Sprawozdanie z realizacji projektu:
Inteligentny Asystent Zarządzania Wydatkami (Smart Receipt)

Martyna Gaj
Patrick Bajorski
Kacper Gałek
Jan Banasik
Gabriel Filipowicz

5 stycznia 2026

Spis treści

1 Wstęp

1.1 Geneza i motywacja projektu

W obliczu dynamicznie zmieniającej się sytuacji gospodarczej, rosnącej inflacji oraz powszechnej niepewności rynkowej, umiejętność świadomego zarządzania budżetem domowym staje się jedną z kluczowych kompetencji współczesnego człowieka. Dla wielu osób śledzenie codziennych wydatków jest jednak procesem żmudnym, czasochłonnym i zniechęcającym. Konieczność gromadzenia papierowych paragonów, a następnie ręcznego przepisywania zawartych na nich danych do arkuszy kalkulacyjnych lub prostych aplikacji mobilnych, stanowi skuteczną barierę dla systematyczności. W rezultacie, wiele prób wdrożenia kontroli finansowej kończy się niepowodzeniem już po kilku tygodniach, a użytkownicy tracą cenne informacje o strukturze swoich kosztów życia.

Projekt „Inteligentny Asystent Zarządzania Wydatkami" (Smart Receipt) narodził się z potrzeby rozwiązania tego problemu poprzez technologię. Naszym celem było stworzenie narzędzia, które zminimalizuje wysiłek użytkownika do absolutnego minimum. Zamiast ręcznego wprowadzania danych, system przejmuje na siebie ciężar ekstrakcji informacji, interpretacji danych i ich kategoryzacji. Wierzymy, że automatyzacja tych procesów jest kluczem do budowania trwałych nawyków finansowych.

1.2 Cel i zakres prac

Głównym założeniem projektu jest dostarczenie kompleksowego systemu informatycznego, który umożliwi użytkownikom:

1. **Automatyzację wprowadzania danych:** Poprzez wykorzystanie zaawansowanych technik optycznego rozpoznawania znaków (OCR) oraz modeli językowych, aplikacja potrafi odczytać paragon ze zdjęcia wykonanego smartfonem.

2. **Inteligentną kategoryzację:** System nie tylko rozpoznaje tekst, ale rozumie jego kontekst. Potrafi automatycznie przypisać produkty do odpowiednich kategorii (np. „Spożywcze", „Transport", „Chemia"), co pozwala na natychmiastową analizę struktury wydatków.

3. **Wizualizację i analizę:** Zgromadzone dane są prezentowane w formie czytelnych wykresów i raportów na interaktywnym dashboardzie, co pozwala użytkownikowi na szybką ocenę swojej sytuacji finansowej.

Projekt realizowany w ramach przedmiotu Studio Projektowe miał również na celu praktyczne sprawdzenie możliwości integracji nowoczesnych rozwiązań z zakresu sztucznej inteligencji (AI) w architekturze mikroserwisowej.

1.3 Zastosowane technologie i architektura

System został zaprojektowany w oparciu o architekturę modularną, co zapewnia łatwość skalowania i niezależny rozwój poszczególnych komponentów. Główne filary technologiczne to:

- **Backend:** Zaimplementowany w języku Java (Spring Boot), odpowiedzialny za logikę biznesową, zarządzanie użytkownikami i bezpieczeństwo.

- **Frontend:** Dynamiczny interfejs użytkownika stworzony w bibliotece React, zapewniający responsywność i intuicyjną obsługę.
- **Moduł AI:** Niezależny mikroserwis napisany w języku Python (FastAPI), integrujący modele uczenia maszynowego.

Dalsza część niniejszego sprawozdania skupia się szczegółowo na działaniu modułu sztucznej inteligencji, który stanowi najbardziej innowacyjny element systemu Smart Receipt.

2 Moduł AI – Szczegóły implementacyjne

Moduł AI to serce systemu analitycznego. Funkcjonuje on jako oddzielny kontener Docker, wystawiający API RESTowe. Jego zadaniem jest przyjęcie pliku graficznego (zdjęcia paragonu), przetworzenie go i zwrócenie ustrukturyzowanych danych JSON zawierających listę zakupionych produktów wraz z ich cenami, ilościami i przypisanymi kategoriami.

Proces ten jest wieloetapowy i składa się z:

1. Optycznego rozpoznawania znaków (OCR).
2. Wstępnego czyszczenia tekstu (Pre-processing).
3. Semantycznego parsowania przy użyciu LLM.
4. Kategoryzacji produktów (NLP).
5. Walidacji końcowej (Post-processing).

2.1 Środowisko uruchomieniowe i sprzętowe

Ze względu na wykorzystanie dużego modelu językowego (Llama 3.1 8B), moduł AI posiada specyficzne wymagania sprzętowe. Aplikacja wspiera akcelerację GPU (NVIDIA CUDA), co jest kluczowe dla zapewnienia akceptowalnego czasu odpowiedzi.

Wdrożono mechanizm **GPU Offloading**, sterowany zmienną środowiskową `SR_GPU_LAYERS`. Pozwala on na elastyczne zarządzanie zasobami pamięci VRAM:

- **Tryb Hybrydowy (15 warstw w GPU):** Umożliwia uruchomienie modelu na kartach z 4 GB VRAM. Część obliczeń wykonywana jest na procesorze graficznym, a reszta na CPU.
- **Tryb Pełny (33+ warstw w GPU):** Przy dostępności 8 GB+ VRAM, cały model jest ładowany do pamięci karty, co skraca czas przetwarzania jednego paragonu do ok. 2-3 sekund.
- **Tryb CPU:** W przypadku braku karty NVIDIA, system automatycznie przełącza się na obliczenia procesorowe.

