



POLSKO-JAPOŃSKA AKADEMIA
TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Maciej Niszczoła

Nr albumu 32679

**Bezpieczny i elastyczny system zdalnej konfiguracji i
monitorowania dla aplikacji mobilnych**

Praca magisterska,

promotor:
dr inż. Mariusz Trzaska,
profesor PJATK

Warszawa, Czerwiec, 2026

Streszczenie

Praca podejmuje problematykę bezpiecznego i elastycznego zarządzania cyklem życia funkcjonalności w aplikacjach mobilnych funkcjonujących w środowisku o zerowym zaufaniu (ang. Zero Trust). Tradycyjne platformy zdalnej konfiguracji weryfikują tożsamość klientów za pomocą statycznych kluczy API, co ze względu na specyfikę systemów mobilnych czyni je wysoce podatnymi na inżynierię wsteczną, wycieki danych oraz ataki typu Man-in-the-Middle.

Głównym celem pracy było zaprojektowanie, zaimplementowanie i poddanie ewaluacji prototypu autorskiego systemu zdalnej konfiguracji, który eliminuje zidentyfikowane luki bezpieczeństwa poprzez oparcie autoryzacji na kryptograficznej, sprzętowej atestacji urządzeń. W ramach prac inżynierskich wytworzono rozproszone środowisko składające się z czterech głównych komponentów: serwera dystrybucyjnego zbudowanego w oparciu o framework Spring Boot, analitycznego panelu administracyjnego klasy SPA (Single Page Application) w technologii React/TypeScript, oraz dwóch dedykowanych bibliotek (SDK) wraz z aplikacjami referencyjnymi dla platform iOS (Swift) i Android (Kotlin).

Innowacyjność rozwiązania polega na integracji natywnych interfejsów Apple App Attest oraz Google Play Integrity z procesem uwierzytelniania, co uniemożliwia sfałszowanie zapytań o konfigurację przez emulatorzy, farmy botów czy skompromitowane urządzenia. Ponadto system implementuje mechanizm Server-Sent Events (SSE), zapewniając subsekundową latencję w dystrybucji flag funkcji, co pozwala na bezpieczną realizację wyłączników awaryjnych (kill-switch) i rzetelną orkiestrację testów A/B. Wyniki pracy udowadniają, że zastosowanie atestacji sprzętowej w architekturze Backend-for-Frontend pozwala na bezprecedensowe zabezpieczenie warstwy analityczno-konfiguracyjnej, redefiniując standardy bezpiecznego wdrażania i wydawania oprogramowania mobilnego.

Słowa kluczowe:

zdalna konfiguracja, flagi funkcji, testy A/B, atestacja sprzętowa, Zero Trust, bezpieczeństwo aplikacji mobilnych, Server-Sent Events, App Attest, Play Integrity.

Spis treści

1.	WSTĘP.....	5
1.1.	Kontekst zdalnej konfiguracji aplikacji mobilnych.....	5
1.2.	Cel pracy.....	6
1.3.	Rozwiązanie przyjęte w pracy.....	7
1.4.	Rezultaty pracy.....	8
1.4.1	Aspekt teoretyczny i badawczy.....	8
1.4.2	Wymiar praktyczny.....	8
1.4.3	Znaczenie rozwojowe i skalowalność.....	8
1.5.	Organizacja pracy.....	8
2.	ROLA SYSTEMÓW ZDALNEJ KONFIGURACJI W WYTWARZANIU OPROGRAMOWANIA	10
2.1.	Definicje.....	10
2.2.	Przegląd i ewaluacja istniejących rozwiązań od dostawców chmurowych	11
2.2.1	Firestore Remote Config.....	11
2.2.2	AWS AppConfig.....	13
2.2.3	Azure App Configuration.....	14
2.2.4	Wnioski z analizy rozwiązań dostawców chmurowych	14
2.3.	Rozwiązania dostawców dedykowanych, zamkniętych rozwiązań SaaS	14
2.3.1	LaunchDarkly.....	14
2.3.2	ConfigCat	15
2.3.3	Optimizely.....	15
2.3.4	Wnioski z analizy rozwiązań SaaS.....	16
2.4.	Platformy o otwartym kodzie źródłowym.....	16
2.4.1	Unleash.....	17
2.4.2	Flagsmith.....	17
2.4.3	FeatBit	18
2.4.4	Wnioski z analizy rozwiązań o otwartym kodzie źródłowym.....	18
2.5.	Porównanie możliwości platform i identyfikacja luk bezpieczeństwa	19
2.6.	Sposoby zarządzania sekretami w aplikacjach mobilnych.....	20
2.6.1	Metody Statyczne	20
2.6.2	Bezpieczne magazyny sprzętowe – Secure Enclave w iOS.....	22
2.6.3	Bezpieczne magazyny sprzętowe – TEE i StrongBox w Android	24
2.6.4	Bezpieczeństwo w tranzycie.....	25
2.6.5	Architektura Backend-for-Frontend.....	26
2.6.6	Atestacja na platformie Android – Play Integrity API	27
2.6.7	Atestacja na platformie iOS – DeviceCheck i AppAttest.....	29
2.7.	Sposoby monitorowania użycia i testowania funkcjonalności w aplikacjach mobilnych.....	31
2.8.	Pola do innowacji - unikalne cechy proponowanego rozwiązania.....	33
3.	PROPOZYCJA BEZPIECZNEGO SYSTEMU ZDALNEJ KONFIGURACJI	35
3.1.	Projekt funkcjonalny i przypadki użycia.....	35
3.1.1	Modelowanie procesu zarządzania aplikacjami	36
3.1.2	Modelowanie procesu zarządzania funkcjonalnościami	37
3.1.3	Modelowanie sterowania testami A/B.....	38
3.2.	Architektura systemu i przepływ danych.....	39
3.3.	Mechanizmy bezpieczeństwa.....	43
3.3.1	Uwierzytelnienie - systemy weryfikacji integralności sprzętowej	43
3.3.2	Autoryzacja.....	43
3.3.3	Kryptografia	43
4.	OPIS NARZĘDZI STOSOWANYCH W PRACY	44
4.1.	Narzędzia backendowe.....	44

4.1.1	Frameworki serwerowe	44
4.1.2	Technologie bazodanowe	44
4.1.3	Protokoły komunikacyjne	45
4.2.	Technologie frontendowe.....	45
4.3.	Technologie mobilne.....	46
4.3.1	Technologie iOS	46
4.3.2	Technologie Android	46
5.	PROTOTYP SYSTEMU ZDALNEJ KONFIGURACJI I MONITOROWANIA	47
5.1.	Implementacja serwera oraz mechanizmu natychmiastowej propagacji zmian.....	47
5.1.1	Warstwa przechowywania danych	48
5.1.2	Warstwa logiki biznesowej	50
5.1.3	Warstwa kontrolerów	50
5.2.	Panel administratora	52
5.3.	Integracja na platformie iOS	57
5.3.1	Aplikacja testowa.....	57
5.3.2	Implementacja dedykowanego SDK	58
5.4.	Integracja na platformie Android	59
5.4.1	Aplikacja testowa.....	60
5.4.2	Implementacja dedykowanego SDK	60
5.5.	Walidacja rozwiązania	62
5.5.1	Weryfikacja wyłącznika awaryjnego (SSE)	62
5.5.2	Walidacja mechanizmów bezpieczeństwa	64
5.5.3	Test spójności deterministycznej konfiguracji.....	65
6.	PROBLEMY I WYZWANIA ZWIĄZANE Z REALIZOWANYM SYSTEMEM.....	67
6.1.	Ograniczenia komunikacji w tle a funkcjonalność natychmiastowej propagacji zmian	67
6.2.	Aspekty prywatności i regulacje prawne w testach A/B	67
6.3.	Zależność wobec zewnętrznych usług	68
6.4.	Refaktoryzacja kodu serwera	68
6.4.1	Optymalizacja warstwy transportowej	69
6.4.2	Implementacja szyny zdarzeń (event bus).....	69
6.4.3	Dekompozycja połączeń Serwis-Repozytorium	69
6.4.4	Eliminacja narzutu ORM.....	69
6.5.	Ograniczenia procesu programistycznego i zapewnienia jakości	69
7.	PODSUMOWANIE.....	71
	PRACE CYTOWANE.....	72

1. Wstęp

Niniejszy rozdział stanowi ogólne wprowadzenie do problematyki pracy. Przedstawiono w nim szeroki kontekst biznesowy i technologiczny związany z procesami wdrażania i wydawania oprogramowania wraz z wpływem zdalnej konfiguracji na tenże. Zdefiniowano także główny cel oraz zakres podjętych działań badawczo-inżynierskich. Ponadto, skrótowo omówiono zaproponowane rozwiązanie, uzyskane rezultaty w postaci zintegrowanego prototypu systemu oraz organizację struktury całej pracy.

1.1. Kontekst zdalnej konfiguracji aplikacji mobilnych

Wdrożenie – słowo to znajduje swoje specjalne miejsce w umyśle każdego inżyniera oprogramowania od samego początku ewolucji systemów rozproszonych. Godziny poświęcone na sukcesywną integrację zmian. Przenoszenie ich na środowisko produkcyjne. Oczekiwanie na wyniki testów kanarkowych bądź dymnych. Poinformowanie reszty zespołu o sukcesie – okres wzmoczonego monitorowania telemetrii w oczekiwaniu na pierwsze reakcje klientów – a następnego ranka, trzask zgłoszonych incydentów. Pełna gama niepewności – od odczuć użytkowników końcowych aż do obciążenia infrastruktury w obliczu nowych funkcjonalności. Cały ten proces jest wyjątkowo stresogenny dla każdego zaangażowanego – inżynierów odpowiedzialnych za wykonanie, ich menedżera oraz pozostałych interesariuszy, np. pracowników działu zarządzania produktem lub marketingu.

Z czego wynika taki stan rzeczy? W praktyce inżynierskiej można wyróżnić następujące wektory błędów [1], [2], [3]:

- **Warstwa programistyczna** – implementacja naznaczona błędami,
- **Warstwa infrastrukturalna** – niepewność zachowania implementacji na infrastrukturze produkcyjnej wobec obciążenia użytkowników,
- **Warstwa doświadczeń użytkownika** – niepewność dotycząca wpływu zmiany na użytkownika,
- **Warstwa zarządzania produktem lub marketingu** – presja dostarczenia funkcjonalności wynikająca z komunikacji poprzedzającej wdrożenie (w relacji B2C) lub z założeń kontraktowych (w relacji B2B).

Każde z wymienionych ryzyk ma swoje szeroko akceptowane drogi mitygacji. Warstwa programistyczna może być testowana automatycznie bądź manualnie – rękami testerów lub użytkowników w trakcie testów akceptacyjnych. Warstwa infrastrukturalna może być weryfikowana w przypadku odpowiedniej dystrybucji ruchu użytkowników – jeśli jest on dopasowany skalą do infrastruktury w sposób pozwalający na zachowanie bezpiecznych przedziałów obciążenia. Warstwa doświadczeń użytkownika może osiągnąć znaczącą pewność słuszności swoich rozwiązań poprzez ogólne testy akceptacyjne oraz badanie reakcji użytkowników na konkretne elementy systemu, np. poprzez testy A/B. Warstwa zarządzania produktem lub marketingu wymaga zgrania zespołów technicznych w sposób pozwalający na wydanie danych funkcjonalności w konkretnym przedziale czasowym, bez przejmowania się technicznymi aspektami tego procesu [4].

Ryzyka te można więc podzielić na dwie kategorie [1], [3]:

- **Ryzyka wdrożenia** – ściśle powiązane z techniczną realizacją funkcjonalności. Niezależne od wydania,
- **Ryzyka wydania** – powiązane z utylitarnym lub biznesowym uzyskiem z implementacji poszczególnych funkcjonalności. Zależne od wdrożenia.

W związku z przedstawionymi zależnościami, praktyką zgodną z pryncypiami DevOps jest rozgraniczenie wdrożenia od wydania, aby umożliwić maksymalną elastyczność zarówno w pracy zespołu programistycznego jak i produktowego. Mechanizm ten wymaga wdrożenia zmian na środowisko produkcyjne w ukryciu przed użytkownikiem końcowym – a następnie wydanie ich za pomocą zdalnej konfiguracji. Praktyka ta w wersji minimalnej może być realizowana plikami konfiguracyjnymi (np. JSON lub XML), lecz dojrzałe organizacje i projekty skłaniają się ku użyciu dedykowanych narzędzi do zarządzania funkcjonalnościami – zazwyczaj dostępnymi w formie SaaS (Firebase Remote Config [5], LaunchDarkly [6]) [2]. Organizacje o wyjątkowo rozbudowanych działach inżynieryjnych skłaniają się z kolei ku narzędziom wytworzonym wewnętrznie – np. Gatekeeper w Facebooku, Feature API w Etsy lub Archaius w Netflixie [1], [2], [7].

Wymienione rozwiązania – a w szczególności te dostępne jako usługa – zachowują znaczny stopień generyczności. Podejście takie pozwala potencjalnym klientom na zastosowanie ich w pełnym przekroju projektów opierających się na wielu platformach – jednakże wymusza kompromisy w następujących dziedzinach:

- **Bezpieczeństwo** – rozwiązania bardzo często traktują wszelkie wartości konfiguracyjne jako wartości publiczne, udostępniając je w ramach niezabezpieczonych zasobów jako otwarty tekst [5], [8],
- **Czas reakcji** – typowe rozwiązania skupiają się na pasywnym modelu dostarczania zawartości, uzależniając pobieranie konfiguracji całkowicie od procesu odpytywania prowadzonego przez klienta [5],
- **Doświadczenia użytkownika i/lub programisty** – dostępne rozwiązania są uciążliwe w użyciu w przypadku wielośrodowiskowych aplikacji mobilnych. Ich interfejs i logika zachowują wymaganą generyczność, obniżając łatwość obsługi,
- **Powiązanie elementów biznesowych z technicznymi** – rozwiązania dostawców chmurowych skupiają się na technicznej zdolności do dostarczenia pewnej konfiguracji, lecz zupełnie ignorują wartość biznesową, którą można za ich pośrednictwem dostarczyć (np. poprzez testy A/B).

1.2. Cel pracy

Mając na uwadze przedstawione wyzwania związane z procesem wdrażania i wydawania oprogramowania oraz wyraźne niedoskonałości istniejących systemów na rynku, głównym celem niniejszej pracy jest dogłębna analiza i porównanie istniejących rozwiązań, zaprojektowanie, implementacja oraz ewaluacja bezpiecznego i elastycznego systemu zdalnej konfiguracji, dedykowanego aplikacjom mobilnym.

