid recommending identical outfits across consecutive days, where alternatives exist.
- **FR-3.6:** The engine shall take time-of-day into account (e.g., colder mornings vs. warm afternoons) if available in the forecast.
- **FR-3.7:** The engine shall generate outfit recommendations for specific user events.
- **FR-3.8:** The engine shall provide quick recommendations on demand.

FR-4: Social-Sharing Platform

- **FR-4.1:** Users shall be able to publish an outfit post (image + caption + list of items) to a social feed.
- **FR-4.2:** Followers shall be able to like and comment on a post in real time.
  - **FR-4.2.1:** Likes and comments shall update asynchronously without page reload.
- **FR-4.3:** The system shall filter or block offensive text or images before a post becomes visible.
- **FR-4.4:** Privacy controls shall let the author restrict visibility to:
  - **FR-4.4.1:** Public
  - **FR-4.4.2:** Followers only
  - **FR-4.4.3:** Private (visible only to user)
- **FR-4.5:** The system shall allow users to follow and unfollow others.
  - **FR-4.5.1:** A users feed shall show posts of those who they follow
- **FR-4.6:** The system shall allow users to search for other users by name or username.
- **FR-4.7:** The system shall generate notifications for user actions (likes, comments, follows, requests).
- **FR-4.8:** The system shall allow users to accept or reject follow requests.

FR-5: User-Feedback & Learning

- **FR-5.1:** The system shall record each outfit rating action against the corresponding outfit version.
- **FR-5.2:** The user's preferences shall adapt based on all forms of feedback (ratings, likes, favourites).
  - **FR-5.2.1:** The recommendation engine shall incorporate user's preference and weather conditions into future ranking within 24 hours or upon user refresh.
- **FR-5.3:** The system shall allow users to view and manage their past feedback history.
- **FR-5.4:** The engine shall increase outfit diversity if consistent negative feedback is received for similar combinations.
- **FR-5.5:** The system shall generate outfit inspiration content based on user likes and favourites.

FR-6: Virtual-Closet Analytics

- **FR-6.1:** The system shall display usage statistics (e.g., most-worn items, cost-per-wear).
- **FR-6.2:** Users shall be able to tag clothing by occasion and filter their closet on those tags.
- **FR-6.3:** The system shall visualize wardrobe insights using graphs (e.g., pie chart for usage distribution).

FR-7: Emergency-Outfit Response

- **FR-7.1:** When the forecast changes beyond a configurable threshold (e.g., ≥ 5 °C drop or rain chance > 60%), the system shall:
  - **FR-7.1.1:** Notify the user
  - **FR-7.1.2:** Suggest a new outfit suitable for the new forecast
- **FR-7.2:** Notifications shall be sent via in-app alert or push notification if enabled.

FR-8: Fashion Time-Machine Planner

- **FR-8.1:** The system shall allow users to attach events or trips to future calendar dates.
- **FR-8.2:** For each future date with an event, the system shall:
  - **FR-8.2.1:** Pre-compute an outfit based on long-range forecast
  - **FR-8.2.2:** Update the outfit automatically if the forecast changes
- **FR-8.3:** Users shall be able to update or delete events.

FR-9: Virtual Try-On

- **FR-9.1:** The app shall provide a virtual try on feature.
  - **FR-9.1.1:** Via an image of the user.
  - **FR-9.1.2:** A customizable 2D avatar.
- **FR-9.2:** The try-on experience shall function in modern browsers that support camera access.
- **FR-9.3:** The system shall allow users to upload a personal photo for outfit try-on.
- **FR-9.4:** Users shall be able to delete uploaded photos and try-on results.
- **FR-9.5:** The system shall allow saving fitted outfits for later viewing.

FR-10: Cross-Cutting Functionalities

- **FR-10.1:** Authentication & Authorisation
  - **FR-10.1.1:** All user-modifiable endpoints shall require authentication.
  - **FR-10.1.2:** Each user shall only be able to access and modify their own wardrobe, preferences, and posts.
- **FR-10.2: Offline Mode**
  - **FR-10.2.1:** The PWA shall cache the latest wardrobe and recommendations.
  - **FR-10.2.2:** The app shall queue user actions (e.g., ratings) and sync them once back online.
  - **FR-10.2.3:** Weather forecasts shall be cached to allow the recommendation engine to generate outfits while offline
- **FR-10.3:** Multiplatform Delivery
  - **FR-10.3.1:** The app shall work in modern browsers (Chrome, Safari, Firefox).
  - **FR-10.3.2:** The app shall support installation as a home screen shortcut on desktop.

- **FR-10.4: Logging & Monitoring**
  - **FR-10.4.1:** The backend shall log API failures, user actions, and system warnings.
  - **FR-10.4.2:** Admins shall be able to review logs via the dashboard.
- **FR-10.5: Testability**
  - **FR-10.5.1:** Each module (auth, weather, recommendation) shall include unit and integration tests.
  - **FR-10.5.2:** The system shall be CI/CD integrated, running tests on each publish.
- **FR-11: Remove Background**
  - **FR-11.1:** Upload automatically removes the background
  - **FR-11.2:** User confirms removed background is accurate

- **FR-12: Packing Lists**
  - **FR-12.1:** The system shall allow users to create packing lists for trips or events.
  - **FR-12.2:** Users shall be able to update and delete packing lists.
  - **FR-12.3:** The system shall generate packing list suggestions based on destination weather and events.

- **FR-14: User Profile Management**
  - **FR-14.1:** The system shall allow users to update their username and email.
  - **FR-14.2:** The system shall allow users to update their profile photo.
  - **FR-14.3:** The system shall allow users to configure privacy settings.

NON-FUNCTIONAL REQUIREMENTS

OVERVIEW

Each subsection states **what we do** (Architectural Strategies), **why it helps**, and **how we measure.**

PERFORMANCE

Users expect outfit suggestions and image uploads to feel quick. Slow endpoints lead to drop-offs and poor ratings.