2.2 Etap 1: Ekstrakcja tekstu (PaddleOCR)

Do zadania detekcji tekstu wybrano bibliotekę **PaddleOCR**. Jest to lekkie i wydajne rozwiązanie, które charakteryzuje się wysoką skutecznością w rozpoznawaniu języka polskiego oraz tekstów zniekształconych perspektywicznie. W projekcie wykorzystano model ‘PP-OCRv3‘, który działa w trybie ’use_angle_cls=True’, co pozwala na automatyczną korekcję orientacji tekstu, gdy zdjęcie paragonu jest obrócone o 90 czy 180 stopni.

Co istotne, w celu optymalizacji zasobów, OCR jest uruchamiany wyłącznie na procesorze (CPU). Dzięki temu cała pamięć VRAM karty graficznej pozostaje dostępna dla modelu językowego LLM, który jest znacznie bardziej wymagający obliczeniowo.

2.3 Etap 2: Pre-processing i czyszczenie surowych danych

Tekst zwrócony przez OCR zawiera dużą ilość szumu informacyjnego – nagłówki sklepów, dane podatkowe, stopki marketingowe. Przed przekazaniem go do modelu LLM, konieczne jest jego oczyszczenie. Pozwala to zmniejszyć liczbę tokenów wejściowych (oszczędność mocy obliczeniowej) oraz zwiększyć precyzję modelu.

W pliku `cleaning.py` zdefiniowano zestaw reguł filtrujących. Funkcja `clean_raw_text` analizuje tekst linia po linii i usuwa te, które zawierają tzw. „Garbage Markers".

Tabela 1: Lista markerów powodujących usunięcie linii (Garbage Markers)

Kategoria	**Przykłady markerów**
Dane podatkowe	NIP, REGON, BDO, PTU, PODATEK, OPODATKOWANA, VAT
Adresowe	ADRES:, UL., SKLEP, MIASTO
Płatności	SUMA:, DO ZAPŁATY, RAZEM, RESZTA, KARTA, GOTÓWKA
Techniczne	PARAGON FISKALNY, NR SYS, KASJER, WYDRUK, F20, EAG
Inne	SPRZEDAŻ, DATA, GODZINA, WALUTA, PLN, EUR

Dodatkowo stosowane są wyrażenia regularne (Regex) usuwające:

- Daty w formatach DD-MM-YYYY lub DD.MM.YYYY.
- Linie krótsze niż 3 znaki.
- Ciągi znaków będące wyłącznie liczbami (bez nazw produktów).

2.4 Etap 3: Parsowanie semantyczne (Llama 3.1)

To kluczowy moment przetwarzania. Oczyszczony tekst, będący nadal "płaską"listą napisów, trafia do lokalnie uruchamianego modelu **Llama 3.1 8B Instruct** (kwantyzacja Q4_K_M).

Model nie działa tutaj jako chatbot, lecz jako precyzyjny parser. Wykorzystano technikę *Structured Output* (Grammar-based sampling), wymuszając na modelu zwrócenie odpowiedzi ściśle zgodnej ze schematem JSON.

2.4.1 Konfiguracja Promptu

Aby uzyskać powtarzalne wyniki, opracowano specjalistyczny prompt systemowy i użytkownika.

```
system_prompt = (
    "Jesteś parserem paragonów. Zwracaj TYLKO JSON zgodnie ze schematem.
    "
)

user_prompt = f"""
Wyciągnij listę produktów.

ZASADY:
1. Analizuj linia po linii.
2. Quantity zawsze 1.0 (chyba że tekst mówi inaczej).
3. Zachowaj każde wystąpienie produktu.
4. Poprawiaj literówki.
```

```
13
14 INPUT (OCR):
15 '''
16 {text_to_process}
17 '''
18 """
```

Listing 1: Definicja promptu dla modelu Llama

2.4.2 Parametry inferencji

Model uruchamiany jest z bardzo restrykcyjnymi parametrami, aby ograniczyć halucynacje:

- `temperature=0.1`: Minimalizuje losowość, model wybiera najbardziej prawdopodobne tokeny.
- `repeat_penalty=1.05`: Zapobiega wpadaniu modelu w pętle powtórzeń (częsty problem przy OCR z wieloma liniami o podobnej strukturze).
- `n_ctx=4096`: Długość kontekstu wystarczająca na przetworzenie nawet bardzo długich paragonów zakupowych.

2.5 Etap 4: Hybrydowa kategoryzacja produktów

Po uzyskaniu listy produktów, system musi przypisać im odpowiednie kategorie biznesowe. Wykorzystano podejście dwuetapowe, łączące szybkość słowników statycznych z inteligencją modelu NLP.

2.5.1 Ścieżka szybka (Słownikowa)

W pierwszej kolejności nazwa produktu jest porównywana z wbudowaną bazą słów kluczowych (`KEYWORDS` w pliku `categorizer.py`). Jeśli nazwa zawiera dany ciąg znaków, kategoria jest przypisywana natychmiastowo, z pominięciem sieci neuronowej. Jest to rozwiązanie wysoce optymalne wydajnościowo dla najpopularniejszych produktów.

Poniższa tabela prezentuje wybrane definicje słownikowe zaimplementowane w systemie:

Tabela 2: Słownik słów kluczowych dla kategoryzacji (fragment)

Kategoria	**Słowa kluczowe (podciągi)**
Alcohol and stimulants	PIWO, WÓDKA, WINO, WHISKY, PAPIEROSY, TYTOŃ, L&M, MARLBORO, HEETS, VUSE, LIKIER, SPIRYTUS
Groceries	BUŁKA, CHLEB, KAIZERKA, BAGIETKA, MLEKO, ŁACIATE, SER, JOGURT, MASŁO, TWARÓG, ŚMIETANA, JAJA, SZYNKA, SCHAB, KIEŁBASA, KURCZAK, MIESO, POMIDOR, OGÓREK, ZIEMNIAK, MARCHEW, PASSATA, KNORR, WODA, NAPÓJ, COLA, PEPSI, SPRITE

Kategoria	Słowa kluczowe (podciągi)
Household and chemistry	DOMESTOS, PŁYN, PROSZEK, KAPSUŁKI, BATERIE, WORKI, PAPIER TOALETOWY, RĘCZNIK, MYDŁO, TORBA, REKLAMÓWKA
Transport	BILET, PALIWO, PB95, ON, UBER, BOLT, PARKING, MPK, PKP

Jak widać, słownik uwzględnia również częste błędy OCR (np. „LACIATE" zamiast „ŁACIATE", „PLATNOSC") oraz nazwy marek („KNORR", „WINIARY", „UBER"), co znacząco zwiększa skuteczność systemu.