Zaproponowane rozwiązanie ma stanowić odpowiedź na kompromisy wymuszane przez generyczne usługi dostawców chmurowych, realizując następujące cele:

- **Zapewnienie wysokiego poziomu bezpieczeństwa:** Stworzenie mechanizmu, który zagwarantuje, że dostęp do krytycznej konfiguracji będą miały wyłącznie w pełni autoryzowane i niezmodyfikowane instancje aplikacji klienckich,

- **Minimalizacja czasu reakcji systemu:** Wdrożenie mechanizmu natychmiastowej propagacji zmian, pozwalającego na użycie trybu awaryjnego wyłączania funkcjonalności (tzw. *Kill-switch*),
- **Poprawa doświadczeń programistycznych i administracyjnych:** Skonstruowanie intuicyjnego interfejsu CMS oraz dedykowanych bibliotek (SDK) ułatwiających bezproblemową integrację po stronie aplikacji klienckich, co zredukuje narzut operacyjny w wieloplatformowych projektach mobilnych,
- **Zintegrowanie wartości biznesowej z techniczną:** Wyposażenie systemu w moduły analityczne umożliwiające realizację testów A/B oraz zbieranie metryk użycia, co pozwoli na bezpośrednie badanie wpływu wydania funkcjonalności na zachowanie użytkowników końcowych.

1.3. Rozwiązanie przyjęte w pracy

Celem rozwiązania wyżej zdefiniowanych problemów, w niniejszej pracy zaprojektowano i zaimplementowano prototyp autorskiego systemu zdalnej konfiguracji, inspirowany wzorcem architektonicznym Backend-for-Frontend oraz paradygmatem zerowego zaufania. Struktura proponowanego rozwiązania dzieli się na cztery komponenty:

- Serwer Zdalnej Konfiguracji,
- Konsolę Administracyjną w formie aplikacji webowej,
- Natywną Aplikację Mobilną iOS wraz z dedykowaną biblioteką,
- Natywną Aplikację Mobilną Android wraz z dedykowaną biblioteką.

Warstwę serwerową zrealizowano przy użyciu języka Java wraz z środowiskiem *Spring Boot*, wykorzystując w szczególności moduły Spring Web oraz Spring Security. Za bezstanowe i relacyjne przechowywanie struktur drzew decyzyjnych dystrybucji flag odpowiada baza H2 – z założeniem użycia bazy PostgreSQL w przypadku uruchomienia produkcyjnego. Połączenie serwera z aplikacjami oparto na hybrydowym modelu: realizowane są zarówno żądania asynchroniczne przez REST API, jak i strumieniowe połączenia z pomocą *Server-Sent Events* w celu jednokierunkowej, natychmiastowej propagacji aktualizowanych konfiguracji do wszystkich aplikacji klienckich.

Konsola administracyjna służąca zarządzaniu cyklem życia flag konfiguracyjnych oraz ewaluacji eksperymentów została zbudowana jako aplikacja typu *Single Page Application*. Podstawą technologiczną tego modułu jest język TypeScript, użyty w ramach biblioteki React w połączeniu ze środowiskiem budowania Vite. Na potrzeby realizacji atrakcyjnego graficznie interfejsu, zaadoptowano komponenty biblioteki TailAdmin, wraz z dedykowanymi stylami zarządzanymi przy użyciu narzędzia Tailwind CSS. Reprezentacja wyników testów A/B została zrealizowana przy pomocy komponentu ApexCharts.

Warstwę klientów mobilnych oparto w całości na technologiach natywnych, co pozwoliło na wierną i głęboką integrację ze sprzętowymi modułami kryptograficznymi. Klient wraz z biblioteką zdalnej konfiguracji dla systemu iOS został zrealizowany w języku Swift – używając przy tym deklaratywnej kompozycji interfejsu graficznego przez SwiftUI. Mechanika bezpieczeństwa (atestacji) została oparta o mechanizm DeviceCheck i AppAttest, współpracujące z układem sprzętowej, bezpiecznej enklawy urządzenia.

Aplikacja oraz biblioteka na system Android napisane zostały w języku Kotlin, a interfejsy graficzne zrealizowane zostały przy użyciu Jetpack Compose. Zapewnienie bezpieczeństwa odbywa się poprzez delegację weryfikacji integralności środowiska do usługi Play Integrity API, zintegrowanej ze sprzętowym komponentem Keystore.

1.4. Rezultaty pracy

Kluczowym artefaktem-rezultatem niniejszej pracy jest zaprojektowanie, zaimplementowanie oraz weryfikacja prototypu systemu zdalnej konfiguracji dla aplikacji mobilnych, rozwiązującego problem asymetrii zaufania poprzez użycie natywnych mechanizmów atestacji – oferując unikalny poziom bezpieczeństwa w porównaniu do istniejących na rynku rozwiązań. Osiągnięte wyniki można scharakteryzować w trzech wymiarach.

1.4.1 Aspekt teoretyczny i badawczy

Wymiar teoretyczny polega na systematyzacji wiedzy w obszarze obecnie istniejących systemów konfiguracji zdalnej oraz analizy ich bezpieczeństwa w kontekście klientów mobilnych. Porównywane są najpopularniejsze rozwiązania o różnorodnym pochodzeniu (chmura, SaaS, otwarte źródło) – w wymiarze nie tylko funkcjonalnym, ale również biznesowym, uwzględniając kompleksowość operacyjną adopcji systemu. Przeprowadzona analiza dowodzi niewystarczalności klasycznych metod ochrony sekretów aplikacyjnych w obliczu środowiska o zerowym zaufaniu, jakim jest każde urządzenie mobilne. Poprzez identyfikację zjawiska problemu pierwszego sekretu nakreśla znaczenie zabezpieczenia komunikacji klientów mobilnych z serwerami z wykorzystaniem komponentów sprzętowych od zaufanej trzeciej strony – czyli wytwórców urządzeń.

1.4.2 Wymiar praktyczny

Po uprzedniej identyfikacji luk oraz najpopularniejszych mechanik istniejących systemów zdalnej konfiguracji, aspekt praktyczny skupia się na dostarczeniu funkcjonalnego prototypu ekosystemu oferującego nową jakość w tej domenie. W ramach prac zrealizowano wszystkie komponenty kompletnego systemu zdalnej konfiguracji: serwer, portal administracyjny oraz biblioteki umożliwiające sprawną integrację na urządzeniach mobilnych. Ponadto, przygotowane zostały dwie przykładowe aplikacje mobilne demonstrujące działanie systemu wraz z bibliotekami – odpowiednio dla systemów iOS i Android. Dzięki realizacji prototypu możliwa jest dokładna ewaluacja proponowanych usprawnień w wielu wymiarach – audytu bezpieczeństwa, operacji, wygody programisty oraz rzeczywistej realizacji potrzeb biznesowych.

1.4.3 Znaczenie rozwojowe i skalowalność

Zaprojektowana architektura stanowi elastyczny fundament, umożliwiający dalszą rozbudowę do realizacji większej ilości przypadków użycia – zarówno komponent serwerowy, jak i panel administracyjny zbudowane są w sposób modułarny. Potencjał samej rozbudowy systemu nie jest jednak tak istotny jak dowód ogólnego paradygmatu, który on realizuje – czyli budowę zaufania między klientem mobilnym a serwerem. Przy realizacji tej mechaniki i założeniu, że bezpieczeństwo procedur atestacji pozostaje nienaruszone, otwiera to ścieżkę do zupełnie nowych zastosowań systemów konfiguracji zdalnej – wliczając w to np. pośredniczenie w zarządzaniu wszelkimi sekretami wykorzystywanymi przez klientów mobilnych. Do dalszej ewaluacji tego założenia – prototyp jest realizowany na poziomie dostarczenia minimalnej wartości, gotowej do uruchomienia i pilotażu – lecz pomijając niektóre aspekty w pełni produkcyjnego systemu, takie jak np. kompletny model autoryzacyjny panelu administracyjnego.

1.5. Organizacja pracy

Niniejsza praca została podzielona na siedem rozdziałów, mających na celu przeprowadzenie czytelnika przez pełen proces inżynieryjny stojący za wynikowym prototypem systemu zdalnej konfiguracji.

Rozdział pierwszy stanowi wprowadzenie do ogólnej problematyki pracy i motywacji biznesowej zajmowania się tematem konfiguracji zdalnej jako przykładu systemu operującego w środowisku o zerowym zaufaniu.

Rozdział drugi zawiera dogłębną analizę obecnych na rynku rozwiązań – przyczyniając się do sformułowania kluczowej listy funkcjonalności podstawowego systemu zdalnej konfiguracji. Oprócz aspektów stricte funkcjonalnych bądź operacyjnych, traktuje on również współczesne podejście do kwestii bezpieczeństwa, omawiając najpopularniejsze praktyki wraz z ich lukami.

Rozdział trzeci podejmuje informacje zagregowane w ramach rozdziału drugiego i selektywnie przenosi je w dziedzinę powstającego prototypu. Znajduje się w nim omówienie kluczowych przypadków użycia wraz z zamodelowaniem ich w rozpatrywanej dziedzinie biznesowej. Ilustruje on pierwsze fazy klasycznego procesu inżynierii oprogramowania, którego artefaktem jest precyzyjny model architektury systemu wraz z planem implementacji.

Rozdział czwarty omawia z kolei narzędzia służące do realizacji uprzednio zaprojektowanego systemu, z wyszczególnieniem stosu technologicznego dla każdego z realizowanych komponentów: serwera, panelu administracyjnego i aplikacji mobilnych.

Rozdział piąty poświęcony został opisowi dokładnej implementacji prototypu, stanowiąc wzbogacenie artefaktów rozdziału trzeciego o szczegóły ściśle wynikające z zastosowania technologii opisanych w rozdziale czwartym. Przeprowadza on czytelnika przez całość rozwiązania, nadając kontekstu analizie mającej miejsce w następnym rozdziale.

Rozdział szósty skupia się na krytycznej ewaluacji zrealizowanego prototypu – identyfikacji jego zalet i wad, nawiązując częściowo do kryteriów rozpoznanych w rozdziale poświęconym stanowi sztuki. Zawiera on informację o wszelkich problemach badawczych i inżynierskich, które wynikły bądź mogą wynikać z założeń przyświecających realizacji prototypu.

Końcowy, siódmy rozdział podsumowuje treść i artefakty pracy w kontekście założonych celów oraz podkreśla ostateczne wnioski płynące z przeprowadzonych prac projektowych i implementacyjnych.

2. Rola systemów zdalnej konfiguracji w wytwarzaniu oprogramowania

Niniejszy rozdział poświęcono dokładnemu omówieniu kontekstu technologicznego oraz problematyki, którą traktuje praca – związanej z elastycznym i bezpiecznym zarządzaniem aplikacjami mobilnymi przy zachowaniu niskiej latencji. W poszczególnych podrozdziałach dokonano przeglądu stanu sztuki, analizując popularne rozwiązania oferowane w różnorodnych modelach – natywne systemy dostawców chmurowych, dedykowane platformy usługowe lub jako projekty społecznościowe. Celem tej analizy jest identyfikacja wad i niedoskonałości obecnych na rynku systemów – a następnie skomponowanie wymagań i unikalnej propozycji wartości systemu realizowanego w ramach niniejszej pracy.

2.1. Definicje

Zrozumienie zagadnień poruszanych w niniejszej pracy wymaga ujednolicenia zestawu pojęć. Poniżej zdefiniowano najważniejsze terminy domenowe będące w szerokim użyciu w kontekście systemów zdalnej konfiguracji.

Flaga funkcji (ang. *Feature Flag*) – zaimplementowany w kodzie źródłowym przełącznik, który umożliwia warunkowe ukrywanie, włączanie lub wyłączanie określonych funkcjonalności w czasie działania programu. Stan tego przełącznika ewaluowany jest dynamicznie, na podstawie konfiguracji pobranej z serwera zarządzającego funkcjami.

Testy A/B (ang. *A/B Testing*) – metoda badawcza w dziedzinie analityki produktowej, polegająca na jednoczesnym udostępnieniu dwóch lub więcej wariantów danej funkcjonalności różnym, wyizolowanym grupom użytkowników. Celem procesu jest gromadzenie i porównanie metryk użycia, aby zweryfikować który wariant optymalnie realizuje założone cele biznesowe.

Wdrożenie (ang. *Deployment*) – proces technicznej realizacji i przeniesienia zmian w kodzie źródłowym na środowisko produkcyjne, obejmujący ich integrację i weryfikację, przechodząc dalej w monitoring stabilności infrastruktury. Jest to proces ściśle technologiczny i wedle najnowocześniejszych praktyk wytwarzania oprogramowania – niezależny od momentu faktycznego udostępnienia implementowanej funkcjonalności użytkownikowi końcowemu.

Wydanie (ang. *Release*) – proces udostępnienia konkretnych funkcjonalności systemu użytkownikom końcowym w celu uzyskania założonych korzyści biznesowych lub użytkowniczych. Wydanie jest operacją zależną od uprzedniego wdrożenia technicznego – jednak dzięki zastosowaniu mechanizmów zdalnej konfiguracji może zostać od niego całkowicie odseparowane w czasie – zapewniając elastyczność w zarządzaniu produktem.

Wyłącznik awaryjny (ang. *Kill-switch*) – krytyczny wariant flagi funkcji, zaprojektowany z myślą o natychmiastowym przerwaniu działania wadliwego modułu lub logiki, w celu zapobieżenia dalszym szkodom – technicznym (np. poprzez wpływ na infrastrukturę) lub biznesowym (np. reputacji).

Zdalna konfiguracja (ang. *Remote Config*) – mechanizm architektury oprogramowania umożliwiający dynamiczną zmianę zachowania, parametrów oraz interfejsu użytkownika aplikacji klienckiej po jej instalacji na urządzeniu docelowym, bez konieczności publikowania i pobierania nowej wersji z kanałów dystrybucyjnych (np. Apple App Store, Google Play).

2.2. Przegląd i ewaluacja istniejących rozwiązań od dostawców chmurowych

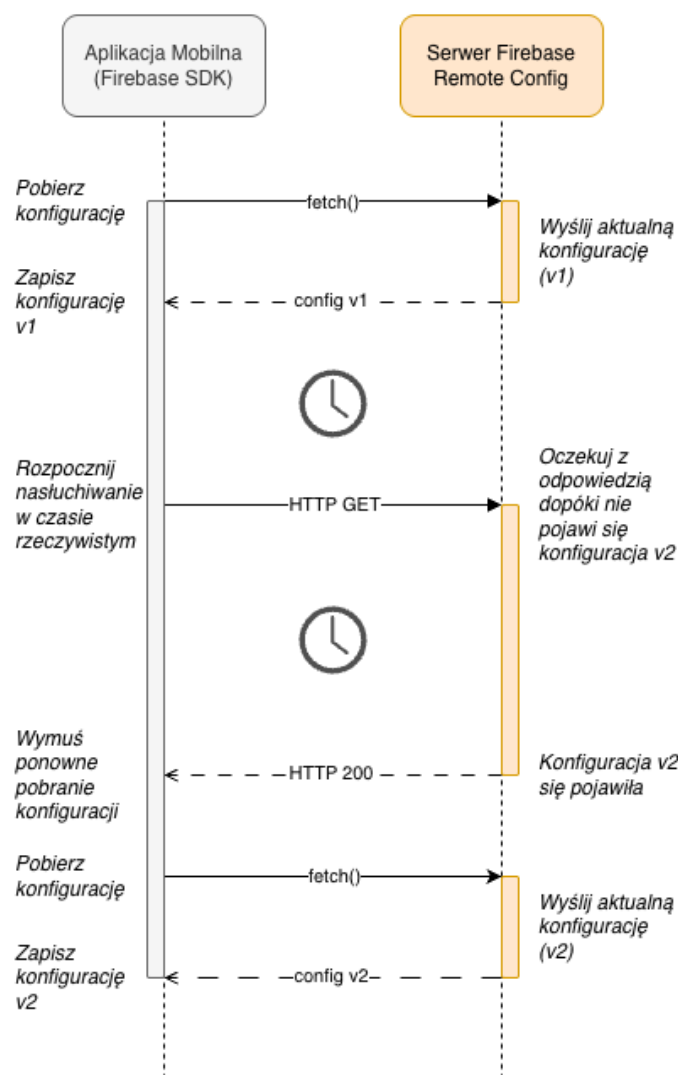
Wraz z wymaganiem elastyczności stawianym przez współczesne paradygmaty wytwarzania oprogramowania [3], na rynku oprogramowania pojawiło się wiele gotowych narzędzi zdalnej konfiguracji – realizujących filozofię rozdzielenia wdrożenia od wydania [1], [2]. W celu rzetelnej ewaluacji zastanego krajobrazu rozwiązań zadanej domeny, przeanalizowano wiodące platformy o różnorodnym rodowodzie – skupiając się na ich architekturze, bezpieczeństwie oraz możliwościach analitycznych.