ARCHITECTURAL STRATEGIES

- **DB tuning:** indices on ownerId/userId/createdAt, cursor pagination.
- **Caching:** short-lived weather cache (15 min per city) and outfit cache (until regenerated or logout) to avoid repeated calls.
- **Queueing and concurrency** for uploading closet items, removing background, and extracting colours.
- **Static media offload:** images served from **S3 + CDN** instead of the API.
- **Efficient client:** bundle splitting and lazy loading so the browser downloads only what's needed.

WHY IT HELPS

Indexed queries + pagination keep tail latencies low as data grows. Caches convert slow externals into fast reads. CDN removes bytes from the API.

SLOS

- API reads p95 ≤ 800 ms (p99 ≤ 1500 ms) @ 20 req/s
- 5-outfit generation ≤ 1.5 s with 50 items
- BG removal median ≤ 5 s; colour extraction median ≤ 2 s.
- API for weather reads p95 ≤ 3500 ms (p99 ≤ 4000 ms) @ 20 req/s

AVAILABILITY

Users rely on the app daily; if it's frequently down, they won't trust it.

ARCHITECTURAL STRATEGIES

- App Runner **rolling deploys** with **health checks**.
- Managed Postgres (RDS) **automated backups**
- **S3 for media** (high durability) so images remain accessible even if the API restarts.

WHY IT HELPS

Instances rotate without downtime. Durable storage reduces data-loss risk.

SLOS

- API ≥ 99.5% per month
- Microservices combined ≥ 99.0%
- Zero-downtime routine deploys (verified by canary checks).

SCALABILITY

More users and larger closets should not make the system crawl.

ARCHITECTURAL STRATEGIES

- Horizontal scale of stateless backend and microservices.
- Selective microservices for image processing so heavy CPU work scales independently.
- CDN absorbs image spikes
- Caches reduce external load.
- Concurrency for the image processing queue to increase throughput as users grow.

WHY IT HELPS

Scale out without redesign and isolate CPU-heavy work.

SLOS

- ≥ 20 concurrent users with ≥ 5 req/s within latency SLOs
- Bursts of 6 uploads/min for 5 minutes with each completing in 10 seconds.

SECURITY

We store personal data and user photos; we must protect accounts and media.

ARCHITECTURAL STRATEGIES

- **JWT authentication** on all non-public endpoints and **RBAC** (user/admin).
- **Password hashing** with **bcrypt** (≥10 rounds).
- **HTTPS/TLS** end-to-end and strict CORS allowlist.
- **Private S3 buckets** with **pre-signed URLs** for upload/download
- **Input validation** and lightweight **rate limiting** to reduce abuse.

WHY IT HELPS

Strong auth controls protect accounts; encrypted transport and private storage prevent leaks; least-privilege reduces blast radius.

SLOS

- 100% protected routes require valid JWT
- All passwords hashed
- All traffic over HTTPS
- Security headers present

LATENCY

Even if servers are fast, users judge speed by how quickly pages become usable.

ARCHITECTURAL STRATEGIES

- **CDN** for images with regional caching.
- **Client code-splitting** and **lazy loading** to reduce initial download.
- **Cache weather** summaries to avoid blocking on external calls.
- **Cache outfits** to avoid regeneration upon refresh.

WHY IT HELPS

Fewer bytes/round-trips means faster interactivity.

SLOS

- PWA Time-to-Interactive ≤ **3.5 s** on mid-range mobile over 4G.
- Route transitions ≤ **500 ms**
- Image TTFB via CDN ≤ **300 ms** regionally.

RELIABILITY

Users shouldn't lose data or see random errors. External APIs and networks do fail. Our app should handle it gracefully.

ARCHITECTURAL STRATEGIES

- **Remove single points of failure** by having **backup** weather APIs.
- **Logging** failures to external calls to detect faults with weather APIs.
- **Timeouts and retries with exponential backoff** for outbound calls (weather, image services).
- **Idempotent operations** for ratings/uploads.
- **Daily backups** and tested restore; application-level validation to prevent bad writes.

WHY IT HELPS

Controlled retries recover from transient issues. Idempotency prevents duplicated actions. Backups/restore plans protect data.

SLOS

- Error budget ≤ 0.5% of requests returning 5xx over 30 days.
- Backups nightly
- **RTO ≤ 2h**

EXTENSIBILITY

We plan to evolve from a rules/KNN engine to a richer ML model without breaking the app.

ARCHITECTURAL STRATEGIES

- **Stable REST contracts** under /api/v1.
- **Pluggable outfit scoring:** rules engine + KNN currently; ML scorer will be added behind the same service interface.
- **Prisma migrations** for controlled schema evolution.

WHY IT HELPS

Clear boundaries let us change internals without changing clients. Migrations keep DB changes safe and reviewable

SLOS

- New scoring method can be deployed without client changes.
- Adding a clothing category or style only touches schema + enums + rules configuration.

ARCHITECTURAL SPECIFICATIONS

OVERVIEW

Weather to Wear is a Progressive Web App (PWA) that recommends daily outfits from a user's wardrobe based on local weather, selected style, user preferences, and prior outfit ratings. The solution uses a **Multi-Layered** architecture:

- **Presentation Layer (MVC):**
  - React PWA.
- **Application Layer:**
  - HTTP edge & routing, controller entry points.
- **Logic Layer:**
  - Domain services (Auth, Closet, Outfit, Events/Trips, Social, User Prefs, Weather client).
- **Persistent Layer:**
  - Repositories, Prisma ORM, media gateways, and local caches — the abstraction between domain logic and data stores.
- **Data Access Layer:**
  - RDS PostgreSQL (relational data) and S3 (+ CloudFront) for media.
- **Microservices:**
  - FastAPI services for background removal and colour extraction.