2.5.2 Ścieżka inteligentna (SetFit)

Jeżeli produkt nie zostanie dopasowany słownikowo, uruchamiany jest model **SetFit** (Sentence Transformer Fine-tuning). Jest to model wyspecjalizowany w klasyfikacji krótkich tekstów przy małej liczbie próbek treningowych.

Model analizuje semantykę nazwy produktu i zwraca przewidywaną kategorię wraz z pewnością. W systemie zdefiniowano stałą `CONFIDENCE_THRESHOLD = 0.6`.

- Jeśli pewność predykcji $\geq 60\%$, kategoria jest akceptowana.
- Jeśli pewność predykcji $< 60\%$, produkt otrzymuje kategorię „Other" (Inne).

Mechanizm ten zapobiega "zgadywaniu"kategorii przez AI w przypadku nazw niejednoznacznych lub błędnie odczytanych przez OCR.

2.6 Etap 5: Post-processing i walidacja końcowa

Ostatnim etapem jest czyszczenie i weryfikacja danych wyjściowych przed wysłaniem ich do bazy danych. Funkcja `clean_items` w pliku `cleaning.py` realizuje szereg reguł biznesowych i sanitarnych.

2.6.1 Czarna lista (Blacklist)

System usuwa pozycje, które mimo filtracji wstępnej zostały błędnie zaklasyfikowane jako produkty. Są to najczęściej słowa związane z promocjami i obsługą kasy, które przeniknęły przez wcześniejsze etapy. Przykłady z czarnej listy: *OBNIZKA, RABAT, PROMOCJA, GRATIS, ZYSKUJESZ, PODSUMOWANIE, DO ZAPŁATY, ROZLICZENIE, KARTA, GOTÓWKA, NIEFISKALNY, NETTO, BRUTTO.*

2.6.2 Reguły walidacji cen i nazw

Zaimplementowano następujące warunki logiczne dla każdego produktu:

1. **Cena:** Musi mieścić się w przedziale od 0.01 do 2000.00 jednostek. Ceny zerowe lub skrajnie wysokie (często wynikające z błędnego odczytania daty jako ceny, np. 2023 r. $\rightarrow$ 2023.00 PLN) są odrzucane.
2. **Nazwa:**
   - Długość nazwy musi wynosić co najmniej 3 znaki.

- Nazwa nie może składać się z samych cyfr i znaków interpunkcyjnych.
- Jeśli nazwa jest dłuższa niż 20 znaków i nie zawiera spacji, jest traktowana jako błąd OCR (ciąg losowych znaków) i usuwana.

3. **Oczyszczanie z "śmieci":** Z końcówek nazw usuwane są automatycznie oznaczenia ilościowe, które OCR często dokleja do nazwy produktu, np. „Mleko * 2szt" → „Mleko". Wykorzystywany jest do tego wzorzec Regex:
`r"(\*|\s\d+[\s]?szt|\s\d+[\s]?kg).*"`.

2.7 Podsumowanie modułu AI

Przedstawiony moduł stanowi przykład nowoczesnego podejścia do przetwarzania dokumentów. Zamiast polegać na sztywnych szablonach (co jest typowe dla starszych systemów OCR), Smart Receipt wykorzystuje elastyczność modelu językowego Llama 3.1 do "zrozumienia"struktury paragonu. Połączenie tego z hybrydową kategoryzacją (Reguły + SetFit) oraz wieloetapowym czyszczeniem danych (Regex) pozwoliło na stworzenie systemu, który jest jednocześnie odporny na błędy, wydajny i łatwy w utrzymaniu.

3 Backend – Architektura i Implementacja

Warstwa serwerowa aplikacji została zaprojektowana jako centralny punkt orkiestracji usług, odpowiedzialny za logikę biznesową, bezpieczeństwo danych oraz komunikację z bazą danych i modułem AI. Projekt realizowany był zgodnie z nowoczesnymi standardami inżynierii oprogramowania, kładąc szczególny nacisk na spójność kontraktów API oraz wydajność przetwarzania zapytań.

3.1 Stos technologiczny

Wybór technologii podyktowany był potrzebą stabilności, silnego typowania oraz dostępnością dojrzałych bibliotek wspierających architekturę REST.

- **Język programowania:** Java 21 (LTS) – wybrana ze względu na nowoczesne funkcje języka i wysoką wydajność.
- **Framework:** Spring Boot 3.5.7 – stanowi szkielet aplikacji, zapewniając mechanizmy Dependency Injection oraz konfigurację "convention over configuration".
- **Baza danych:** PostgreSQL – relacyjna baza danych wykorzystywana do przechowywania profili użytkowników, historii wydatków oraz definicji budżetów. Połączenie realizowane jest przez sterownik JDBC.
- **ORM:** Hibernate / Spring Data JPA – do mapowania obiektowo-relacyjnego.
- **Narzędzie budowania:** Gradle – wykorzystywane do zarządzania zależnościami i automatyzacji zadań.

3.2 Filozofia API-First Design

Jedną z kluczowych decyzji architektonicznych w projekcie było przyjęcie podejścia **API-First**. Oznacza to, że kontrakt interfejsu (specyfikacja OpenAPI/Swagger) jest tworzony przed napisaniem jakiegokolwiek kodu logiki biznesowej.