Pierwszą z analizowanych grup stanowią usługi wchodzące w skład szerokich ekosystemów dostawców chmur obliczeniowych, skryte zazwyczaj pod płaszczem modelu *Backend-as-a-Service* (BaaS). Głównym założeniem architektonicznym tych rozwiązań jest centralizacja zarządzania i bezproblemowa integracja z pozostałymi komponentami infrastruktury danego dostawcy – ograniczając klasyczne bolączki adopcji fragmentarycznych rozwiązań dedykowanych, tj. zarządzanie dostępem i tożsamością administratorów takowych rozwiązań oraz widoczność kosztów. Obecność takich rozwiązań u wiodących dostawców przyczynia się do ograniczenia rozwoju tzw. „*Shadow IT*” w organizacji – gdzie pracownicy uciekają się ku rozwiązaniom spoza oficjalnego portfolio usług. W kontekście rozpatrywanych możliwości, rynek ten zdominowany jest przez trzy wiodące platformy: Firebase / Google Cloud, AWS (Amazon) oraz Azure (Microsoft).

2.2.1 Firebase Remote Config

Platforma Firebase, utrzymywana i rozwijana przez firmę Google od 2014 roku, powstała w celu dostarczenia programistom gotowych usług serwerowych, eliminując potrzebę budowania i utrzymywania dedykowanej infrastruktury – umożliwiając szybką implementację i wdrażanie nowych aplikacji mobilnych. Moduł zdalnej konfiguracji [5], udostępniony w 2016 roku, stanowi jedno z najpopularniejszych narzędzi do dynamicznego zarządzania parametrami aplikacji.

Integracja rozwiązania opiera się na dedykowanym pakiecie programistycznym (SDK) [5], który natywnie wspiera platformy iOS oraz Android. Mechanizm transportu danych początkowo opierał się wyłącznie na architekturze typu *pull* – wymuszając na kliencie cykliczne odpytywanie serwera za pomocą żądań HTTP. Relatywnie niedawno – w połowie 2023 roku rozszerzono możliwości systemu o wsparcie propagacji w czasie rzeczywistym, którego realizację przedstawia Rysunek 1. Funkcjonalność ta została zrealizowana poprzez tzw. Długie odpytywanie (ang. *Long polling*) – otwarcie kanału komunikacyjnego zapytaniem HTTP – i zamknięcie go w momencie wymuszenia zmiany [5].

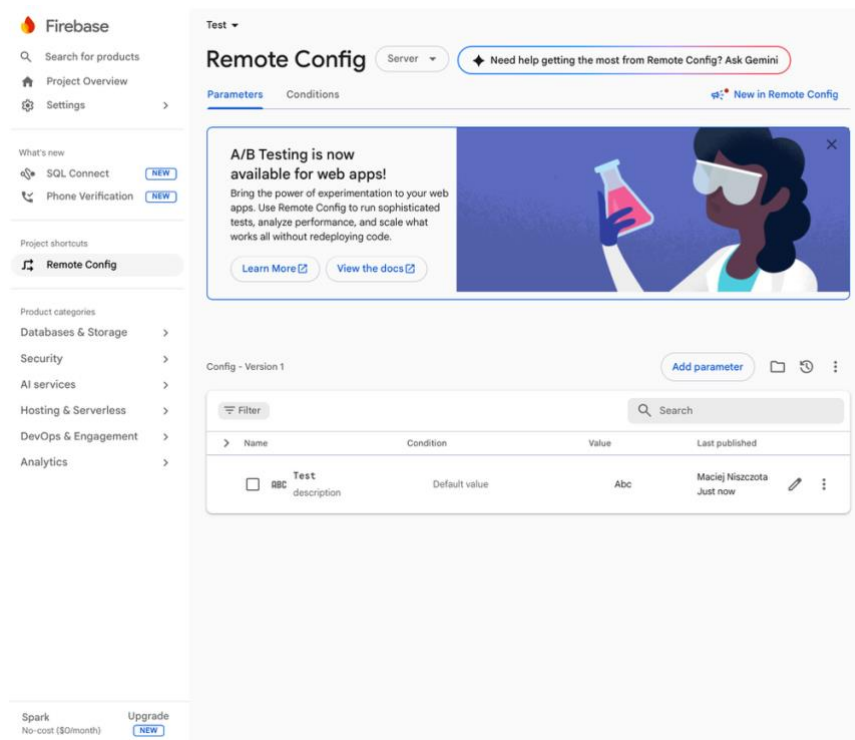


Rysunek 1. Sekwencja długiego odpytywania realizująca natychmiastową propagację konfiguracji w Firebase Remote Config

Firebase pozwala na znaczną elastyczność w przesyłaniu danych – możliwe jest wysłanie dowolnej struktury zgodnej z formatem JSON [5]. Niestety, pomimo tej zalety – sam system zarządzania flagami opiera się o płaską listę parametrów. Mechanizm ewaluacji warunków logicznych (np. kierowanie flagi do konkretnej grupy docelowej) realizowany jest oparciu o liniową listę priorytetów. Brakuje w nim zaawansowanych mechanizmów znanych z dedykowanych systemów SaaS, takich jak kaskadowe dziedziczenie ustawień [5], tworzenie wielościeżkowych drzew decyzyjnych czy definiowanie bezpośrednich zależności między poszczególnymi flagami w panelu konfiguracyjnym, które przedstawia Rysunek 2.

Jednakże najpoważniejszą wadą systemu dostarczanego przez Google jest jego model autoryzacji – potencjalnie dyskwalifikując go w przypadku aplikacji o podwyższonym rygorze bezpieczeństwa. Zgodnie z założeniami architektury Firebase Remote Config, usługa nie szyfruje poszczególnych zmiennych, a *endpointy* serwujące konfigurację są dostępne publicznie. Identyfikacja klienta opiera się wyłącznie na statycznym kluczu API oraz identyfikatorze projektu, które muszą zostać osadzone w kodzie źródłowym aplikacji klienckiej (w postaci plików `google-services.json` lub `GoogleService-Info.plist`) [5], [9]. Oznacza to, że przy zastosowaniu podstawowych technik inżynierii wstecznej, takiej

jak dekompilacja pliku APK / IPA, pozwala na bezproblemowe wyekstrahowanie poświadczeń [8] i odpytanie interfejsu API Firebase, pobierając wartości konfiguracyjne, serwowane otwartym tekstem. Tym samym, każdą wartość udostępnianą przez Firebase Remote Config należy traktować jako publiczną [5].



Rysunek 2. Panel konfiguracyjny flag funkcji w platformie Firebase

2.2.2 AWS AppConfig

Odmienne podejście architektoniczne, szczególnie w domenie bezpieczeństwa, prezentuje usługa AWS AppConfig, udostępniona przez firmę Amazon w 2019 roku. Została ona zaprojektowana jako skalowalne rozwiązanie korporacyjne uwzględniając zaawansowane mechaniki kontroli dostępu [10], [11].

Środowisko AWS nie wystawia wartości konfiguracyjnych jako publiczne zasoby – pobieranie konfiguracji przez klienta wymaga poprawnej autoryzacji w ekosystemie AWS IAM – stosując metodę podpisywania żądań AWS SigV4 [10]. Ekosystem ten wymusza również użycie dodatkowych warstw uwierzytelniających klienta – np. poprzez Amazon Cognito, które dokonuje tego poprzez rotowany *token* dostępowy, uniemożliwiając ekstrakcję poświadczeń z pliku binarnego. Same struktury danych konfiguracyjnych oferują znaczną elastyczność – mogą być przechowywane w formatach JSON, YAML lub płaski tekst – natywnie integrując się z takimi usługami jak Amazon S3 [11].

Niestety, AWS AppConfig charakteryzuje się pewnymi ograniczeniami w wymiarze biznesowym dostarczanych funkcjonalności – jest to narzędzie stricte techniczne i nie oferuje natywnego wsparcia dla analityki zachowań użytkowników [11]. Przeprowadzenie testów A/B z użyciem AppConfig byłoby możliwe tylko w przypadku wybudowania własnej logiki pomiarowej i aktywacyjnej. Dodatkowo, system nie oferuje funkcjonalności wyłącznika awaryjnego – rozszerzając lukę względem Firebase. Sam model rozliczeniowy platformy uzależnia koszty wprost od wolumenu zapytań [11] – co może być szczególną barierą dla prostych projektów, zwłaszcza w porównaniu do darmowego rozwiązania od Google [5].

2.2.3 Azure App Configuration

Oferotę gigantów chmurowych zamyka platforma firmy Microsoft, oferowana w ramach środowiska Azure od 2020. Stanowi ona hybrydę rozwiązań rynkowych, skupiając się na centralizacji rozproszonych parametrów konfiguracyjnych [12]. Azure App Configuration weryfikuje zapytania klientów po stronie serwera, integrując się z platformą tożsamości Entra ID – analogicznie do rozwiązania Amazona [12], [11]. Niestety, analogia zostaje zachowana też w przypadku biznesowego aspektu dostarczanych funkcjonalności – platforma całkowicie pomija zdolności, które pozwoliłyby na analizę doświadczeń użytkownika lub przeprowadzenie testów A/B [12].

2.2.4 Wnioski z analizy rozwiązań dostawców chmurowych

Rozwiązania wbudowane w ekosystemy globalnych dostawców chmurowych stanowią optymalny wybór dla zespołów poszukujących centralizacji infrastruktury i relatywnie niskiego poziomu wejścia – zarówno pod kątem technicznym, jak i formalnym – użycie dodatkowego modułu spośród już używanego ekosystemu zazwyczaj nie będzie wymagało przejścia ścieżki akceptacji, typowej dla dużych przedsiębiorstw. Ich największą zaletą jest bezproblemowa skalowalność i gotowa integracja z innymi usługami chmury (np. z analityką, magazynami danych, systemem IAM) [9], [5], [10] oraz niskie koszty (zakładając uprzednie użycie danej chmury w ramach firmy). Niestety, wygoda ta wymusza też pewne kompromisy – brakuje w nich zaawansowanych funkcjonalności i pełnego pokrycia logiki biznesowej – co jest argumentem za poszukiwaniem dedykowanych rozwiązań, dostępnych poza ekosystemami chmurowymi – jako oprogramowanie w modelu SaaS.

2.3. Rozwiązania dostawców dedykowanych, zamkniętych rozwiązań SaaS

Drugą istotną kategorią systemów zdalnej konfiguracji są platformy rozwijane w zamkniętym modelu usługowym SaaS. W przeciwieństwie do infrastrukturalnych rozwiązań oferowanych przez dostawców chmurowych, systemy te są projektowane ściśle z myślą o zarządzaniu cyklem życia funkcjonalności. Ich niezależność od wielkich ekosystemów pozwala na dostarczanie wysoce wyspecjalizowanych narzędzi wspierających zarówno inżynierów, jak i działy produktowe. Pośród rozbudowanego portfolio platform dostępnych na rynku, najbardziej wyróżniającymi się są LaunchDarkly, ConfigCat oraz Optimizely.

2.3.1 LaunchDarkly

Uruchomiona w 2014 roku platforma LaunchDarkly plasuje się jako jedno z najbardziej dojrzałych i zaawansowanych narzędzi w obszarze zdalnej konfiguracji. Jej architektura została zaprojektowana z myślą o minimalizacji opóźnień w propagacji zmian oraz pełnej synchronizacji konfiguracji między serwerami [6]. Traktując działanie w czasie rzeczywistym jako priorytet, cała platforma działa zgodnie z mechanizmem wyłącznika awaryjnego. Opiera ona komunikację kliencką na strumieniowaniu danych konfiguracyjnych w czasie rzeczywistym – wykorzystując do tego celu protokół SSE (*Server-Sent Events*) [6].

Istotną cechą jest również wyjątkowa elastyczność rozwiązania dzięki rozróżnieniu logiki ewaluacji reguł flag funkcji w zależności od tego, czy konfigurowany jest komponent serwerowy (np. serwis Spring Boot) czy komponent stricte kliencki (np. aplikacja mobilna). W przypadku ewaluacji flag dla serwera, zwracane jest pełne drzewo konfiguracyjne, z którego wybierany jest odpowiedni wariant [6]. Aplikacje klienckie otrzymują od razu dedykowany pakiet konfiguracyjny, częściowo ograniczając

ryzyko wycieków danych. Co więcej, platforma wyróżnia się kilkoma funkcjonalnościami wychodzącymi poza czysty zakres dostarczania zdalnych konfiguracji – oferuje na przykład moduł zarządzania i monitorowania aplikacji bazujących na użyciu modeli generatywnej sztucznej inteligencji [6]. Rygorystyczne podejście do architektury i zakres zdolności skrojony na miarę zarządzania produktem ma jednak swoje odzwierciedlenie w modelu biznesowym – LaunchDarkly jest rozwiązaniem wysoce skomercjalizowanym, de facto pozbawionym bezpłatnego planu zdolnego obsłużyć ruch na środowiskach produkcyjnych - co znacząco kontrastuje z podejściem Firebase.

2.3.2 ConfigCat

Zupełnie inne podejście do dystrybucji względem LaunchDarkly prezentuje również powstała w 2018, niezależna platforma ConfigCat. Głównym założeniem tego systemu jest dostarczenie lekkiego narzędzia o niskim progu wejścia. W kontekście aplikacji mobilnych (dla których udostępniono dedykowane, natywne SDK w językach Swift oraz Kotlin [13]), platforma opiera się w całości na dokonywaniu ewaluacji flag po stronie klienta [13]. Oznacza to, że aplikacja mobilna pobiera pełen zestaw reguł, a samo obliczenie wartości końcowej odbywa się lokalnie na urządzeniu użytkownika – co jest rozwiązaniem nie tylko nieminimalnym, wpływając na wydajność – ale po prostu niebezpiecznym. Plusem jest zabezpieczenie *endpointów*, wymagające unikalnych identyfikatorów przypisanych do danego SDK – choć wciąż podatnym na dekompilację kodu źródłowego klienta.