Hosted on AWS: **CloudFront + S3 (frontend & media), App Runner** (backend & microservices), **RDS PostgreSQL (private)**, and **Secrets Manager.**

Architectural Design Strategy

DESIGN DRIVERS

- Fast, reliable recommendations and media handling on student hardware evolving to AWS.
- Isolate compute-heavy image work for independent scaling and fault isolation.
- Stable contracts so outfit scoring can evolve (rules/KNN → ML) without client breakage.
- Keep data access concerns **out** of controllers/service logic (via the **Persistent Layer).**

STRATEGY SUMMARY (BY LAYER)

- **Presentation (MVC):**
  - Views (React components), Controllers (event handlers/API calls), Model (component/query state). Keeps UI responsive and testable.
- **Application Layer:**
  - Defines routes and request lifecycle (validation, auth, mapping to services).
- **Logic Layer:**
  - Pure domain services; no direct I/O. Calls repositories and media gateways.
- **Persistent Layer:**
  - Repositories (Prisma), media gateway (S3 via pre-signed/CDN URLs), and bounded in-process caches (weather, event-day snapshots). This separates how we store/fetch from what the domain needs.
- **Data Access:**
  - RDS with indices/pagination; S3 behind CloudFront for immutable images.
- **Microservices:**
  - FastAPI containers for background removal (rembg) and colour extraction (K-Means), invoked by the backend.

HOSTING & OPS

- **App Runner** services (backend + two microservices) with health checks and rolling deployments.
- **RDS** in private subnets (VPC connector from App Runner).
- **S3** buckets private; CloudFront OAC for reads; backend writes via IAM.
- **Secrets Manager** for DATABASE_URL, JWT_SECRET; plain env for non-sensitive config.
- **Prisma migrations** baked into the image and executed on boot (prisma migrate deploy).

Architectural Patterns

MULTI-LAYERED ARCHITECTURE

(Presentation -> Application -> Logic -> Persistent -> Data Access)
A structural split into distinct layers with strict boundaries:

- **Presentation (MVC)** – React PWA: Views, Controllers (handlers), Model (component/query state).
- **Application Layer** – HTTP routing, request validation, auth checks, mapping to domain services.
- **Logic Layer** – Stateless domain services (auth, closet, outfit, events/trips, social, userPreference, weather) containing business rules only.
- **Persistent Layer** – Repositories (Prisma), media gateways (S3/CDN), and bounded in-process caches (e.g., weather). This layer abstracts storage and external data concerns away from the Logic Layer.
- **Data Access Layer** – RDS PostgreSQL for relational data; S3 + CloudFront for immutable media objects.

IMPLEMENTATION

- **Presentation:** React PWA (MVC on the client).
- **Application:** Node/Express REST API (stateless) on AWS App Runner; validation & auth happen here.
- **Logic:** Domain services orchestrate use cases (e.g., “recommend outfits”).
- **Persistent:**
  - ***Repositories:*** Prisma queries/mutations against Postgres (RDS via VPC Connector).
  - ***Media Gateway:*** writes images to S3 and returns CDN (CloudFront) URLs.
- **Caches:** in-process weather/event-day summaries (short TTL).
- **Data:** PostgreSQL (RDS) for entities, S3 for images with CloudFront for fast global delivery.
- **Example flow:** View -> GET /api/v1/outfits/recommendations -> Application Layer validates/auth → Logic Layer Outfit pulls closet & ratings via Repositories + Weather via Cache -> returns JSON with image CDN URLs.

IMPACT ON QUALITY ATTRIBUTES

- **Performance & Latency:**
  - Logic stays CPU-light (I/O pushed to Persistent), reducing per-request overhead.
  - Media offload to CDN cuts payload size/round-trips to the API.
  - Caches in Persistent convert slow external calls into fast reads.
- **Scalability:**
  - Stateless Application/Logic layers scale horizontally on App Runner.
  - Data and media scale independently (RDS vs S3/CloudFront).
- **Availability and Reliability:**
  - Layered fault isolation: media/CDN outages don't take down core logic; DB issues surface at Persistent and can be retried/fail gracefully.
  - Rolling deployments on App Runner reduce downtime.
- **Security:**
  - Clear ingress/egress control: auth only at Application; storage access confined to Persistent via least-privilege IAM; RDS private via VPC Connector.
- **Extensibility & Maintainability:**
  - New algorithms (e.g., ML scorer) slot into Logic without storage rewrites.
  - Schema changes are localized to Repositories (Prisma migrations).

TRADE OFFS AND MITIGATIONS

- **Trade-off:**
  - More layers can add indirection.
- **Mitigation:**
  - Keep interface contracts slim; enforce dependency direction (Presentation → … → Data) with lint/rules and module structure.

REST (RESOURCE-ORIENTED)

Stateless HTTP with resource URIs, standard methods (GET/POST/PUT/PATCH/DELETE), standard status codes, cache control, and versioning.

IMPLEMENTATION

- All modules expose endpoints under /api/v1.
- Consistent error envelope: { error: { code, message, details? } }.
- Pagination/filters; input validation (App Layer).
- Auth via Authorization: Bearer <JWT>; RBAC for admin routes.
- Example: GET /api/v1/closet?layer=base_top → validate → indexed repo query → paginated JSON (200) or errors (422/401/403/404).

IMPACT ON QUALITY ATTRIBUTES

- **Performance & Latency:**
  - Cacheable GETs (weather summaries, outfits) reduce server load and response time.
  - Predictable payloads enable client-side memoization.
- **Reliability:**
  - Idempotent PUT/PATCH/DELETE semantics allow safe retries on transient failures.
  - Clear error codes simplify client fallback logic.
- **Security:**
  - Standardized auth headers, consistent authz checks in one place (App Layer), and CORS enforcement per route.
- **Extensibility:**
  - Versioning (/api/v1) allows internal refactors or new scorers without breaking clients.
  - New resources (e.g., ratings, outfit-ml-scores) can be added alongside existing routes.