Plik ‘openapi.yaml‘ stanowi w projekcie tzw. „Jedyne Źródło Prawdy” (Single Source of Truth). Na jego podstawie automatycznie generowane są:

1. **Modele DTO (Data Transfer Objects)** na backendzie (Java).
2. **Interfejsy i typy** na frontendzie (TypeScript/Axios).

3.2.1 Workflow developerski

Aby zapewnić spójność między specyfikacją a kodem, wdrożono zautomatyzowany proces budowania oparty na pluginie ‘org.openapi.generator‘. W pliku konfiguracyjnym ‘build.gradle‘ zdefiniowano zadanie ‘generateServer‘, które jest uruchamiane przy każdej kompilacji projektu.

Skonfigurowano tryb generowania „Model-Only”, co daje programistom pełną kontrolę nad implementacją kontrolerów HTTP, jednocześnie zdejmując z nich ciężar ręcznego utrzymywania klas modeli JSON.

```
tasks.register('generateServer', org.openapitools.generator.gradle.plugin.tasks.GenerateTask) {
    generatorName = "spring"
    inputSpec = "$rootDir/src/main/resources/api/openapi.yaml"
    outputDir = "$buildDir/generated/server"
    modelPackage = "com.sp.smartreceipt.model"

    globalProperties = [
            models: "",
            apis: "false", // Generujemy tylko modele, kontrolery piszemy recznie
            supportingFiles: "false"
    ]

    // Mapowanie typow, np. Double -> BigDecimal dla precyzji finansowej
    typeMappings = [
            Double: "java.math.BigDecimal",
            float:  "java.math.BigDecimal"
    ]
}
```

Listing 2: Konfiguracja generatora OpenAPI w Gradle

Dzięki takiemu podejściu wyeliminowano ryzyko rozbieżności między dokumentacją a faktycznym działaniem API, a także zapewniono ścisłe typowanie danych finansowych poprzez wymuszenie mapowania typów zmiennoprzecinkowych na ‘java.math.BigDecimal‘.

3.3 Kluczowe wzorce projektowe i optymalizacje

W implementacji backendu zastosowano szereg wzorców mających na celu optymalizację wydajności, szczególnie w kontekście obsługi aplikacji przez urządzenia mobilne.

3.3.1 Wzorzec Summary vs Detail

Aby uniknąć przesyłania nadmiarowych danych (Over-fetching), system rozróżnia dwa typy modeli dla tego samego zasobu biznesowego:

- **ExpenseSummary:** „Lekki" obiekt zawierający tylko podstawowe dane (kwota, data, kategoria), wykorzystywany na listach wydatków. Pozwala to na szybkie pobranie historii setek transakcji bez obciążania łącza.
- **Expense (Detail):** Pełny obiekt zawierający listę wszystkich pozycji z paragonu, pobierany dopiero w momencie wejścia użytkownika w szczegóły konkretnego wydatku.

3.3.2 Backend-for-Frontend (Agregacja Dashboardu)

Endpoint '/dashboard' realizuje logikę agregacji danych po stronie serwera. Zamiast zmuszać aplikację frontendową do wykonywania wielu niezależnych zapytań (np. osobno o KPI, osobno o wykresy trendów, osobno o strukturę kategorii), backend przygotowuje jeden, zoptymalizowany obiekt 'DashboardData'. Redukuje to liczbę zapytań sieciowych i przyspiesza czas pierwszego renderowania interfejsu.

3.4 Bezpieczeństwo i autoryzacja

Za bezpieczeństwo aplikacji odpowiada moduł **Spring Security**. Implementacja opiera się na bezstanowym mechanizmie uwierzytelniania przy użyciu tokenów **JWT (JSON Web Token)**.

- **Rejestracja i logowanie:** Użytkownicy są weryfikowani, a system generuje parę tokenów: 'Access Token' oraz 'Refresh Token'.
- **Ochrona zasobów:** Każde zapytanie do API (z wyjątkiem endpointów autoryzacyjnych) jest przechwytywane przez filtr bezpieczeństwa, który weryfikuje poprawność podpisu cyfrowego tokena JWT.
- **Konfiguracja:** Klucze szyfrujące są wstrzykiwane z zmiennych środowiskowych lub plików konfiguracyjnych, co zapewnia separację kodu od danych wrażliwych.

3.5 Integracja z usługami zewnętrznymi

Backend pełni rolę pośrednika w komunikacji z mikroserwisem AI. Adres usługi AI jest konfigurowalny poprzez właściwość 'ai.service.url'. Proces dodawania paragonu przebiega synchronicznie: backend odbiera plik od użytkownika, przesyła go do modułu AI, a następnie zwraca użytkownikowi rozpoznaną strukturę JSON do zatwierdzenia. Taka architektura pozwala na ukrycie złożoności przetwarzania obrazu przed klientem mobilnym/webowym.

3.6 Model Danych i Mapowanie Obiektowo-Relacyjne (ORM)

System wykorzystuje standard JPA do mapowania obiektów Java na relacje w bazie danych PostgreSQL. Dzięki zastosowaniu biblioteki **Lombok**, kod encji został znacznie zredukowany poprzez automatyczne generowanie metod dostępowych, wzorca Builder oraz metod `equals` i `hashCode`.

3.6.1 Encje i Relacje

Główną encją w systemie jest `UserEntity`, która reprezentuje użytkownika aplikacji. Posiada ona relacje typu *One-to-Many* z kluczowymi zasobami systemu: kategoriami, wydatkami oraz budżetami miesięcznymi.