System ten, mimo szerokiej dostępności pełnej funkcjonalności już w ramach darmowego planu (ograniczonego jedynie wolumenem zapytań), wykazuje braki w zakresie elastyczności. Zgodnie z analizą dokumentacji platformy, ConfigCat nie wspiera skomplikowanych warunków logicznych ani mechanizmów dziedziczenia konfiguracji [13], co upodabnia go strukturalnie do ograniczeń prezentowanych przez rozwiązania dostawców chmurowych.

2.3.3 Optimizely

Platforma Optimizely stanowi unikalny przypadek na rynku systemów konfiguracji, ze względu na swoją nie techniczną, lecz marketingową genezę. W przeciwieństwie do LaunchDarkly czy ConfigCat, które wyewoluowały z potrzeb działów inżynierskich / DevOps, Optimizely od samego początku było tworzone z myślą o stworzeniu potężnego silnika do analityki produktowej, badania konwersji oraz przeprowadzani zaawansowanych testów A/B [14]. Obsługa flag funkcji oraz mechanizmów stopniowego wydawania została do tej platformy dobudowana wtórnie, jako techniczny efektork dla procesów badawczych.

Geneza ta ma bezpośrednie przełożenie na sposób działania bibliotek klienckich. Optimizely SDK narzuca aplikacjom mobilnych architekturę silnie zorientowaną na śledzenie zdarzeń [15]. Moduły te agregują dane o zachowaniu użytkownika, na bieżąco weryfikując hipotezy biznesowe przy użyciu wbudowanych silników statystycznych. O ile czyni to platformę absolutnym liderem w warstwie analityki produktowej – o tyle generuje zauważalny narzut wydajnościowy. Wykorzystanie Optimizely wyłącznie do prostych operacji inżynierskich, takich jak przełączanie trybu serwisowego poprzez flagi funkcji, awaryjne wyłączniki i inne (ilustrowane przez Rysunek 3.) – jest zwyczajnie nieoptymalne. Aby adopcja narzędzia takiego jak Optimizely miała sens, konieczne jest posiadanie zaawansowanego strumienia wartości dotyczącego inżynierii wymagań produktów – szczególnie istotnego w przypadku dostarczania systemów w modelu B2C.

Name	Key	Status	Type	Modified	Created	Audience	Traffic Allocation	Days Running	Primary Metric	Variations
Red Button Update	red_button_update	Draft	Targeted Delivery	Oct 2, 2024	Oct 2, 2024	End users	100 %			On
Yellow button experiment	yellow_button_experiment	Draft		Oct 2, 2024	Oct 2, 2024					
Yellow button mab	yellow_button_mab	Running	Multi-Armed Bandit	Oct 2, 2024	Oct 2, 2024		100 %	0	Overall Revenue	Off Yellow and white text Yellow and black text
Mutex-bug-bash-flag	mutex-bug-bash-flag	Running		May 22, 2024	Apr 27, 2022					
Bug Bash Experiment	mutex-bug-bash-	Running	A/B Test	Apr 27, 2022	Apr 27, 2022		100 %	889	bug_bash_even	Off

Rysunek 3. Panel konfiguracyjny flag funkcji w platformie Optimizely

2.3.4 Wnioski z analizy rozwiązań SaaS

Dedykowane platformy usługowe oferują najwyższą na rynku dojrzałość w zakresie zarządzania cyklem życia flag oraz analityki produktowej. Rozwiązują one problemy ograniczonej logiki biznesowej rozwiązań dostawców chmurowych, oferując potężne i kompletne silniki reguł wraz z serwisami towarzyszącymi, takimi jak wyłączniki awaryjne. Wysoki poziom, na którym stoją rozwiązania SaaS jest jednak okupiony wymiernym kosztem, na który składają się:

- Wysokie ceny subskrypcji dla planów produkcyjnych [6], [13], [14],
- Przekazanie kontroli nad krytycznymi regułami – biznesowymi i technicznymi – stronie trzeciej,
- Niska modyfikowalność, rozszerzalność – ogólna elastyczność rozwiązań ponad standardowy zakres usług.

Wszystkie z wymienionych ryzyk mogą być zmitigowane w pełni tylko w przypadku adopcji rozwiązania pochodzącego z unikalnego, lecz niesamowicie popularnego segmentu rynku – oprogramowania o otwartym kodzie źródłowym.

2.4. Platformy o otwartym kodzie źródłowym

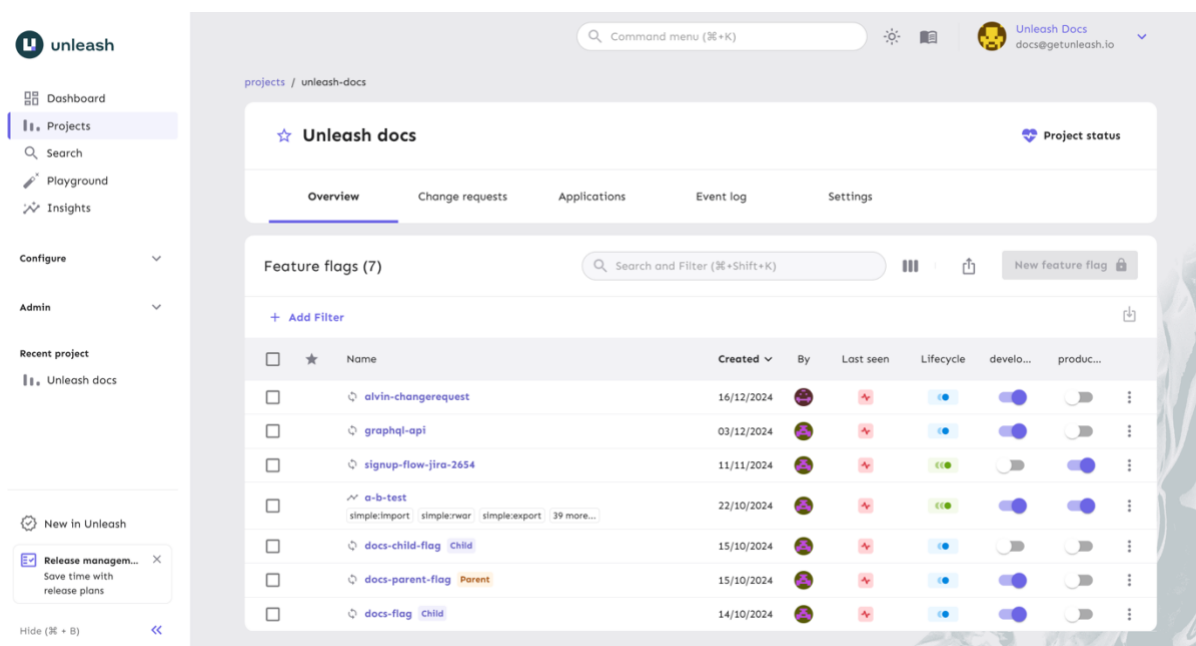
Trzecią, wyraźnie wyodrębnioną kategorię na rynku systemów zdalnej konfiguracji stanowią platformy o otwartym kodzie źródłowym. Ich główna przewaga architektoniczna leży w dziedzinie elastyczności wdrożenia – najczęściej rozwiązania takie oferują nie tylko model subskrypcyjny w myśl SaaS, ale przede wszystkim możliwość uruchomienia całego środowiska na własnej infrastrukturze serwerowej – w trybie *on-premise* lub *self-hosted*. Takie podejście nabiera szczególnego znaczenia w przypadku projektów objętych restrykcyjnymi regulacjami prawnymi, np. w zakresie prywatności. Model ten gwarantuje pełną suwerenność danych – żadne informacje, chociażby powiązane z użytkownikami lub telemetrią aplikacji nie opuszczają wewnętrznej sieci firmy. W tej kategorii rynkowej, najbardziej prominentnymi przykładami platform zdalnej konfiguracji są Unleash, Flagsmith oraz FeatBit.

2.4.1 Unleash

Rozwijany aktywnie od 2014 roku, jest jednym z najstarszych, a zarazem najbardziej dojrzałych systemów otwartych na rynku. Został on zaprojektowany z myślą o spełnieniu rygorystycznych wymogów środowisk korporacyjnych – jego architektura wdrożeniowa opiera się na konteneryzacji, zachowując kompatybilność z wieloma modelami infrastrukturalnymi – wdrożenie może odbyć się na dowolnym komponencie obsługującym obrazy Docker [16]– np. Azure Container lub Kubernetes.

W domenie bezpieczeństwa aplikacji, Unleash nie odstaje od platform takich jak Firebase lub Optimizely – mechanizm autoryzacji systemu opiera się w pełni na statycznych kluczach API [16]. Choć to rozwiązanie stanowi pewien standard branżowy, w przypadku platformy Optimizely posiada ono znaczącą zaletę – umożliwia bardzo ziarniste zarządzanie uprawnieniami w zależności od *tokena* – natomiast pełna integracja w przypadku np. *frontendu* i *backendu* wymaga użycia dwóch różnych *tokenów*. Choć mechanizm ten nie eliminuje ryzyka ekstrakcji klucza – znacząco redukuje on promień rażenia w przypadku kompromitacji. Istotny jest fakt, iż dokumentacja systemu wprost wskazuje, że klucz API dedykowany *frontendowi* należy uznawać za wartość publiczną [16]. Ponadto, platforma wspiera szyfrowanie danych nie tylko w tranzycie (co jest standardem), ale również w spoczynku [16].

Z perspektywy pokrycia funkcjonalności, platforma oferuje bogaty zestaw narzędzi do zarządzania cyklem życia flag, dostępnych z przyjaznego użytkownikowi panelu administracyjnego (Rysunek 4) – od progresywnego wydania opartego na funkcjach haszujących [16] (każdy użytkownik otrzymuje konfigurację na podstawie swojego unikalnego identyfikatora), aż po zbieranie danych telemetrycznych w kontekście testów A/B [16] – lecz niestety, pozostawiając całą część analityczną zewnętrznym integracjom (np. do Google Analytics) [16]. Sposób implementacji flag funkcji na platformie, pozwala na dostarczanie tylko tych flag, które są wymagane w danym momencie dla danego urządzenia, aplikacji i użytkownika – co ułatwia integrację oraz zmniejsza ryzyko wycieków danych.



Rysunek 4. Panel administracyjny Unleash

2.4.2 Flagsmith

Uruchomiona w 2018 roku platforma Flagsmith stanowi bezpośrednią – choć otwartą – konkurencję dla rozwiązań takich jak LaunchDarkly. System oferuje zaawansowane mechaniki

zarządzania flagami funkcji (np. wielościeżkowe drzewa logiczne i sterowanie grupami odbiorców), przewyższając zdolnościami wszystkie rozwiązania od dostawców infrastruktury chmurowej [17].

Rozwiązanie, podobnie jak Unleash, jest dostępne jako otwarty kod w zawężonym zakresie funkcjonalności – jako tzw. *Open Core* [17], [18]. Najbardziej zaawansowane i pożądane mechanizmy w przypadku dojrzałych użytkowników systemów zdalnej konfiguracji, takie jak natychmiastowa propagacja flag (bądź wyłącznik awaryjny) oraz automatyczne harmonogramowanie włączeń pozostają funkcjonalnościami dostępnymi tylko w wersji płatnych planów subskrypcyjnych [17].

2.4.3 FeatBit

Zestawienie zamyka najświeższy projekt – udostępniony w 2021 roku FeatBit. Pomimo krótkiego stażu na rynku, platforma ta stanowi jedno z najbardziej kompletnych otwartych w rozwiązaniu w domenie zdalnej konfiguracji. Oprócz pokrywania funkcji stricte biznesowych, takich jak testy A/B wraz z metrykami impresji [18], zakłada również interesujące rozwiązania w dziedzinie integracji mobilnych.

Jedną z innowacji architektonicznych w FeatBit jest natywne wsparcie dla mechanizmów buforowania i trybu offline po stronie klienta [18]. Jest to kluczowe w środowiskach charakteryzujących się wysoką niestabilnością połączenia internetowego – takich jak urządzenia mobilne. SDK dostarczane przez FeatBit implementuje możliwość zapamiętywania ostatniej znanej, spójnej konfiguracji, co redukuje blokowanie głównego wątku w przypadku nierozumnej integracji – poprawiając niestabilność i wydajność całego rozwiązania, szczególnie w trakcie rozruchu aplikacji. Co więcej, funkcjonalności oferowane jako dodatkowo płatne w takich platformach jak Flagsmith – np. wyłącznik awaryjny działający przez *websocket* – dostępne są za darmo w ramach całego systemu o otwartym kodzie źródłowym [18].

2.4.4 Wnioski z analizy rozwiązań o otwartym kodzie źródłowym

Decyzja o wyborze oprogramowania o otwartym kodzie źródłowym i wdrożenie go w ramach własnych zasobów infrastrukturalnych zdejmując znaczące ryzyko uzależnienia od jednego dostawcy, co ma szczególnie duże znaczenie w przypadku systemów zdalnej konfiguracji – które wymagają ekstensywnych zmian w kodzie źródłowym aplikacji. Zmiana dostawcy staje się wtedy szczególnie problematyczna – należy przy niej rozpatrzyć następujące aspekty:

- **Biznesowy** – tj. migracja danych historycznych dotyczących analityki produktowej: przeprowadzonych badań i ich wyników. Konieczność przerwania aktualnie prowadzonych eksperymentów,
- **Operacyjny** – wdrożenie całego zespołu produktowego (Właściciele Produktu, Analitycy Biznesowi, Zespół Wsparcia Technicznego, Programiści) do funkcjonalności nowego systemu, dostarczenie im odpowiednich poświadczeń (często trudnych do migracji jeden-do-jeden ze względu na różne logiki struktury IAM w ramach rozwiązania),
- **Techniczny** – nowa implementacja systemu, najczęściej wymagająca modyfikacji komponentów na przekroju całego systemu: *backend*, *frontend*, aplikacje mobilne. Natychmiastowe przełączenie z jednego systemu na drugi staje się wysoce ryzykownym przedsięwzięciem – należy je odpowiednio podzielić na etapy i przez czas migracji wspierać oba rozwiązania na raz. Istotne jest, aby zachować w pamięci, iż wszelkie odstępstwo od normy działania w przypadku konfiguracji zdalnej – zgodnie z przeznaczeniem systemu – z pewnością zostanie zauważone przez użytkowników. Mniej istotnymi, lecz wciąż istniejącymi narzutami są aspekty integracyjne w perspektywie całego przedsiębiorstwa – dostarczenie mechanizmu zaopatrzenia użytkowników (*user provisioning*), integracji SSO, powiązanie z narzędziami komunikacyjnymi

wykorzystywanymi przez członków zespołu – np. automatyczne powiadomienia na kanałach Teams lub Slack.