TRADE OFFS AND MITIGATIONS

- **Trade-off:**
  - REST can be chatty for composite reads.
- **Mitigation:**
  - Server-side composition in Logic, pagination & includes on repository queries, and selective response shaping.

SELECTIVE MICROSERVICES (IMAGE PROCESSING)

Small, independently deployable services focused on compute-heavy, specialized tasks that scale separately from the core API.

IMPLEMENTATION

- **Background Removal** (FastAPI + rembg) and **Colour Extraction** (FastAPI + K-Means), each as its own App Runner service.
- The image processing queue (Background removal and Colour Extraction) process concurrently to increase throughput.
- The boundary supports a future **queued/async** pipeline (e.g., SQS) without changing public REST contracts.

IMPACT ON QUALITY ATTRIBUTES

- **Performance:**
  - Moves CPU-intensive work out of the main request path API threads remain responsive.
  - Specialized Python stack (NumPy/ML libs) is faster for image ops than Node.
- **Scalability:**
  - Microservices autoscale independently (CPU-based) as image volume spikes; backend replicas don't need to scale at the same rate.
- **Availability & Reliability:**
  - Fault isolation: microservice failures don't crash the API.
  - Timeouts/retries with jitter are tuned specifically for image ops.
- **Extensibility:**
  - New image steps (e.g., background matting, saliency maps) can be introduced as additional services behind the same gateway interface.
- **Security:**
  - Narrowed IAM permissions (only what the service needs).
  - Traffic remains TLS-protected and there is no public access to RDS.

MVC (CLIENT)

UI pattern separating Controller (event handlers/navigation), Model (React state + React-Query cache), and View (React components).

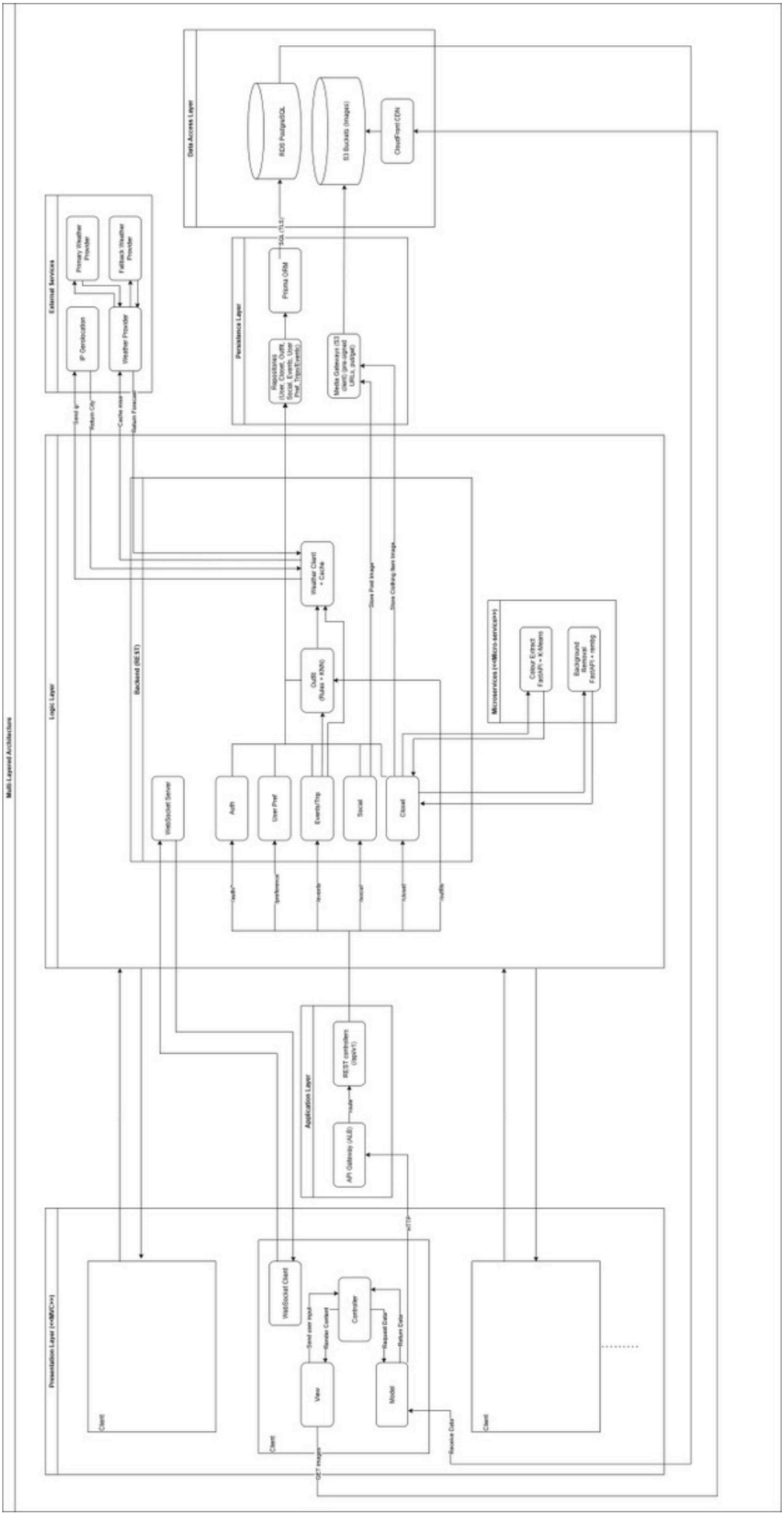
IMPLEMENTATION

- **View:** React components for outfits, closet, events, social feed.
- **Controller:** Component/Hook handlers call REST endpoints, coordinate UI flows, and push updates to the Model.
- **Model:** Local state + query caches (normalized data, pagination cursors, optimistic updates).
- **Realtime:** WebSocket client receives push events (e.g., “item processed”), controller updates Model and app still works fully over REST if sockets are off.