Zastosowano strategię `FetchType.LAZY` dla wszystkich kolekcji powiązanych (np. `expenses`, `categories`). Jest to kluczowa optymalizacja wydajnościowa – dane szczegółowe są pobierane z bazy danych tylko wtedy, gdy są jawnie zażądane przez aplikację, a nie przy każdym pobraniu obiektu użytkownika. Pozwala to uniknąć problemu „N+1 zapytań" i nadmiernego obciążenia pamięci RAM.

Warto zwrócić uwagę na wykorzystanie dwóch rodzajów identyfikatorów:

- **ID (Long):** Klucz główny bazy danych, używany wewnętrznie do relacji i indeksowania.
- **UUID (java.util.UUID):** Unikalny identyfikator biznesowy, wystawiany na zewnątrz w API. Użycie UUID zamiast sekwencyjnych liczb zapobiega możliwości odgadywania identyfikatorów zasobów przez osoby niepowołane.

```
@Entity
@Table(name = "users")
@Builder
@Data
public class UserEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private UUID userId; // Publiczny identyfikator

    @Column(unique = true, nullable = false)
    private String email;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, fetch =
    FetchType.LAZY)
    @Builder.Default
    private List<ExpenseEntity> expenses = new ArrayList<>();

    // ... pozostale pola i relacje
}
```

Listing 3: Definicja encji Użytkownika z relacjami JPA

Encja `ExpenseEntity` przechowuje kluczowe informacje finansowe, takie jak data transakcji oraz całkowita kwota. Wybór typu `BigDecimal` zamiast `Double` jest podyktowany koniecznością zachowania precyzji obliczeń finansowych i uniknięcia błędów zaokrągleń typowych dla liczb zmiennoprzecinkowych.

3.7 Szczegóły Konfiguracji Bezpieczeństwa

Klasa `SecurityConfig` definiuje łańcuch filtrów bezpieczeństwa (`SecurityFilterChain`), który chroni aplikację przed nieautoryzowanym dostępem.

Kluczowe elementy konfiguracji obejmują:

- **Polityka CORS:** Skonfigurowano `CorsConfigurationSource`, aby zezwalać na komunikację wyłącznie z zaufaną domeną frontendową (`http://localhost:3000`). Dozwolone są metody HTTP: GET, POST, PUT, DELETE, PATCH, OPTIONS.
- **Autoryzacja oparta na rolach (RBAC):** Zdefiniowano szczegółowe reguły dostępu do endpointów:
  - `/auth/**` – dostępne publicznie (logowanie, rejestracja).
  - `/admin/**` – dostępne tylko dla użytkowników z rolą `ADMIN`.
  - `/dashboard`, `/expenses/**` – dostępne dla roli `USER`.
- **Filtry:** Dodano niestandardowy filtr `JwtRequestFilter` przed domyślnym filtrem uwierzytelniania hasłem, co umożliwia logowanie bezstanowe przy użyciu tokena.

4 Frontend – Interfejs Użytkownika

Warstwa prezentacji została zaprojektowana z myślą o zapewnieniu intuicyjnej i responsywnej obsługi systemu, zarówno na komputerach stacjonarnych, jak i urządzeniach mobilnych. Aplikacja realizuje założenia typu SPA / Hybrid, wykorzystując nowoczesny framework Next.js.

4.1 Technologie i Narzędzia

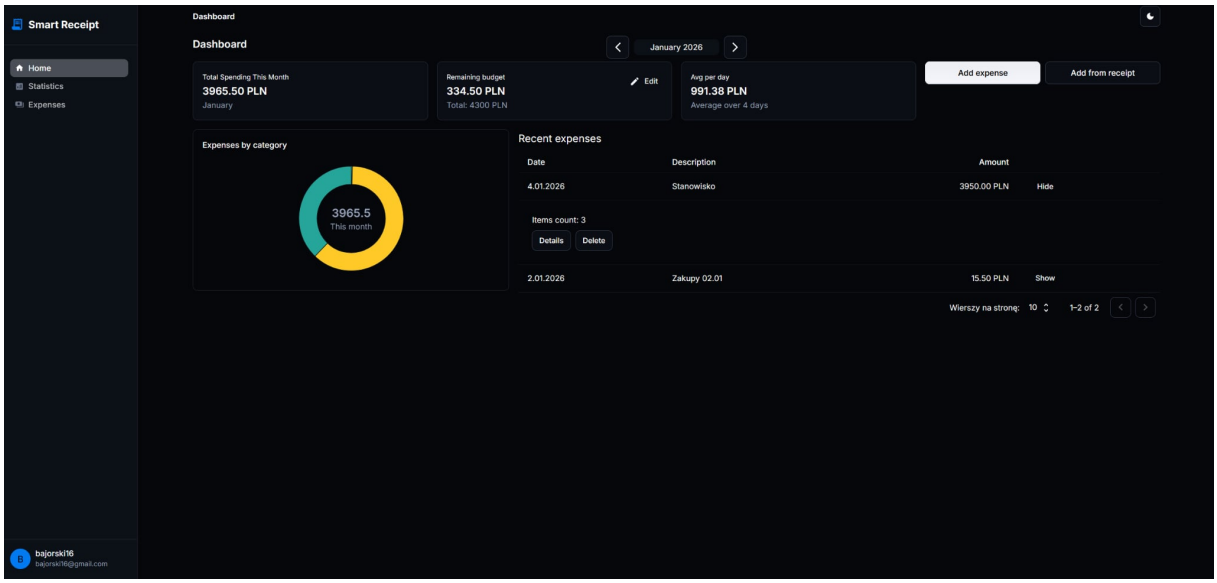
Interfejs użytkownika został zbudowany w oparciu o komponentową architekturę React, wspieraną przez system typowania TypeScript. Wybór ten gwarantuje spójność danych między backendem a frontendem, szczególnie dzięki automatycznemu generowaniu typów z kontraktu OpenAPI.