Wobec tych aspektów, opieranie się na płatnym rozwiązaniu w modelu SaaS niesie ze sobą zauważalne ryzyko projektowe, sugerując przewagę modelu *self-hosted*. Należy jednak zwrócić uwagę na to, iż model ten będzie opłacalny tylko dla niektórych firm – w szczególności tych, posiadających rozbudowane zasoby DevOps – zespoły infrastrukturalne i programistyczne, gotowe nie tylko zapewniać SLA dla wewnętrznego rozwiązania – ale również dostarczać poprawki techniczne, na przykład adresujące podatności systemu o otwartym źródle.

2.5. Porównanie możliwości platform i identyfikacja luk bezpieczeństwa

Celem systematyzacji wyników przeprowadzonej analizy, Tabela 1 prezentuje zbiorcze zestawienie kluczowych funkcjonalności wiodących rozwiązań z dziedziny platform zdalnej konfiguracji. Kryteria oceny dobrano pod kątem wymagań współczesnych systemów mobilnych, ze szczególnością uwzględniając elastyczność logiki ewaluacji reguł, dostępność analityki biznesowej, architekturę wdrożenia oraz zaimplementowane mechanizmy autoryzacyjne.

Tabela 1. Macierz funkcjonalności usług zdalnej konfiguracji

Platforma / Cecha	Złożone warunki logiczne	Testy A/B	Ewaluacja po stronie serwera	Architektura self-hosted	Tryb Offline	Autoryzacja
Firebase	Nie	Tak	Tak	Nie	Tak	Klucz API /ID
AWS AppConfig	Nie	Nie	Nie	Nie	Nie	IAM / SigV4
Azure App Config	Nie	Nie	Nie	Nie	Nie	EntraID
Launch Darkly	Tak	Tak	Tak	Nie	Tak	Klucz API / SDK ID
ConfigCat	Ograniczone	Ograniczone	Nie	Nie	Tak	SDK ID
Optimizely	Tak	Tak	Tak	Nie	Nie	JWT
Unleash	Tak	Ograniczone	Tak	Tak	Tak	Klucz API
Flagsmith	Tak	Ograniczone	Tak	Tak	Tak	Klucz API
Featbit	Tak	Tak	Tak	Tak	Tak	Klucz API

Jak wynika z zestawienia, rynek systemów zdalnej konfiguracji wykazuje się wyraźną fragmentacją. Platformy dostawców chmurowych cechują się wysoką generycznością, a co za tym idzie – istotnymi brakami w kontekście zastosowań w systemach mobilnych. Z wyjątkiem Firebase, ignorują one też warstwę badawczą, a zarazem nie dostarczają dedykowanych SDK, zrzucając obowiązek implementacji buforowania na samych użytkowników (tj. programistów). Z kolei rozwiązania dedykowane (SaaS i Open Source) niemal w całości rozwiązują te problemy, oferując rozbudowane pakiety SDK kompatybilne z natywnymi technologiami natywnymi. Zauważalny jest również podział w podejściu do analityki – platformy takie jak Unleash, Flagsmith czy ConfigCat oferują jedynie ograniczone wsparcie dla testów A/B – umożliwiając ich konfigurację i zbieranie telemetry – lecz z pełnym odprowadzeniem danych do zewnętrznych narzędzi analitycznych, w których dopiero można przeprowadzić wnioskowanie.

Najistotniejszym wnioskiem płynącym z przeprowadzonej analizy jest szablon zabezpieczenia systemów zdalnej konfiguracji: w znaczącej większości przypadków, niezależnie od stopnia zaawansowania czy dojrzałości systemu, formy ochrony ograniczają się wyłącznie do warstwy aplikacyjnej, opierając się na statycznych sekretach umożliwiających komunikację z klientem z serwerem.

Jednocześnie, systemy oferujące wysoki poziom zabezpieczeń, takie jak Azure App Configuration wymuszają pewien model operacyjny aplikacji w najmniejszym zakresie wymuszając konkretną architekturę – w największym, wiążąc twórców w domenę IAM, wymagając poświadczeń płynących z Entra ID. W kontekście aplikacji mobilnych, problemy te wynikają z ich natury, gdzie traktuje się je jako „niezaufanych klientów”. Tym samym, mając na uwadze możliwość dekompilacji kodu – nawet tego, który uprzednio został poddany obfuskacji – nie sposób jest traktować systemy konfiguracji zdalnej jako komponenty w pełni bezpieczne i zdolne do obsługi jakichkolwiek wartości sekretnych, gdy są one używane przez aplikacje mobilne.

Wnioski te skłaniają ku pytaniu – jak można sprawić, żeby aplikacje mobilne mogły być postrzegane za zaufanych klientów – tym samym odblokowując pełen potencjał systemów zdalnej konfiguracji? Aspekt ten stanowi bezpośrednią motywację do dalszego poszukiwania nowych paradygmatów zarządzania sekretami i uwierzytelnienia aplikacji mobilnych, co zostanie omówione w kolejnym rozdziale.

2.6. Sposoby zarządzania sekretami w aplikacjach mobilnych

W domenie inżynierii oprogramowania, termin „sekret” odnosi się do szeroko pojętych danych uwierzytelniających: kluczy kryptograficznych, tokenów dostępowych, certyfikatów bądź zwykłych haseł niezbędnych do zapewnienia bezpiecznej, autoryzowanej komunikacji między komponentami [19]. W przypadku klasycznych systemów backendowych, zarządzanie tego typu poświadczeniami nie jest skomplikowane, gdyż może się ono odbywać w pełni zamkniętym, kontrolowanym środowisku. Wartości sekretne nigdy nie muszą być umieszczane w kodzie źródłowym – wystarczy, że będą zawierały stosowne odnośniki, a rozwiązania CI/CD zadbają o wstrzyknięcie ich zawartości w odpowiednim momencie, korzystając np. z Azure KeyVault jako źródła [19]. Aplikacje mobilne jako niezaufani klienci, stanowią zupełnie odmienne wyzwanie [20] – jednocześnie dostarczając pełen wachlarz rozwiązań o różnym poziomie bezpieczeństwa.

2.6.1 Metody Statyczne

Do najbardziej podstawowych, prostych w implementacji, a zarazem najczęściej stosowanych metod obrony przed nieautoryzowanym wglądem w zawartość aplikacji i inżynierią wsteczną są mechanizmy obfuskacji – tj. zaciemniania kodu wynikowego. W ramach poszczególnych platform, do dyspozycji są narzędzia takie jak:

- Android:
 - ProGuard [21],
 - R8 [22],
 - DexGuard [23],
 - DashO [24].
- iOS:
 - Deployment Postprocessing (część IDE - Xcode) [25],
 - Swift Confidential [26],
 - OLLVM (w przypadku kodu C/C++) [27],
 - iXGuard [28].

Cześć z przytoczonych rozwiązań dostępna jest bez opłat jako narzędzia wbudowane w kompilatory lub IDE, bądź jako systemy o otwartym kodzie źródłowym. Wyraźnie zaznaczają się również opcje dostępne jako płatne rozwiązania dedykowane przedsiębiorstwom (takie jak DexGuard lub iXGuard) – skupiają się one na rozszerzeniu funkcjonalności klasycznych narzędzi obfuskacji, oferując możliwie kompleksowe zabezpieczenie pliku binarnego aplikacji mobilnej [23], [28]. Istnieją również dodatkowe serwisy, takie jak Promon Shield [29], działające na już gotowym, pełnym pakiecie aplikacji, stanowiąc swoiste opakowanie implementujące część zabezpieczeń.

Klasyczne obfuskatory działają poprzez modyfikację kodu – zmieniając nazwy klas, metod i zmiennych na ciągi znaków, za którymi nie stoi żadna semantyka. Praktyka ta znacznie utrudnia zrozumienie znaczenia samego kodu, lecz w przypadku większości darmowych rozwiązań posiada krytyczną lukę – zupełnie pomija wartości stałe typu *string* [23]. Oznacza to, że pomimo użycia takiego narzędzia, klucz API, odczytywany z pliku środowiskowego w trakcie kompilacji i podstawiany w odpowiednie miejsca kodu – pozostawie całkowicie widoczny dla osoby próbującej dokonać inżynierii wstecznej aplikacji, używającej narzędzi takich jak np. jadx [20], [30], [31], [32].

Narzędzia nieco bardziej zaawansowane, takie jak DashO, zakładają rozszerzone, wielowarstwowe środki ochrony w ramach jednego narzędzia. Składają się na nie:

- **Klasyczna obfuskacja** – podmiana nazw funkcji i zmiennych,
- **Obfuskacja przepływu sterowania** – np. poprzez dodawanie fałszywych instrukcji warunkowych,
- **Szyfrowanie wartości typu *string*** – zabezpieczając np. klucze API,
- **Szyfrowanie zasobów aplikacji** – utrudniając ponowne wykorzystanie np. ikon,
- **Dodawanie znaku wodnego** – ułatwiając oznaczenie nieautoryzowanych kopii aplikacji,
- **Usuwanie nieużywanych funkcji, typów lub zmiennych**,
- **Wykrywanie prób modyfikacji kodu źródłowego aplikacji** – uniemożliwiając jej działanie,
- **Wykrywanie użycia trybu *debug* na urządzeniu**,
- **Wykrywanie prób ingerencji w system Android**,
- **Wykrywanie uruchamiania aplikacji w ramach emulatora** – uniemożliwiając dalsze jej działanie,
- **Wykrywanie narzędzi nasłuchujących procesy aplikacji**.

Należy jednak zauważyć, iż narzędzia takie jak DashO, działające nie tylko w trakcie kompilacji, lecz też podczas działania aplikacji same podlegają prawom środowiska o zerowym zaufaniu – mogą stać się ofiarą modyfikacji na poziomie pliku binarnego, gdzie zdeterminowany haker może podjąć próbę neutralizacji mechanizmów obronnych – otwierając drogę do dalszych ingerencji. Ponadto, nawet w przypadku bezpiecznego szyfrowania wartości sekretnych, frameworki takie jak Frida umożliwiają przechwycenie odszyfrowanego klucza bezpośrednio z pamięci RAM – w ułamku sekundy przed jego użyciem do autoryzacji zapytania do serwera [33].

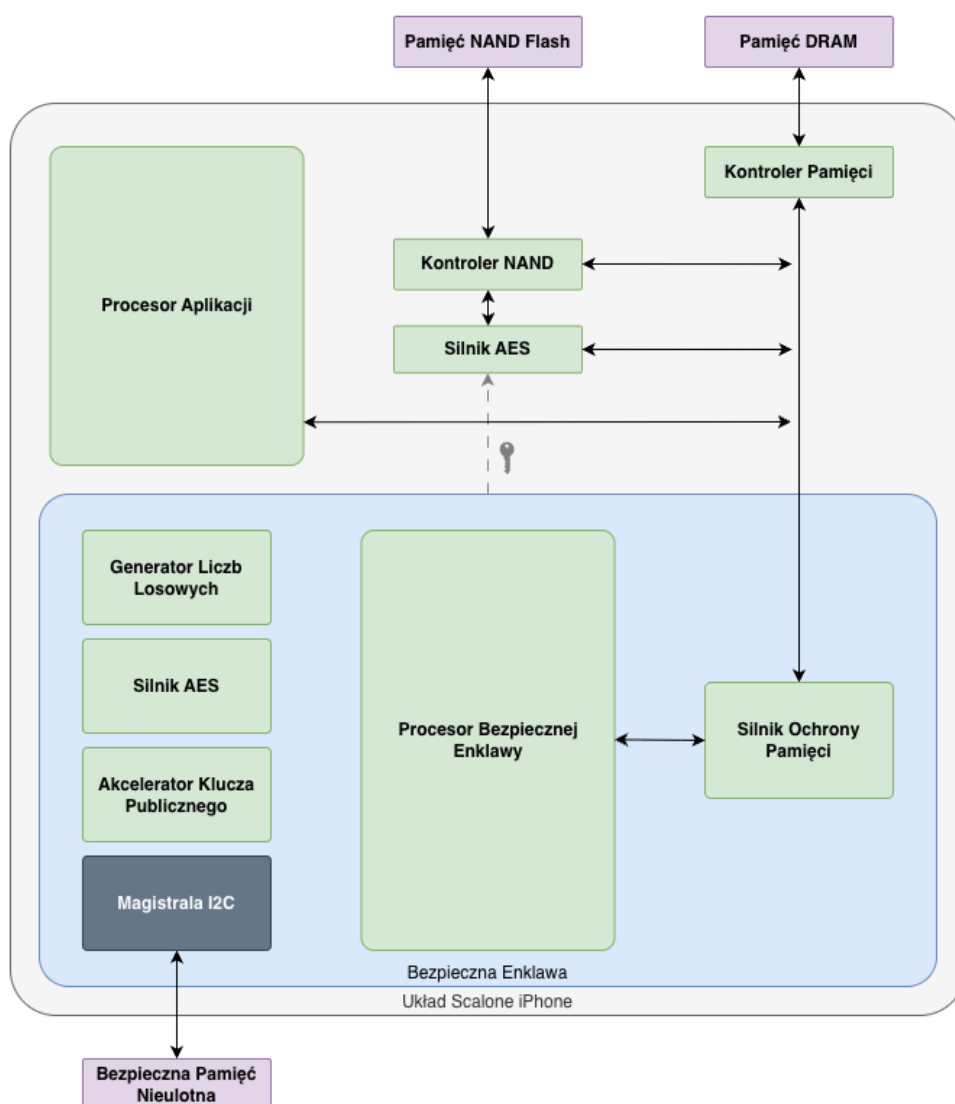
Podobnym mechanizmem, polegającym na dodaniu kolejnej warstwy abstrakcji do penetracji jest podejście zakładające przeniesienie kluczy API do natywnych bibliotek, kompilowanych bezpośrednio do kodu maszynowego. Logika tego typu zabezpieczenia polega tylko na założeniu, że dekompilacja architektury ARM do zrozumiałego kodu C/C++ jest trudniejsza niż próba odczytania plików binarnych bazujących powstałych na podstawie języków takich jak Java czy Kotlin. W rzeczywistości, metoda ta również tylko opóźnia i utrudnia atak – lecz mu nie zapobiega [34], [35]. Sekrety wciąż są możliwe do odczytania poprzez analizy bloków danych tylko do odczytu (rodata) lub nasłuchiwanie komunikacji między warstwą natywną a aplikacją poprzez Java Native Interface.

Tym samym, nie można uznać wymienionych wyżej technik za metody zapewniające odpowiedni poziom bezpieczeństwa. Realizując paradygmat bezpieczeństwa przez zaciemnianie, pozwalają one jedynie na zwiększenie czasu i kosztu ataku – nie rozwiązując w pełni problemu braku zaufania do środowiska klienckiego aplikacji mobilnych.

2.6.2 Bezpieczne magazyny sprzętowe – Secure Enclave w iOS

Mając na uwadze wątpliwą skuteczność metod statycznych, wskazanym kierunkiem rozwoju zabezpieczeń jest wykorzystanie unikalnej architektury urządzeń mobilnych – obecności sprzętowych komponentów kryptograficznych, odseparowanych od reszty systemu na poziomie sprzętowym. Rozwiązania te, wykorzystując proste interfejsy do przekazywania danych pozostawiają najbardziej wrażliwe kwestie bezpieczeństwa systemowi operacyjnemu urządzenia.