IMPACT ON QUALITY ATTRIBUTES

- **Latency (UX) & Performance:**
  - Minimal re-renders via state co-location and memoization.
  - CDN image URLs avoid blocking content.
  - Query caching yields instant back/forward navigation and low TTFB for repeated views.
- **Reliability:**
  - Predictable flow (event → state → render) reduces race conditions and errors surface in a single place (Controller).
  - Optimistic updates with rollback handle flaky networks gracefully.
- **Security:**
  - Consistent JWT handling at controllers.
  - No secrets in views.
  - CORS respected per fetch.
- **Extensibility & Maintainability:**
  - New views or flows compose existing models/controllers and business rules remain in the backend Logic Layer (thin client).

Architectural Diagram



LAYERS & RESPONSIBILITIES

- **Presentation (MVC)** – The React PWA separates responsibilities into Controller (event handlers, navigation, and API calls), Model (local and cached state, hooks), and View (React components). A lightweight WebSocket client receives push notifications.
- **Multi-Layered** – An App Runner routes requests to the stateless Node/Express backend. The backend is organised by domain services/modules: Auth, Closet, Outfit (rules + KNN), Events/Trips, Social, UserPrefs, and a Weather client with an in-process TTL cache. A WebSocket server supports optional realtime UI updates.
- **Microservices (Selective)** – Two FastAPI services perform background removal and dominant colour extraction. They are independently scalable and technology-isolated. The uploads queue is processed concurrently.
- **Data** – Relational domain data in PostgreSQL (RDS). Immutable media objects in S3 behind CloudFront for global delivery. Records in Postgres keep S3 object keys to maintain referential integrity.
- **External** – Weather provider (FreeWeather/OWM) and IP geolocation (ip-api) are accessed via the backend only, with short-lived caching for performance and rate-limit protection.

KEY FLOWS

- **Closet upload:**
  - Controller → Logic.Closet → Persistent.MediaGateway (BG-Removal → Colour-Extract) → S3 → Persistent.Repository (Prisma) → RDS → response with CDN URLs.
- **Outfit recommendation:**
  - Controller → Logic.Outfit (rules + KNN) → Persistent.Repository (closet, ratings) + Persistent.Cache (weather summary) → response with item CDN URLs.
- **Events/trips:**
  - Controller → Logic.Events → Persistent.Repository; per-day outfits reuse the weather/event caches.
  -

The effect this architectural structure has on the quality requirements was discussed in the Architectural Patterns section.

Architectural Constraints

TECHNOLOGY

- Backend: Node.js/Express (TypeScript), Prisma.
- Database: PostgreSQL (Amazon RDS), private subnets.
- Client: React PWA (MVC).
- Image processing: FastAPI microservices (rembg; K-Means).
- Storage/CDN: S3 (private) + CloudFront.
- Hosting: AWS App Runner (backend + microservices), CloudFront+S3 (frontend), Secrets Manager.

OPERATIONAL

- Synchronous media pipeline initially; design permits a message queue without changing public APIs.
- Prisma migrations required for schema evolution; migrations run on container start in prod.
- Caches are bounded and time-boxed (e.g., weather ≈30 min); cache keys include city/date/style.
- No direct SQL in controllers/services — all persistence via Repositories.
- Observability: structured logs; latency/5xx metrics; alarms; daily DB snapshots.

LEGAL & DATA

- Minimal PII; images are user-owned; buckets private; access via pre-signed URLs/CDN.
- Comply with university/project policies.