Kluczowe biblioteki wykorzystane w projekcie to:

- **Next.js (App Router):** Framework React zapewniający routing, optymalizację wydajności oraz renderowanie hybrydowe (Server Components + Client Components).
- **Material UI (MUI):** Biblioteka gotowych komponentów graficznych implementująca wytyczne Google Material Design. Zapewnia estetyczny i spójny wygląd przycisków, formularzy oraz układów stron.
- **Chart.js / React-Chartjs-2:** Silnik do generowania interaktywnych wykresów, wykorzystywany do wizualizacji danych na Dashboardzie.
- **MUI X Data Grid:** Zaawansowana tabela danych umożliwiająca sortowanie, filtrowanie i paginację dużych zbiorów wydatków.
- **Axios:** Klient HTTP służący do komunikacji z API Backendowym. Został on obudowany w wygenerowaną warstwę abstrakcji (‘api-client‘), co upraszcza wywoływanie endpointów.

4.2 Opis kluczowych widoków aplikacji

Poniżej przedstawiono zrzuty ekranu z działającej aplikacji wraz z opisem funkcjonalności poszczególnych modułów.



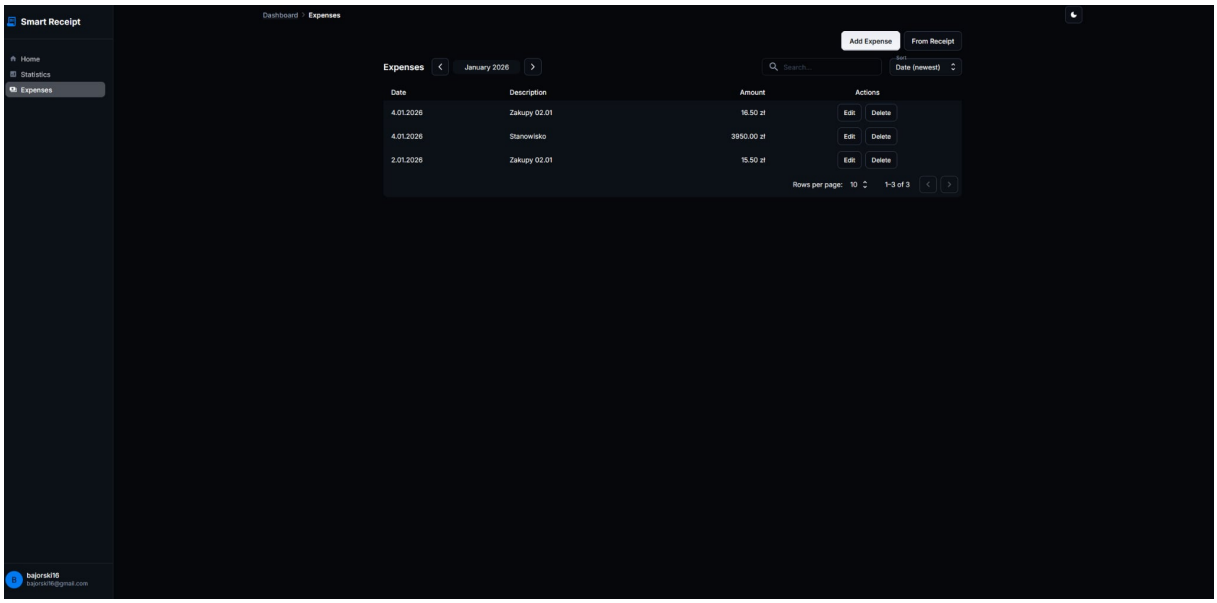
Rysunek 1: Główny pulpit nawigacyjny aplikacji Smart Receipt.

Rysunek 1 przedstawia główny panel analityczny. Jest to widok startowy użytkownika, agregujący najważniejsze informacje finansowe. Widoczne elementy to:

1. **Karty KPI (Key Performance Indicators):** Prezentują sumę wydatków w bieżącym okresie, pozostały budżet oraz średnią dniową transakcji. Karty te dają natychmiastowy wgląd w kondycję finansową.

2. **Wykres Trendów (Bar Chart):** Wizualizuje dynamikę wydatków w ujęciu czasowym, co pozwala na wykrycie anomalii lub sezonowości zakupów.

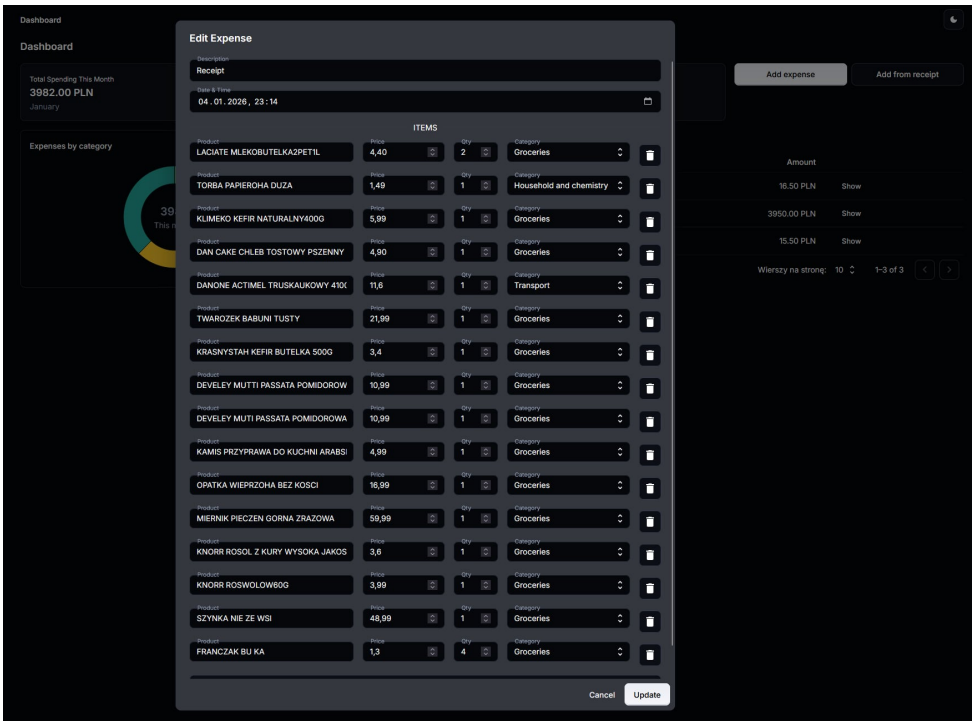
3. **Struktura Kategorii (Donut Chart):** Wykres pierścieniowy pokazujący udział poszczególnych kategorii w ogólnych wydatkach, co ułatwia identyfikację obszarów generujących największe koszty.