W przypadku ekosystemu Apple, funkcje te są realizowane przez komponent *Secure Enclave* [36], [37], który został wprowadzony w 2013 wraz z procesorem A7 w urządzeniach iPhone 5s oraz iPad Mini 3 [38]. Zmiana ta miała bardzo jasną motywację funkcjonalną: szerokie wprowadzenie autoryzacji biometrycznej (*TouchID*) na urządzeniach mobilnych, wyposażając całą rzeszę konsumentów w sprzęt umożliwiający potwierdzenie ich tożsamości w wyjątkowo dokładny sposób [38]. Zadaniem *Secure Enclave* było przechowywanie danych biometrycznych w sposób bezpieczny – zupełnie niedostępny dla kogokolwiek innego w przypadku kradzieży bądź zgubienia urządzenia – zakładając najbardziej radykalne scenariusze prób ingerencji na poziomie sprzętowym w celu ekstrakcji danych. Enklawa ta została zaimplementowana poprzez dodanie zupełnie izolowanego podsystemu wykorzystującego oddzielny, dedykowany system operacyjny – mikrojądro *sepOS* [36]. Wyposażona została również w niezależny generator liczb losowych, a jej pamięć operacyjna – choć fizycznie dzielona z tą główną – została całkowicie zaszyfrowana sprzętowo i połączona dedykowaną szyną IIC [36]. W kontekście szyfrowania – bezpieczna enklawa zawiera również separowany, dedykowany moduł szyfrujący AES256 [36]. Operacje szyfrujące natomiast są wykonywane z użyciem unikalnego identyfikatora urządzenia, który nadawany jest na etapie produkcji – i następnie w żadnym wypadku nie może zostać odczytany przez oprogramowanie – jedyny sposób na wykorzystanie go to wywoływanie ściśle określonych operacji w interfejsie enklawy [36], [37]. Separacja na tym poziomie, uniemożliwia odczytanie jakiegokolwiek prywatnego materiału kryptograficznego na urządzeniu, które zostało skompromitowane. Innowacja ta umożliwiła otwarcie zupełnie nowej ery w bankowości i płatnościach mobilnych – czyniąc ze smartfonu narzędzie znacznie bezpieczniejsze od klasycznej karty płatniczej.



Rysunek 5. Architektura Bezpiecznej Enklawy na przykładzie smartfonu Apple iPhone

Architektura bezpiecznej enklawy przedstawionej przez Rysunek 5, jest również odporna na różne wektory ataków – dzięki odseparowanym blokom pamięci używanym przez jej liczniki, niemożliwe jest wykonanie ataku typu replay – tj. scenariusza, w którym atakujący przejmuje *token* odpowiedzialny za odblokowanie zawartości enklawy i próbuje powtórzyć tę samą akcję. Układ scalony wykryje próbę użycia starego *tokena* [36] – a sama manipulacja jego pamięci jest niemożliwa.

Komponent Apple jest zabezpieczony nie tylko z perspektywy oprogramowania – rozważono również scenariusz fizycznego wymontowania enklawy z urządzenia i prób manipulacji zasilaniem w celu kompromitacji [36]. Również w tym przypadku, próby te się nie powiedą; W przypadku jakichkolwiek odstępstw od standardowych parametrów funkcjonowania układu – obwody monitorujące zatrzymają zegary taktujące w całej enklawie, oczekując na kompletny reset układu [36].

Dokładnie te mechanizmy wykorzystywane są przede wszystkim do ochrony sekretów użytkownika w ramach usługi Keychain [39] – zazwyczaj przechowywane są w nich:

- Asymetryczne klucze parowania FIDO2,

- Poświadczenia aplikacji bankowych,
- Tokeny sesyjne (np. *JWT Refresh Token*).

Gdy z danych należy skorzystać – użytkownik musi wykonać potwierdzenie dostępu do enklawy za pomocą autoryzacji biometrycznej. Z racji generyczności interfejsów, w bezpiecznej enklawie możliwe jest przechowanie niemalże dowolnego rodzaju informacji [39]. Jak wygląda więc sytuacja w przypadku sekretów aplikacji, takich jak klucze API?

Naturalnie, dane tego typu również mogą zostać programistycznie umieszczone w ramach bezpiecznej enklawy – lecz wtedy pojawia się problem – skąd należy je wziąć na początku? W ujęciu logicznym, jest to klasyczny *bootstrapping problem*: aby aplikacji zawierała sekrety, które następnie można umieścić w bezpiecznej pamięci – muszą one znaleźć się w kompilowanym pliku aplikacji – którego nie sposób w pełni zabezpieczyć przed nieautoryzowanym dostępem [40]. Niektóre sekrety (często niewymagane do działania aplikacji od samego początku) mogą zostać pobrane po zautoryzowaniu użytkownika – lecz wtedy, czemu nie użyć samej sesji użytkownika do uwierzytelnienia aplikacji wobec systemu zdalnej konfiguracji, zamiast kluczy API? Faktem jest, że:

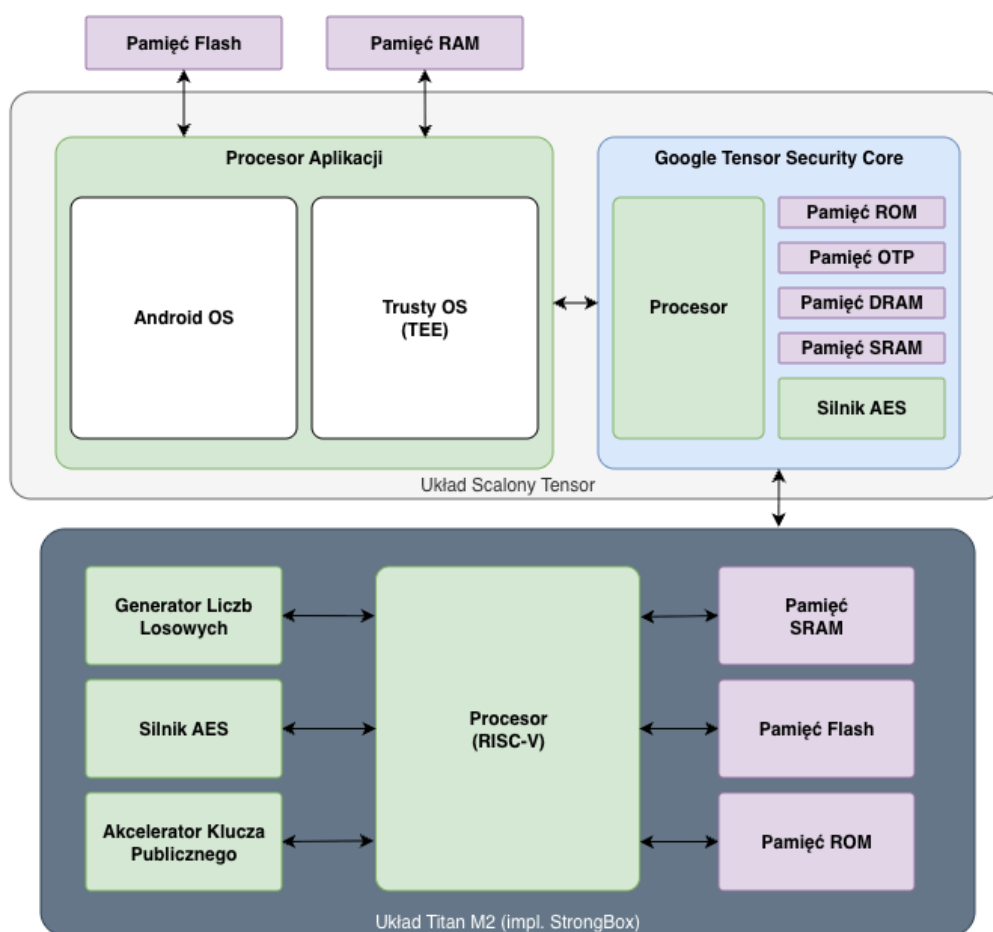
- Nie wszystkie aplikacje wymagają uprzedniego zalogowania użytkownika do działania,
- Wiele aplikacji wymaga informacji z systemów zdalnej konfiguracji jeszcze zanim użytkownik zobaczy okno logowania.

Wobec tego, aby uniknąć problemu *bootstrappingu*, idealnym rozwiązaniem byłoby użycie samej w sobie bezpiecznej enklawy – do uwierzytelnienia nie użytkownika, lecz całego urządzenia i aplikacji. Istnieje mechanizm, który pozwala na realizację tego scenariusza, i zostanie on omówiony w dalszej części pracy.

2.6.3 Bezpieczne magazyny sprzętowe – TEE i StrongBox w Android

Podążając za kierunkiem wyznaczonym przez urządzenia Apple, ekosystem Android również doczekał się swojego ekwiwalentu bezpiecznej enklawy. Pierwszą implementacją komponentów o podwyższonym poziomie bezpieczeństwa był segment *Trusted Execution Environment* [41]. Podobnie jak rozwiązanie iOS – lwia część rozwiązania została zaprojektowana na podstawie technologii ARM TrustZone [42] – mającej na celu programistyczne i sprzętowe oddzielenie domen odpowiedzialnych za standardowe operacje urządzenia (*Rich Execution Environment*) od elementów przechowujących najbardziej wrażliwe dane i kryptografię (*Trusted Execution Environment*). Ponownie, TEE działa na oddzielnym systemie operacyjnym (np. Trusty OS), odpowiedzialnym stricte za operacje kryptograficzne [41], [43]. Pomimo podobnych idei przyświecających architekturze, samo TEE w przypadku telefonów androidowych nie jest wyposażone w równie szeroki wachlarz dedykowanych komponentów sprzętowych.

Lukę tę miał pokryć standard StrongBox, wprowadzony przez firmę Google wraz z systemem Android 9: wprowadzał on oddzielne układy scalone, posiadające zupełnie odseparowany procesor i pamięć, poprawiając bezpieczeństwo w przypadku próby fizycznej kompromitacji układu [43], [44]. Całość architektury, integrującej zarówno TEE i StrongBox przedstawiana jest przez Rysunek 6.



Rysunek 6. Architektura TEE i StrongBox na przykładzie smartfonu Google Pixel

Odpowiednikiem iOSowego Keychain jest Android Keystore API [43]. Pozwala on na generowanie kluczy (asymetrycznych i symetrycznych) powiązanych bezpośrednio z modułem sprzętowym – zachowując jego klucz prywatny tylko w obrębie bezpiecznego środowiska. Do jedynych możliwości programisty w tej sytuacji należy opcja podpisania lub odszyfrowania danych poprzez ściśle określony interfejs – co również służy głównie do przechowywania sekretów użytkownika wspomnianych w poprzednim podrozdziale. Z racji analogicznej logiki, w kontekście zarządzania sekretami aplikacji pojawia się tutaj dokładnie ten sam problem *bootstrappingu*.

Zakładając scenariusz pełnego bezpieczeństwa przechowywanych danych – należy mieć na uwadze, że sekrety takie jak klucze API, aby mogły być użyte, będą musiały opuścić bezpieczne środowisko wykonawcze. Jest to miejsce, w którym ponownie potencjalny atakujący może wpiąć się swoimi narzędziami, dokonując na przykład zrzutu pamięci operacyjnej urządzenia w kluczowym momencie. Wobec tego, aby uwierzytelnienie urządzenia było stuprocentowe – sekret nigdy nie może opuścić bezpiecznego środowiska wykonywania – co również jest opisane przez procedurę atestacji, opisaną w dalszej części pracy.

2.6.4 Bezpieczeństwo w tranzycie

Choć zabezpieczenie danych w izolowanych modułach sprzętowych, takich jak *Secure Enclave* bądź *StrongBox*, rozwiązuje problemy ochrony sekretów w spoczynku – należy zwrócić uwagę na to, co dzieje się z nimi później. W architekturze systemów mobilnych należy założyć, że takowe sekrety prędzej czy później zostaną wykorzystane do autoryzacji komunikacji sieciowej z zewnętrznymi

usługami, wprowadzając potrzebę ochrony danych nie tylko w spoczynku, lecz również w transzycie [45].

Współczesnym standardem zabezpieczenia komunikacji klient-serwer jest protokół TLS, który chroni dane przed pasywnym podsłuchem, możliwym do realizacji np. w niezaufanych sieciach [46]. Pomimo tej podstawowej warstwy ochrony, są to środki niewystarczające wobec zmyślnych hakerów: kluczowym aspektem w tej domenie staje się oparcie modelu TLS na systemowym magazynie zaufanych certyfikatów. W przypadku kompromitacji urządzenia, potencjalny atakujący może bez problemu zainstalować na urządzeniu swój własny certyfikat korzenia i wskazać go jako zaufanego [47], [48]. Następnie, z wykorzystaniem narzędzi udostępniających funkcję *proxy* przechwytyjącego – tj. CharlesProxy lub BurpSuite – atakujący może przekierować ruch sieciowy na swój serwer. Tym samym, aplikacja mobilna nawiąże szyfrowane połączenie z serwerem atakującego, a nie z rzeczywistym systemem zdalnej konfiguracji, realizując klasyczny atak *Man-in-the-middle*.

W obliczu takiego zagrożenia, powstała metoda *SSL Pinning* [49], [50], szeroko wykorzystywana w domenie aplikacji mobilnych. Polega ona na weryfikacji certyfikatu nie na poziomie systemu – lecz aplikacji próbującej nawiązać połączenie, za pomocą skrótu SHA stworzonego na podstawie oczekiwanego poświadczenia serwera docelowego.

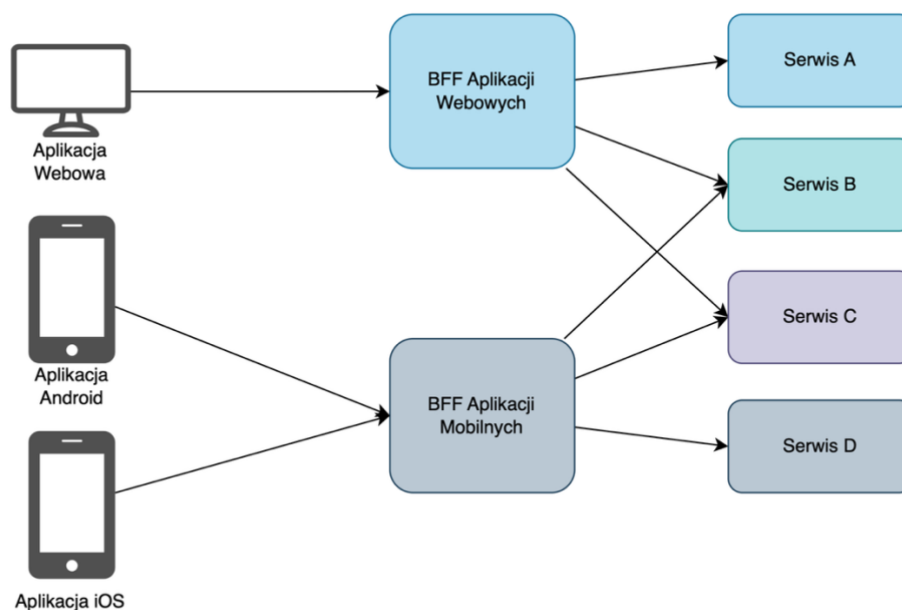
Niestety, weryfikacja certyfikatu po stronie aplikacji, choć jest dodatkowym środkiem bezpieczeństwa – jest również mieczem obusiecznym. W przypadku skompromitowanego sprzętu, logika ta staje się wysoce podatna na analizę, podsłuch i potencjalną ingerencję w sam kod źródłowy aplikacji – prowadząc do wymuszenia zaufania wobec certyfikatów niewiadomego pochodzenia [51].