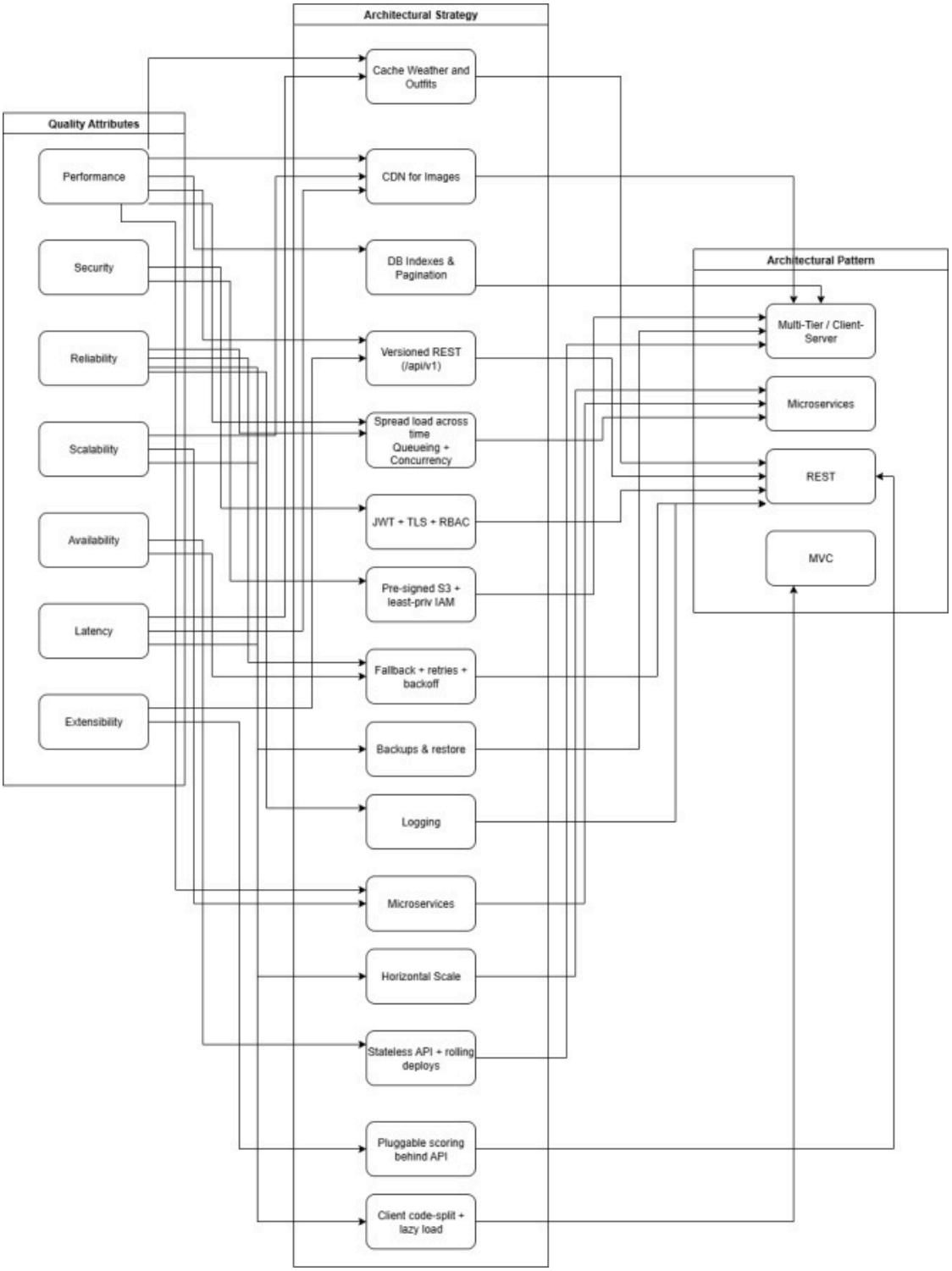
TEAM & TIMELINE

- SLOs chosen for demo-grade reliability with student capacity constraints.
- Prefer managed AWS services to reduce ops overhead.

EXTENSIBILITY

- Outfit scoring is behind a stable REST surface; new ML scorer can be added in the Logic Layer without changing clients.
- Schema evolution via Prisma only; API versioned under /api/v1.

MAPPING BETWEEN QUALITY ATTRIBUTES TO STRATEGIES TO PATTERNS



DEPLOYMENT SPECIFICATIONS

OVERVIEW

Below describes the deployment of Weather to Wear. This includes the Nodes, components and artifacts as well as the communication path and deployment diagram.

Nodes Components and Artifacts

CLIENT EDGE

- Client <<node>>
- Browser/PWA that requests the frontend over HTTPS and calls the backend REST API.

FRONTEND DELIVERY

- Amazon CloudFront – Frontend CDN <<node>>
  - HTTPS <<interface>> to the client.
  - Serves the React PWA from the S3 origin using Origin Access Control (OAC).
- Amazon S3 – Frontend Bucket <<node>>
  - Build React App (build/) <<artifact>> uploaded by CI/CD.
  - Public access blocked; only CloudFront reads via OAC.
- GitHub Actions (OIDC) – Deploy Frontend <<node>>
  - Build Artifact <<artifact>> (React build output) synced to the frontend S3 bucket; CloudFront invalidation completes the release.

BACKEND COMPUTE

- AWS App Runner – Backend API <<node>>
  - Weather to Wear API (Node.js + Express + Prisma) <<component>>
  - REST API v1 /api/* <<interface>> exposed to the client over HTTPS.
  - Docker image weather-backend:prod <<artifact>> pulled from ECR.
  - Reads secrets at start-up; runs prisma migrate deploy before serving traffic (ensures schema parity).
  - Environment (plain): PORT, NODE_ENV, S3_BUCKET_NAME, S3_REGION, UPLOADS_CDN_DOMAIN, BG_REMOVAL_URL, COLOR_EXTRACT_URL.

IMAGE PROCESSING MICROSERVICES

- AWS App Runner – bg-removal (public) <<node>>
  - Background Removal Service (U2-Net) <<component>>
  - HTTP POST /remove-bg <<interface>>
  - Docker image bg-removal:prod <<artifact>>.
- AWS App Runner – color-extract (public) <<node>>
  - Color Extraction Service (KMeans) <<component>>
  - HTTP POST /extract-colors <<interface>>
  - Docker image color-extract:prod <<artifact>>

CONTAINER REGISTRY & SECRETS

- Amazon ECR <<node>>
  - OCI Images (tags: prod) <<artifact>> for backend and both microservices.
- AWS Secrets Manager <<node>>
  - Secrets <<artifact>>: DATABASE_URL, JWT_SECRET, and weather API keys.
  - Backend fetches via GetSecretValue at boot.