Rysunek 2: Widok szczegółowej listy wydatków z możliwością filtrowania.

Rysunek 2 prezentuje moduł zarządzania historią transakcji. Zastosowano tutaj komponent `DataGrid`, który oferuje zaawansowane możliwości operacyjne:

- **Filtrowanie i Sortowanie:** Użytkownik może przeszukiwać wydatki po dacie, kwocie, sklepie lub kategorii.
- **Paginacja:** Lista obsługuje stronicowanie po stronie serwera, co zapewnia płynność działania nawet przy dużej ilości rekordów.
- **Interakcja:** Kliknięcie w wiersz tabeli otwiera okno dialogowe ze szczegółami paragonu, w tym listą poszczególnych produktów pobieraną dynamicznie z backendu.



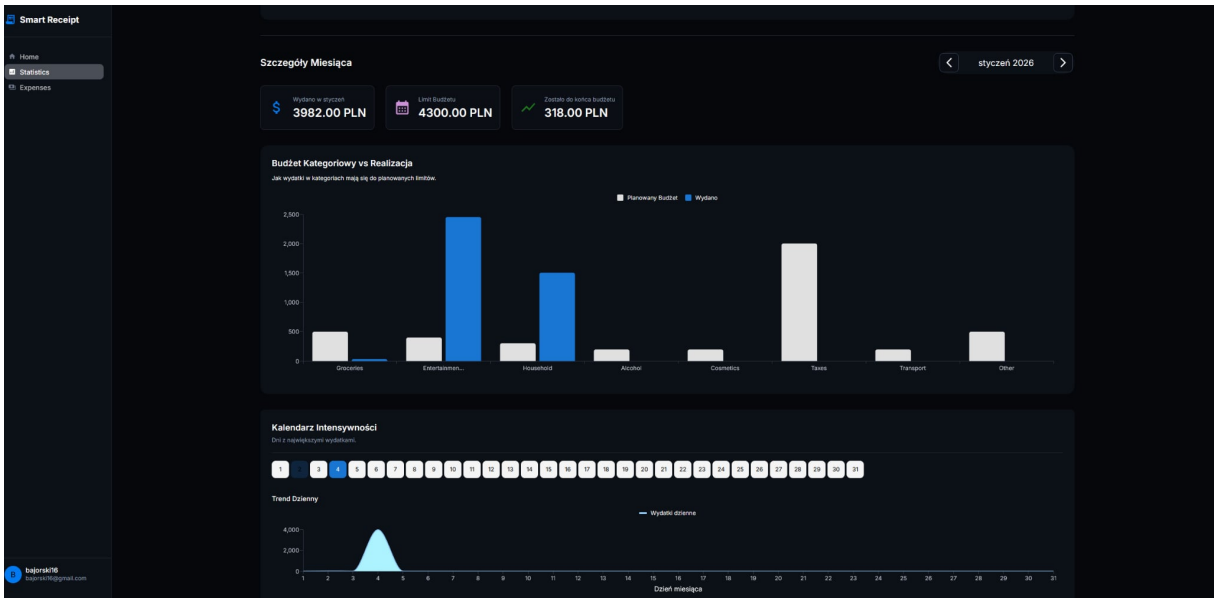
Rysunek 3: Okno dialogowe dodawania paragonu z wykorzystaniem modułu AI.

Kluczową funkcjonalnością systemu jest moduł OCR, przedstawiony na Rysunku 3. Komponent `ReceiptUploadDialog` umożliwia intuicyjne wprowadzenie danych:

1. **Upload Pliku:** Użytkownik może przeciągnąć zdjęcie paragonu lub wybrać je z dysku.

2. **Podgląd i Przetwarzanie:** Aplikacja wyświetla miniaturę zdjęcia i wysyła je do analizy AI. W trakcie przetwarzania wyświetlany jest animowany loader, informujący o postępie prac.

3. **Weryfikacja:** Po otrzymaniu odpowiedzi z AI, formularz jest automatycznie wypełniany rozpoznanymi danymi (Sklep, Data, Pozycje). Użytkownik ma możliwość ręcznej korekty przed ostatecznym zapisem w bazie.



Rysunek 4: Widok szczegółowej analizy wydatków w wybranym miesiącu.

Rysunek 4 przedstawia zaawansowany widok analityczny dedykowany dla konkretnego miesiąca. Jest to kluczowe narzędzie do bieżącej kontroli dyscypliny finansowej. Na interfejs składają się trzy główne sekcje:

1. **Panel KPI i Nawigacja:** W górnej części widoku, analogicznie do Dashboardu głównego, znajdują się karty podsumowujące (`StatCard`): „Wydano”, „Budżet” oraz „Pozostało”. Użytkownik ma możliwość płynnego przełączania się między miesiącami za pomocą selektora daty, co pozwala na szybkie porównywanie okresów rozliczeniowych.