Przeprowadzona analiza sugeruje następujący wniosek: mechanizmy bezpieczeństwa, takie jak *SSL Pinning* są nieocenione w przypadku ochrony przed zewnętrznymi wektorami ataku. Jednakże, w przypadku samej kompromitacji urządzenia, stają się bezwartościowe z racji istnienia łatwych sposobów ich obejścia.

2.6.5 Architektura Backend-for-Frontend

Konieczność każdej aplikacji mobilnej do operowania w środowisku o zerowym zaufaniu – umożliwiającym kompromitację na wiele sposobów – skłania ku poszukiwaniu środków zabezpieczeń sięgających poza ten ekosystem. Głównym przykładem takiego podejścia jest wzorzec architektoniczny *Backend-for-Frontend*, którego najistotniejszym założeniem jest całkowite usunięcie materiału kryptograficznego i poświadczeń dostępowych z kodu aplikacji klienckiej – i przeniesienie go do kontrolowanego, zaufanego środowiska serwerowego [52].

Tym sposobem, aplikacja mobilna nie komunikuje się już bezpośrednio z systemem zdalnej konfiguracji – robi to serwer, służący za bezpieczne *proxy* [52]. Na urządzeniu nie ma kluczy API do systemu – są one pobierane z bezpiecznego magazynu (takiego jak np. Azure KeyVault) i wstrzykiwane przez BFF w trakcie przekazywania żądania do adresata. Plusem tej koncepcji jest zachowanie obsługi sekretu w ramach bezpiecznego środowiska – nie jest on przesyłany przez niezaufane sieci – a w przypadku architektury modelu *self-hosted* komunikacja ta może się odbywać wręcz w ramach tej samej sieci (lub z dodatkowymi zabezpieczeniami, takimi jak *whitelisting* adresów IP lub *Private Endpoint*). W przypadku potrzeby ochrony poświadczeń do usług dostawców zewnętrznych – architektura ta, zilustrowana przez Rysunek 7, w pełni spełnia to wymaganie.



Rysunek 7. Architektura Backend-for-Frontend w przypadku wielokanałowej dystrybucji aplikacji

Niestety, jeśli spojrzeć na rozwiązanie w ujęciu holistycznym – okaże się, że architektura BFF oferuje jedynie złudzenie bezpieczeństwa. Rzeczywiście, chroni ona poświadczenia do bardziej wrażliwych usług – lecz jak poświadczyć dostęp do samego serwera BFF, który przekazuje żądania dalej? Jest to sytuacja, w której występuje tzw. Problem Pierwszego Sekretu – sytuacja, kiedy dostęp do usług umożliwiających komunikację z innymi serwisami nie jest chroniony w wystarczającym stopniu [53]. W przypadku próby ochrony tego połączenia (np. kolejnym kluczem API), powtarzają się problemy opisane w podrozdziale 2.3.1. Naturalnie, problem ten nie występuje w przypadku autoryzowania zapytań z pomocą sesji użytkownika (np. poprzez *tokeny* JWT). Ponownie, tak jak wspomniano w poprzednich punktach – opcja ta nie zawsze jest dostępna dla każdego twórcy aplikacji mobilnych oraz natrafia na problem inicjalizacji usług przed zalogowaniem się użytkownika.

Podsumowując, wzorzec *Backend-for-Frontend* jest doskonałym narzędziem do orkiestracji zapytań i przykrycia ogólnej kompleksowości *backendu* aplikacji – lecz aby był w pełni bezpieczny, wymaga dodatkowej ochrony wobec spreparowanych żądań wykonywanych przez potencjalnie skompromitowane urządzenie. Stanowi to jednocześnie dowód na to, że problemu nie da się w pełni rozwiązać poprzez kolejne, bardziej kompleksowe formy ukrywania sekretów: rozwiązanie wymaga całkowitej zmiany paradygmatu uwierzytelniania i znalezienie zaufanej trzeciej strony do facylitacji połączenia między komponentami systemu [53] – innymi słowy, implementacji weryfikacji integralności mobilnego środowiska wykonawczego.

2.6.6 Atestacja na platformie Android – Play Integrity API

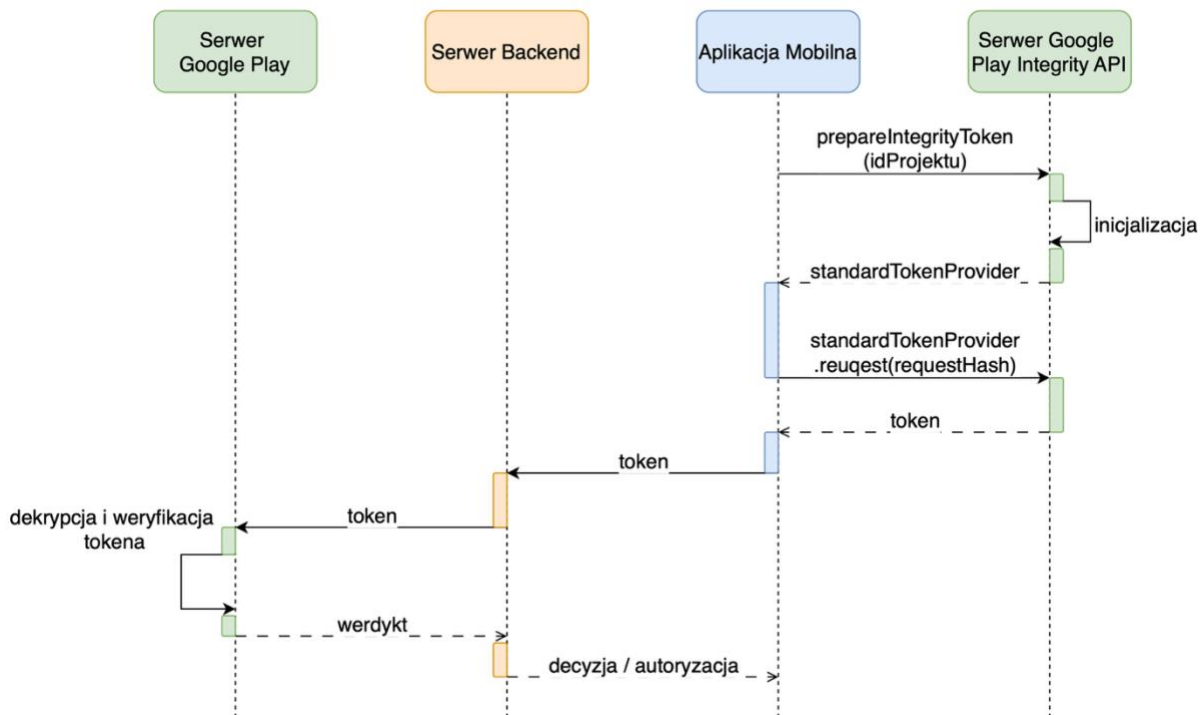
Wobec niepełnej skuteczności tradycyjnych metod zabezpieczeń – od statycznego zaciemniania, poprzez izolację magazynowania, aż po architekturę BFF – wszystkie rozwiązania skazane są na asymetrię relacji pomiędzy programistą a atakującym w kontekście systemów mobilnych. To atakujący ma przewagę, będąc w stanie manipulować skompromitowanym urządzeniem – z biegiem czasu prowadząc do wyekstrahowania wszelkich lokalnie przechowywanych sekretów. Odpowiedzią na tę sytuację jest wdrożenie mechanizmów atestacji środowiska mobilnego.

Idea atestacji opiera się na kryptograficznym dowodzie zdrowia i bezpieczeństwa systemu, który bazuje na fundamencie zaufania sprzętowego [53], [54], [55], [56]. W tym przypadku, system docelowy nie bada już tylko poprawności statycznego poświadczenia w postaci klucza API – zyskuje on zdolność zbadania integralności całego ekosystemu, w którym funkcjonuje aplikacja mobilna. W przypadku Androida, wymaga to pełnej delegacji procesu weryfikacji do zewnętrznego arbitra – usługi Play Integrity API. Jest to potężne narzędzie, zintegrowane bezpośrednio z pozostałymi usługami Google Play, które do oceny stanu urządzenia wykorzystuje między innymi sprzętowe moduły TEE i StrongBox [56].

W holistycznym ujęciu, mechanizm ten zakłada współpracę trzech aktorów: aplikacji mobilnej, serwera *backend* oraz niezależnego serwera atestacyjnego od Google. Aby zapobiec atakom powtórzeniowym, aplikacja dokonuje kryptograficznego powiązania dowodu tożsamości z żądaniem (*request hash*) – w przypadku klasycznego API, rolę tę odgrywa wartość generowana przez serwer *backend* (*nonce*) [57]. Aplikacja żąda wykonania weryfikacji środowiska przez Play Integrity API, którego produktem jest *token* trafiający do serwera docelowego. W trakcie tego procesu, weryfikowane są trzy kluczowe kategorie bezpieczeństwa [56]:

1. **Integralność Urządzenia:** ocena czy środowisko uruchomieniowe aplikacji jest bezpieczne. Zakłada się trzy poziomy uwiarygodnienia: „Podstawowa Integralność”, „Integralność Urządzenia” oraz najwyższy – „Silna Integralność”. Standardy te odpowiadają kolejno następującym procesom - testom po stronie API, weryfikacjom parametrów i poświadczeń urządzenia oraz weryfikacjom integralności urządzenia na poziomie sprzętowego układu StrongBox podczas procesu *Secure Boot*,
2. **Integralność Aplikacji:** gwarancja, że zapytanie pochodzi od aplikacji opublikowanej przez zaufanego dewelopera w sklepie Google Play. System oblicza sumę kontrolną pliku APK/AAB, pozwalając na wykrycie nieautoryzowanych modyfikacji kodu lub prób przepakowania aplikacji,
3. **Licencjonowanie użytkownika:** zwraca informację, czy aplikacja została legalnie zainstalowana z oficjalnego źródła.

Po dokonaniu następujących weryfikacji, serwer *backend* odszyfrowuje token za pomocą publicznych kluczy Google, oblicza skrót kryptograficzny otrzymanego żądania i porównuje go z zawartym w atestacji parametrem *request hash*. Dzięki oparciu uprzedniego mechanizmu weryfikacji na kluczach sprzętowych obecnych w ramach TEE sprawia, że próby sfalszowania *tokena* z poziomu skompromitowanego urządzenia kończą się fiaskiem – w tej sytuacji, atakujący nie walczy już z systemem operacyjnym, lecz z fizyczną konstrukcją układu scalonego [57]. Całość wyżej opisanego przepływu danych, przedstawiana jest przez Rysunek 8.



Rysunek 8. Mechanizm działania atestacji środowiska wykonawczego przez Play Integrity API

2.6.7 Atestacja na platformie iOS – DeviceCheck i AppAttest

Platforma iOS implementuje mechanizmy atestacji analogiczne do tych obecnych w ekosystemie Android. W tym przypadku, Apple w pełni wykorzystuje możliwość jednoczesnej kontroli dokładnej architektury oprogramowania, jak i sprzętu, który je uruchamia [36]. Pozwala to na podwyższenie poprzeczki bezpieczeństwa względem rozwiązania Google. Pierwotnie, założenie opierało się na serwisie DeviceCheck – zdalnej usłudze, pozwalających na anonimowe przechowywanie danych na serwerze Apple, aby dokonywać weryfikacji tożsamości urządzenia – np. umożliwiało to sprawdzanie czy wersja próbna aplikacji była już uruchomiona na danym urządzeniu [58]. Ze względu na konieczność przechowywania danych dla każdego telefonu Apple wraz z każdą zainstalowaną na nim aplikacją, możliwości były dość ograniczone – przechowywanie stanu aplikacji możliwe było na maksymalnie dwóch bitach danych [58]. Wraz z systemem iOS 14, mechanizm DeviceCheck został wzbogacony o mechanizm AppAttest – pozwalający na sprzętową weryfikację komunikacji między aplikacją mobilną i serwerem [54].

App Attest opiera się na infrastrukturze klucza publicznego powiązanej bezpośrednio z walorami sprzętowymi układu scalonego Bezpiecznej Enklawy [54], [37], [36], [55]. W przeciwieństwie do rozwiązania Play Integrity API – nie używane są tutaj jednorazowo generowane *tokens* – rozwiązanie Apple opiera się na kryptograficznym powiązaniu danego urządzenia z serwerem backendowym aplikacji. Proces ten dzieli się na dwie fazy:

- Faza Atestacji (klucza),
- Faza Asercji (zadań).