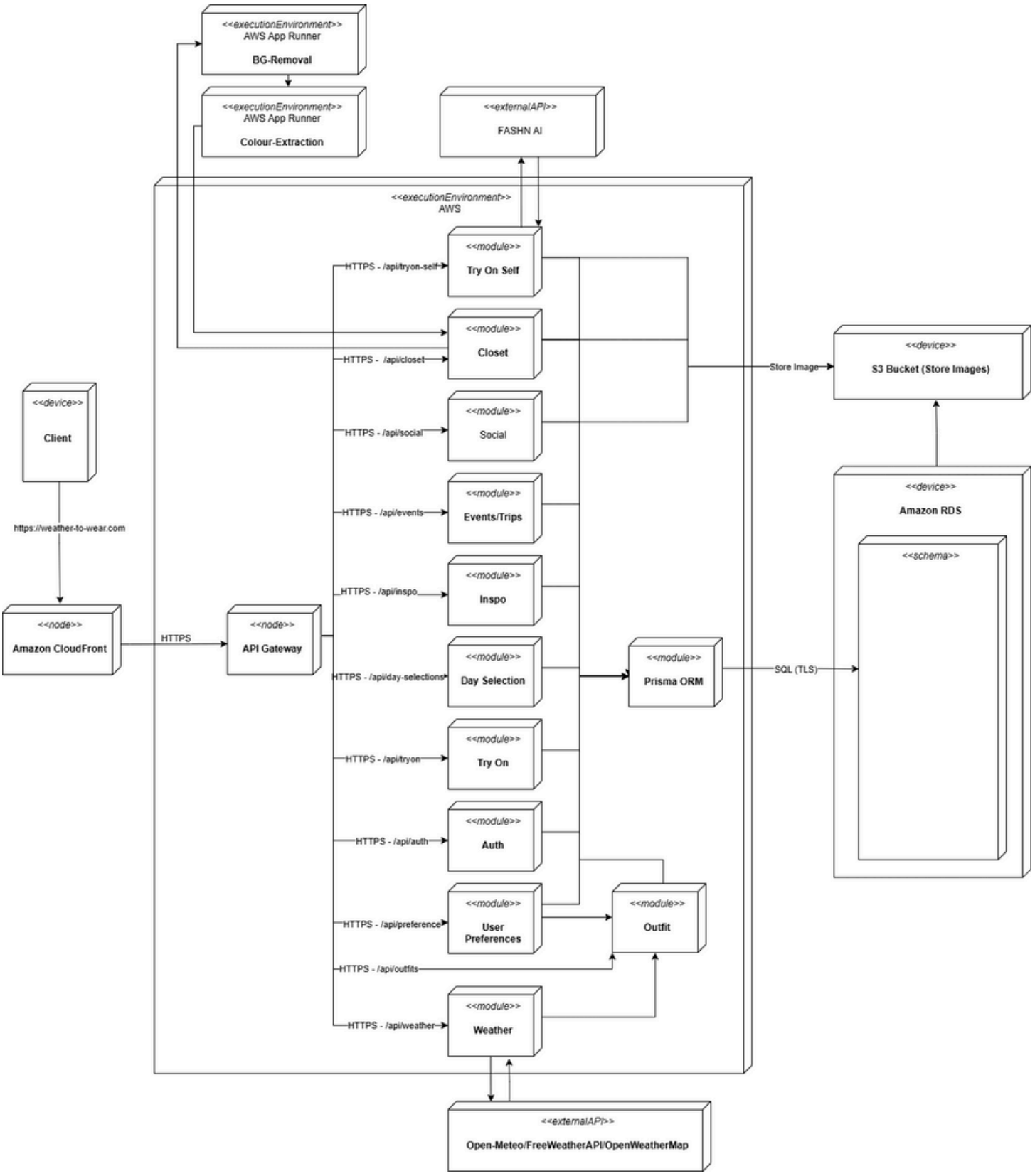
PRIVATE NETWORK & DATA STORES

- VPC: w2w-prod-vpc <<node>>
  - S3 Gateway Endpoint <<node>> for private S3 egress (saves NAT cost/latency).
  - App Runner VPC Connector <<node>> provides private connectivity to RDS on port 5432/TLS.
  - Private Subnets <<node>> host:
    - Amazon RDS PostgreSQL <<node>> (instance w2w-postgres-prod, port 5432, no public endpoint).
      - RDS Automated Backups <<artifact>>
- Amazon CloudFront – Uploads CDN <<node>>
  - HTTPS <<interface>> for serving user images globally (via OAC to S3).
- Amazon S3 – Uploads Bucket <<node>>
  - User Images (PNG/JPEG/WebP) <<artifact>>, private with Block Public Access ON.
  - Backend writes via IAM; CloudFront reads via OAC.

Runtime Communication Paths

1. **PWA delivery**
   a. Client → CloudFront (Frontend) <<interface:HTTPS>> → S3 Frontend Bucket via OAC
   b. The PWA is entirely static; API base URL is injected at build time.
2. **User API traffic**
   a. Client → App Runner – Backend API <<interface:REST /api/v1>>.
   b. Requests pass through the Application Layer (routing/validation/auth) and into domain services.
3. **Secrets retrieval**
   a. Backend API → Secrets Manager (GetSecretValue for DATABASE_URL, JWT_SECRET) at boot.
4. **Database Access**
   a. Backend API → App Runner VPC Connector → RDS PostgreSQL on 5432/TLS (private).
   b. Prisma performs queries via Repositories in the Persistent Layer.
5. **Media write path (uploads)**
   a. Backend API (after processing) → S3 Uploads Bucket (Put/Delete via IAM).
   b. Stored object keys are returned as CDN URLs to the client.
6. **Media read path (images)**
   a. Client → CloudFront (Uploads) <<interface:HTTPS>> → S3 Uploads via OAC.
   b. Buckets remain private; no public ACLs/policies.
7. **Image processing pipeline**
   a. a.Backend API → bg-removal POST /remove-bg → color-extract POST /extract-colors → return metadata/bytes → S3 write → persist DB record.
8. **Container images**
   a. App Runner services pull Docker images <<artifact>> from
   b. ECR during deployment.
9. **Migrations on stat**
   a. Backend container runs prisma migrate deploy at boot
   b. against RDS to ensure the live schema matches code (e.g.,
   c. isTrip column).

DEPLOYMENT DIAGRAM



CLASS DIAGRAM

The class diagram presented below represents the core architecture of the Weather To Wear system. It aligns with the aim to create a personalized outfit planning application that integrates weather data, user preferences, wardrobe management, and social features. This diagram outlines the relationships between entities, services, and business logic components. Below is an outline of each class, its components and relevance to the system.

CLASSES AND COMPONENTS

USER

OVERVIEW

- Represents a registered user of the Weather To Wear system.
- A user registers with information such as their username, password and email.
- A unique identifier is assigned to the user.

RELEVANCE TO SYSTEM

- Aligns with the functional requirement of Authentication and Authorization.
- Ensures each user has a secure profile.

USER PREFERENCE

OVERVIEW

- Represents a set of user-specific clothing preferences within the Weather to Wear system.
- A user defines preferences including style, temperature sensitivity, and preferred colors.
- A unique identifier is assigned to each preference set, linked to a single user.