2. **Analiza Realizacji Budżetu (Wykres Kategorii):** Centralny wykres prezentuje zestawienie wydatków rzeczywistych w odniesieniu do zaplanowanego budżetu dla poszczególnych kategorii. Dane te pochodzą z agregacji wydatków zestawionej z limitami zdefiniowanymi w encji `MonthlyBudgetEntity`. Wizualizacja ta pozwala natychmiast zauważyć przekroczenia budżetu w konkretnych obszarach.

3. **Kalendarz Intensywności Wydatków:** W dolnej części ekranu znajduje się innowacyjny moduł kalendarza, który wizualizuje intensywność wydatków w poszczególnych dniach miesiąca. Dni z dużą liczbą transakcji lub wysokimi kwotami są wyróżnione kolorystycznie. Pozwala to użytkownikowi na identyfikację swoich nawyków zakupowych (np. tendencja do dużych zakupów w weekendy) i lepsze planowanie przepływów pieniężnych w czasie.

5 Podsumowanie i wnioski projektowe

Realizacja systemu Smart Receipt pozwoliła na stworzenie kompletnego, nowoczesnego rozwiązania do zarządzania finansami osobistymi. Połączenie wydajnego backendu w Javie, reaktywnego frontendu w Next.js oraz innowacyjnego modułu AI opartego na modelu Llama 3.1 pozwoliło spełnić główne założenia projektowe. Aplikacja nie tylko automatyzuje żmudny proces wprowadzania danych, ale również dostarcza wartościowej wiedzy analitycznej, wspierając użytkowników w podejmowaniu lepszych decyzji finansowych.

Projekt ten był jednak również wyzwaniem, na które zespół musiał zmierzyć się z szeregiem wyzwań architektonicznych i sprzętowych.

5.1 Analiza wydajności warstwy danych

Jednym z najciekawszych problemów inżynierskich, napotkanych w trakcie prac, była kwestia wyboru środowiska uruchomieniowego dla bazy danych. W toku testów wydajnościowych zaobserwowano istotną zależność między lokalizacją bazy danych a czasem odpowiedzi aplikacji (Latency), szczególnie w kontekście wolumenu przesyłanych danych.

5.1.1 Lokalna instancja vs Usługa chmurowa (DBaaS)

Początkowe testy prowadzone na lokalnej instancji bazy danych wykazały natychmiastowy czas dostępu. Dla małych zbiorów danych (pojedyncze rekordy użytkowników, proste listy wydatków), lokalne rozwiązanie okazywało się bezkonkurencyjne pod względem responsywności interfejsu.

Wraz z decyzją o migracji do rozwiązania chmurowego (Database-as-a-Service, np. MongoDB Atlas lub Aiven Postgres), zespół zderzył się z problemem opóźnień sieciowych.

- **Problem „małych danych”:** W przypadku operacji na niewielkich ilościach danych, narzut czasowy związany z transmisją sieciową stawał się wąskim gardłem. Jeśli serwer bazy danych zlokalizowany był w odległym regionie (np. Singapur), fizyczna odległość powodowała, że prosty zapis wydatku trwał zauważalnie dłużej niż na maszynie lokalnej. Czas potrzebny na „podróż” pakietu danych (RTT - Round Trip Time) przewyższał czas samego wykonania zapytania SQL.
- **Skalowalność:** Mimo początkowych opóźnień, rozwiązanie chmurowe zyskuje przewagę przy rosnącej skali systemu. Usługi takie zapewniają automatyczne skalowanie, replikację i backupy, co w środowisku lokalnym wymagałoby skomplikowanej konfiguracji i utrzymania.

Wnioskiem z tego doświadczenia jest konieczność świadomego doboru regionu chmurowego (jak najbliżej użytkownika końcowego) oraz stosowanie mechanizmów cache'owania po stronie backendu, aby zminimalizować liczbę zapytań do odległej bazy.

5.2 Inne napotkane wyzwania techniczne

Oprócz kwestii bazodanowych, w trakcie cyklu wytwórczego rozwiązano szereg innych problemów:

1. **Wymagania sprzętowe modułu AI:** Uruchomienie modelu językowego Llama 3.1 8B stanowiło wyzwanie dla standardowych komputerów developerskich. Początkowo napotkano problemy z brakiem pamięci VRAM oraz instrukcjami procesora

(błędy typu *Exit 132* na starszych CPU). Rozwiązaniem okazało się wdrożenie trybu hybrydowego (*GPU Offloading*), który dynamicznie dzieli warstwy modelu między kartę graficzną a pamięć RAM, umożliwiając działanie AI nawet na kartach z 4 GB VRAM.

2. **Spójność kontraktów API:** Równoległa praca nad frontendem i backendem rodziła ryzyko rozbieżności w definicjach obiektów JSON. Ręczna aktualizacja typów TypeScript przy każdej zmianie w Javie była podatna na błędy ludzkie. Problem ten wyeliminowano poprzez rygorystyczne stosowanie podejścia **API-First**. Plik `openapi.yaml` stał się jedynym źródłem prawdy, a generowanie kodu (zarówno modeli DTO, jak i klientów frontendowych) zostało w pełni zautomatyzowane w procesie budowania (Gradle).

3. **Jakość danych z OCR:** Surowe dane z biblioteki PaddleOCR często zawierały szum informacyjny (np. numery NIP, adresy sklepów interpretowane jako produkty). Wymusiło to stworzenie zaawansowanego modułu *Pre-processingu*, opartego na wyrażeniach regularnych i czarnych listach słów kluczowych, przed przekazaniem tekstu do analizy przez LLM.

5.3 Wnioski końcowe

Projekt Smart Receipt udowodnił, że wykorzystanie lokalnych modeli LLM w aplikacjach użytkowych jest możliwe i efektywne, pod warunkiem odpowiedniej optymalizacji. Doświadczenia zdobyte przy konfiguracji infrastruktury hybrydowej (lokalne AI + chmurowa baza danych) stanowią solidny fundament pod dalszy rozwój systemu, np. o wersję mobilną czy integrację z bankowością.