RELEVANCE TO SYSTEM

- Supports the functional requirement of User-Feedback & Learning by allowing preference updates.
- Enables personalized outfit recommendations, aligning with the preference-setting user story.

CLOTHING ITEM

OVERVIEW

- Represents an individual item in the user's virtual wardrobe within the Weather to Wear system.
- An item is created with details such as an image, type, color, material, warmth factor, and favorite status.
- A unique identifier is assigned to each item, owned by a specific user.

RELEVANCE TO SYSTEM

- Aligns with the functional requirement of Clothing-Catalog Management (FR-1) for adding, editing, and tagging items.
- Provides the data pool for outfit generation, supporting the outfit generation user story.

WARDROBE

OVERVIEW

- Represents a collection of ClothingItem instances forming the user's virtual closet in the Weather to Wear system.
- It supports operations to add, edit, or delete items, managed by a single user.
- The wardrobe is uniquely associated with each user’s profile.

RELEVANCE TO SYSTEM

- Supports the functional requirement of Clothing-Catalog Management for item management and filtering.
- Serves as the source of clothing data for the outfit recommendation engine.

OUTFIT

OVERVIEW

- Represents a recommended combination of ClothingItem instances in the Weather to Wear system.
- An outfit is composed of multiple items and includes methods to rate the outfit or generate a 3D model for virtual try-on.
- Each outfit is generated based on user preferences and weather conditions.

RELEVANCE TO SYSTEM

- Aligns with the functional requirement of Wardrobe-Recommendation Engine and Virtual Try-On.
- Central to the outfit generation user story, enhancing user confidence with 3D previews.

OUTFIT RECOMMENDATION ENGINE

OVERVIEW

- Represents the core logic component for generating outfit recommendations in the Weather to Wear system.
- It processes user data, wardrobe items, and weather information to produce ranked outfit suggestions.
- The engine includes a method to generate recommendations tailored to specific conditions.

RELEVANCE TO SYSTEM

- Supports the functional requirement of Wardrobe-Recommendation Engine for personalized recommendations.
- Directly implements the outfit generation user story, integrating event and preference data.

WEATHER SERVICE

OVERVIEW

- Represents the service responsible for retrieving weather data in the Weather to Wear system.
- It detects user location, fetches weather forecasts from primary or secondary APIs, and returns structured weather data.
- The service ensures reliable data access with fallback mechanisms.

RELEVANCE TO SYSTEM

- Aligns with the functional requirement of Weather-Data Integration for forecast retrieval.
- Provides critical input for outfit recommendations, supporting the outfit generation user story.

WEATHER DATA

OVERVIEW

- Represents the structured weather forecast data used by the Weather to Wear system.
- It includes details such as temperature, humidity, condition, timestamp, and location.
- This data is retrieved and formatted by the WeatherService.

RELEVANCE TO SYSTEM

- Supports the functional requirement of Weather-Data Integration for accurate forecasts.
- Enables weather-based outfit ranking in the recommendation engine.

LOCATION

OVERVIEW

- Represents geographic location data used by the Weather to Wear system.
- It includes city, country, latitude, and longitude, used for weather queries and event planning.
- Location data can be detected automatically or input manually by the user.

RELEVANCE TO SYSTEM

- Aligns with the functional requirement of Weather-Data Integration (FR-2.4, FR-2.5) for location handling.
- Supports event creation by providing location context for forecasts.

AUTH SERVICE

OVERVIEW

- Represents the authentication service for the Weather to Wear system.
- It handles user registration, login, and token verification with methods like signUp, login, and verifyToken.
- A unique user profile is created or validated upon successful authentication.

RELEVANCE TO SYSTEM

- Supports the functional requirement of Authentication and Authorization for secure access.
- Ensures user data privacy, critical for tech-savvy users managing personal wardrobes.

SOCIAL PROFILE

OVERVIEW

- Represents a user's social presence within the Weather to Wear system.
- It enables users to create posts sharing their outfits and interact with others.
- A social profile is linked to a registered user’s account.

RELEVANCE TO SYSTEM

- Aligns with the functional requirement of Social-Sharing Platform for sharing posts.
- Enhances user engagement through social feedback, aligning with user characteristics.

SOCIAL FEED PAGE

OVERVIEW

- Represents the social feed interface in the Weather to Wear system.
- It displays posts from users a person follows, updated in real time.
- The feed is accessible through the user's dashboard.

RELEVANCE TO SYSTEM

- Supports the functional requirement of Social-Sharing Platform for feed display.
- Encourages social interaction, catering to users' desire for community engagement.

POST

OVERVIEW

- Represents a social media post within the Weather to Wear system.
- A post includes an outfit image, caption, and item list, created by a user's SocialProfile.
- Posts are subject to privacy and moderation controls.

RELEVANCE TO SYSTEM

- Aligns with the functional requirement of Social-Sharing Platform for publishing.
- Facilitates sharing and feedback, supporting users' social engagement needs.

DASHBOARD

OVERVIEW

- Represents the central hub interface of the Weather to Wear system.
- It provides access to user profiles, outfit recommendations, and social feeds.
- The dashboard is the primary navigation point for all user interactions.

RELEVANCE TO SYSTEM

- Supports the functional requirement of Multiplatform Delivery as a unified interface.
- Enhances usability for tech-savvy users by centralizing key features.

AI PREFERENCE MACHINE

OVERVIEW

- Represents an AI-driven component for refining user preferences in the Weather to Wear system.
- It updates style, temperature sensitivity, and preferred colors based on user feedback.
- The machine interacts with UserPreference to adapt recommendations.

RELEVANCE TO SYSTEM

- Aligns with the functional requirement of User-Feedback & Learning for preference updates.
- Improves personalization, supporting the preference-setting user story.

CLASS DIAGRAM