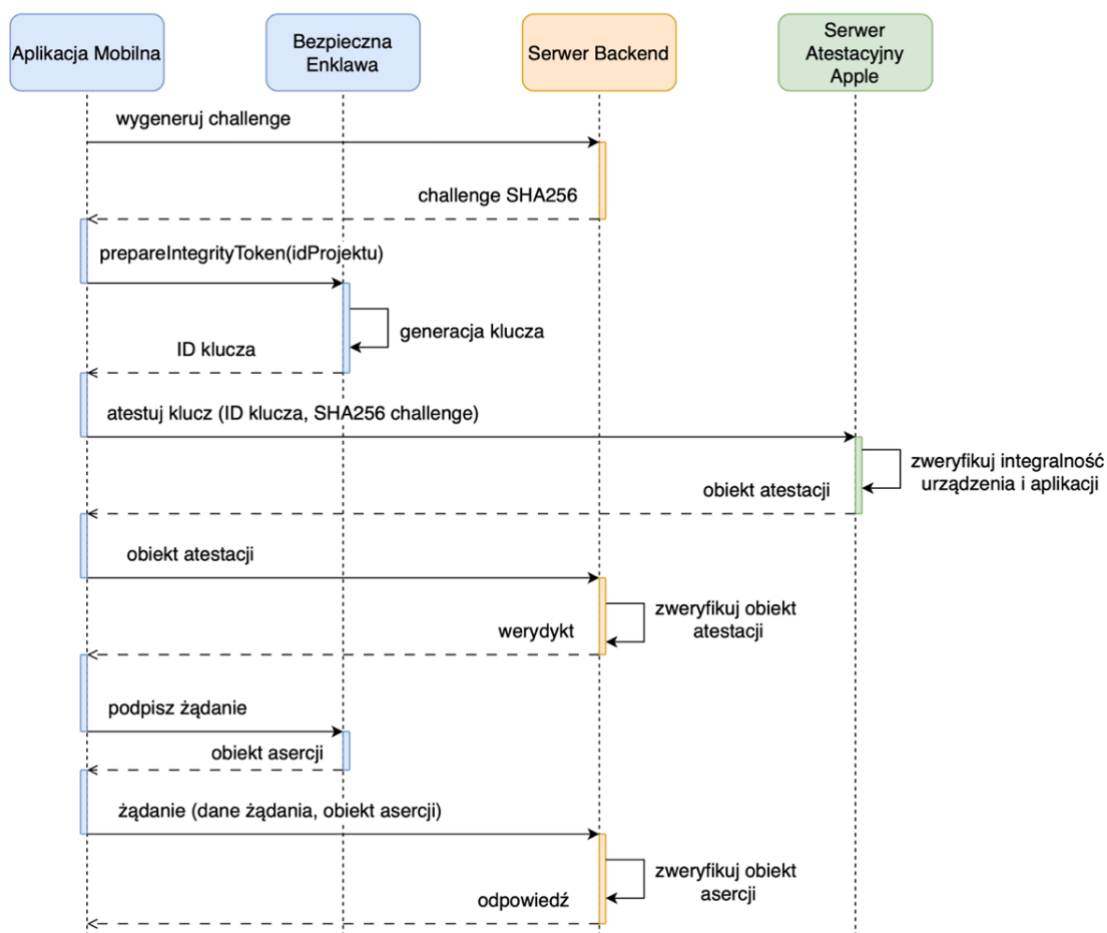
W pierwszej kolejności, aplikacja generuje świeżą parę kluczy asymetrycznych w ramach Bezpiecznej Enklawy: klucze nigdy nie opuszczają tego układu (klucz prywatny jest wręcz wypalony w układzie), mogą być jedynie używane za pomocą ich unikalnego identyfikatora. Aby zapobiec atakom replay, serwer *backendowy* generuje unikalne wyzwanie (*challenge*), które aplikacja przekazuje do metody atestacji. System iOS następnie przesyła dane do serwerów Apple, które po weryfikacji integralności

systemu zwracają obiekt atestacyjny. Obiekt ten zawiera klucz publiczny oraz łańcuch certyfikatów potwierdzający, że został on wygenerowany w autentycznym module *Secure Enclave* przez oryginalną aplikację pochodzącą ze sprawdzonego źródła (tj. podpisaną przez programistę) [54], [55].

Następnie aplikacja mobilna może wykorzystać uprzednio wykonaną weryfikację do uwierzytelnienia swojej komunikacji z serwerem *backend* (lub BFF): wykorzystywany jest do tego obiekt asercji. Jest on tworzony dla każdego żądania HTTP na podstawie danych zapytania (w formie skrótu kryptograficznego). Następnie, skrót ten jest podpisywany przez klucz prywatny wewnątrz Bezpiecznej Enklawy, potwierdzając, że owe żądanie zostało wygenerowane przez autentyczną instancję aplikacji [54], [55]. Dokładny przepływ danych pomiędzy wymienionymi komponentami przedstawiony jest przez Rysunek 9.

Serwer *backendowy*, odbierając żądanie dokonuje weryfikacji obiektu asercji poprzez użycie wcześniej poznanego i zweryfikowanego klucza publicznego powiązanego z danym urządzeniem. Dzięki temu, uzyskiwane jest potwierdzenie trzech aspektów bezpieczeństwa komunikacji:

- **Integralność danych:** Dzięki weryfikacji skrótu kryptograficznego, pewne jest że żądanie nie zostało zmodyfikowane w locie,
- **Autentyczność aplikacji:** Dzięki weryfikacji podpisu i uprzedniej atestacji, pewne jest, że zapytanie pochodzi od niemodyfikowanej aplikacji,
- **Bezpieczeństwo urządzenia:** Dzięki konieczności wykorzystania komponentu bezpiecznej enklawy, pewne jest, że zapytanie zostało wykonane przy pomocy nieskompromitowanego, prawdziwego urządzenia.



Rysunek 9. Mechanizm atestacji środowiska wykonawczego przez Apple App Attest

Podsumowując, wprowadzenie mechanizmu App Attest znacząco zwiększa poziom bezpieczeństwa komunikacji aplikacji mobilnej z serwerami – wszelkie metody prób ingerencji, takich jak instrumentacja pamięci RAM, użycie zmodyfikowanych urządzeń lub emulatorów, nie są w stanie wymusić na procesie asercji podpisania zmanipulowanych danych. Czyni to z mechanizmu atestacji kluczowy element współczesnej architektury Zero Trust na aplikacjach mobilnych.

2.7. Sposoby monitorowania użycia i testowania funkcjonalności w aplikacjach mobilnych

Przez długie lata, proces wytwarzania oprogramowania posiadał mocno ograniczone zdolności kreacji pętli sprzężenia zwrotnego z ich klientami, gdy produkt już znalazł się na fizycznych lub wirtualnych półkach sklepowych. Jakakolwiek informacja zwrotna musiała opierać się na zaangażowaniu samego użytkownika, skorego do podzielenia się swoimi doświadczeniami, np. drogą mailową. Tym samym, aplikacje powstawały na podstawie limitowanego zakresu informacji – programiści nieraz realizowali wiele nowych wydań oprogramowania, nie mając świadomości czy zmiany, które wprowadzają rzeczywiście spełniają założony cel biznesowy [1], [59], [60]. Wraz z premierą pierwszych smartfonów z systemami Android i iOS powstała zupełnie nowa przestrzeń na rynku aplikacji – zawierająca szeroką rzeszę użytkowników B2C – miliony ludzi, których wybory konsumenckie musiały być rozumiane na zupełnie innym poziomie niż wcześniej.

W myśl tej potrzeby, współczesne aplikacje mobilne stopniowo zaczęły odchodzić od paradygmatu statycznych, okazjonalnie aktualizowanych plików binarnych. Współczesna aplikacja mobilna jest zupełnie inna: jest ona dynamicznym systemem, którego zawartość sterowana jest zdalnie w czasie rzeczywistym, co osiągane jest m.in. przez implementację systemów konfiguracji zdalnej omówionych w podrozdziale 2.2. Tym samym, możliwe staje się pełne rozdzielenie wdrożenia nowego kodu od wydania nowej funkcjonalności – pozwala to również na krótkotrwałe udostępnianie użytkownikom nowych doświadczeń bez konieczności przechodzenia przez długotrwały proces weryfikacji i publikacji aktualizacji w sklepach App Store czy Google Play [1], [60]. Mechanizm ten otwiera drogę do zaawansowanego monitorowania zachowań użytkowników wobec zmian wprowadzanych przez zespół inżynierski – oraz ewaluacji hipotez zespołu produktowego poprzez wielowariantowe eksperymenty przeprowadzane na produkcyjnej grupie odbiorców – szeroko zwane Testami A/B.

Z inżynierskiego punktu widzenia, testy A/B realizowane są przez pseudolosowy, a zarazem deterministyczny podział populacji użytkowników na minimum dwie grupy:

- **Grupę kontrolną**, korzystającą ze standardowego, uprzednio dostępnego interfejsu lub logiki,
- **Grupę eksperymentalną**, która otrzymuje dostęp do jednego z wariantów nowego, przeprojektowanego interfejsu lub logiki.

Przydział do grupy następuje najczęściej poprzez obliczenie kryptograficznego skrótu na podstawie parametrów indywidualnej instalacji lub użytkownika, zanim pobrana zostanie zdalna konfiguracja aplikacji [59], [60]. Użycie do tego celu parametrów instalacji (takich jak unikalny identyfikator urządzenia) jest bardziej uniwersalne, gdyż nie wymaga logowania – lecz należy mieć na uwadze, że w tym scenariuszu ta sama osoba korzystająca z dwóch urządzeń, może zobaczyć dwa zupełnie różne interfejsy oraz niemożliwe będzie ewaluowanie logiki odbywającej się przed zautoryzowaniem użytkownika. W przypadku użycia identyfikatora użytkownika, możliwe jest kompletne otoczenie danej osoby testowanymi zmianami, nie tylko w ramach jednego urządzenia, ale również na przestrzeni wielu platform dostarczania aplikacji – to samo doświadczenie może być widoczne i ewaluowane na aplikacjach Android, iOS oraz Web.

Pobranie konfiguracji i wyświetlenie konkretnej wersji docelowemu odbiorcy jest jedynie dopiero początkiem badania – aby móc wyciągnąć z niego wnioski, konieczne jest zebranie pełnej informacji odnośnie do sposobu interakcji użytkownika ze zmienioną aplikacją [59], [60]. Konieczne jest zaimplementowanie w ramach aplikacji wielu znaczników, które będą wysyłane do centralnej bazy danych jako telemetria sesji użytkownika. Warto mieć przy tym na uwadze fakt, iż przydzielenie danego użytkownika do danego wariantu i uruchomienie przez niego aplikacji nie oznacza automatycznie, że zmiana została przez niego zobaczona – kluczowe jest rejestrowanie nie tylko samego przydziału, lecz również impresji, która z niego wynika [59]. Oprócz samego faktu, istotne jest zarejestrowanie wszystkich pobocznych parametrów zdarzeń – zaczynając od najbardziej podstawowych miar, jak dokładny czas, a kończąc na najbardziej ogólnych czynnikach mogących korelować z lub wpływać na decyzje użytkowników – takich jak model telefonu, język, położenie geograficzne, itd.

Z perspektywy klienta mobilnego, proces zapisywania zdarzeń telemetrycznych nie przebiega w pełni w czasie rzeczywistym. Natychmiastowe wysyłanie tak szczegółowych danych wymaga wykonywania operacji sieciowych z dużą częstotliwością, co może przyczyniać się do znaczącego drenażu baterii – oraz utraty części zdarzeń w wyniku tymczasowego zakłócenia połączenia. Wobec tego, optymalnym podejściem jest zbieranie zapisów opisujących akcje użytkownika w lokalnym buforze, a następnie wysyłanie ich na serwer w momencie spełnienia konkretnych warunków. Mogą do nich należeć zdarzenia takie jak przekroczenie zdefiniowanego progu liczby zdarzeń w buforze lub upływanie zadanego interwału czasowego [61].

Zdarzenia te, po trafieniu na serwer trafiają do silników kalkulacyjnych dostarczających najbardziej istotne wskaźniki analityczne. Można wyróżnić trzy podstawowe metryki ewaluacyjne. Pierwszą i

zarazem najprostszą jest bezwzględny wolumen zarejestrowanych zdarzeń [62] – czyli sumaryczną liczbę śledzonych interakcji w ramach k-tego wariantu eksperymentu:

$$C_k = \sum_{i=1}^{n_k} e_i$$

Gdzie n_k oznacza całkowitą liczbę użytkowników przypisanych do k-tego wariantu, natomiast e_i przyjmuje wartość 1 jeśli i-ty użytkownik wyzwolił docelowe zdarzenie – a 0 w przeciwnym wypadku. Przydatność tej miary jest wątpliwa – w najlepszym wypadku pozwala ona na diagnozę ewentualnych problemów z przydziałem grup lub aktywacji wariantu – jednakże, jest to element stanowiący fundament dalszych kalkulacji. Znacznie większą wartość analityczną niesie współczynnik konwersji [62] – tj. procentowy stosunek występowania zdarzeń docelowych (np. zakup artykułu) do zdarzenia bazowego (np. wyświetlenie zielonego przycisku „kup” przy artykule). Pozwala on na obiektywne porównanie wariantów, pomimo różnej wielkości próbki:

$$R_k = \frac{\sum E_{target,k}}{\sum E_{base,k}} \cdot 100\%$$

Ostatecznym wyznacznikiem sukcesu bądź porażki wariantu hipotezy testu A/B jest wskaźnik przyrostu [62], określający względną różnicę skuteczności w wywołaniu zdarzenia docelowego między wariantem eksperymentalnym (V), a grupą kontrolną (C):

$$U = \frac{R_v - R_c}{R_c} \cdot 100\%$$

Wprowadzenie tak znaczące, zaawansowanej architektury analitycznej jest jednak ściśle powiązane z wcześniej omówionymi problemami z domeny inżynierii bezpieczeństwa. W przypadku komunikacji z niezaufanym klientem, będącej częścią większości rozwiązań zdalnej konfiguracji dedykowanym testom A/B, przy wykorzystaniu narzędzi analizy dynamicznej możliwe jest nadpisywanie wartości przesyłanych przez aplikację. W przypadku w pełni umyślnego działania (skierowanego na konkretny skutek biznesowy, a nie działań eksploracyjnych), możliwe staje się przeprowadzenie ataku zatrucia danych – niwecząc sens prowadzonych eksperymentów – i potencjalnie prowadząc do mylnych decyzji biznesowych.

2.8. Pola do innowacji - unikalne cechy proponowanego rozwiązania

Przeprowadzona analiza stanu sztuki w domenie rozwiązań zdalnej konfiguracji uwidacznia fundamentalny problem architektury w kontekście systemów mobilnych – asymetrię rozwoju między mechanizmami z dziedziny logiki biznesowej, a mechanizmami ją chroniącymi. Komercyjne platformy analityczne w większości pomijają aspekt zaufania do klienta mobilnego – co wymusza kompromisy w kwestii klasyfikacji danych, które mogą serwować. Jednocześnie, klasyczne mechanizmy obronne pozostają zupełnie nieskuteczne wobec zdeterminowanych atakujących, skupiając się głównie na stawianiu „zasłon dymnych” – przez które można przejść bez większych problemów.

Dysfunkcje te motywują definicję luki badawczej i architektonicznej na przestrzeni kilku obszarów. Obecnie na rynku brakuje ujednoliconych wzorców projektowych, które pozwoliłyby na traktowanie systemu zdalnej konfiguracji jako środowiska w pełni zaufanego – np. poprzez użycie sprzętowej atestacji środowiska do ustanowienia fundamentu bezpiecznej komunikacji z serwerem. Na tym tle, klaryfikują się główne pola do innowacji, które służą jako bezpośrednia motywacja i wskazówka architektoniczna dla autorskiego, bezpiecznego i elastycznego ekosystemu zdalnej konfiguracji,

będącego przedmiotem niniejszej pracy. Łącznie wyróżnić można cztery kluczowe obszary innowacyjne.

Pierwszym polem do innowacji jest zapewnienie pełnego bezpieczeństwa komunikacji w środowisku anonimowym (*pre-auth*) – gdy użycie poświadczeń użytkownika do udostępnienia informacji z systemu konfiguracji jest niemożliwe. Docelowe rozwiązanie musi kategorycznie odrzucić wykorzystanie statycznych kluczy API, zawartych w kodzie źródłowym aplikacji. Przestrzenią do innowacji jest zaaplikowanie uwierzytelnienia komunikacji poprzez wykorzystanie natywnych mechanizmów opartych o sprzętowy korzeń zaufania, takich jak Play Integrity API lub App Attest. Ponadto, dodatkowym aspektem zabezpieczenia może być kryptograficzne powiązanie zapytania, uniemożliwiając jakkolwiek podszywanie się pod rzeczywistą aplikację – gwarantując matematyczną pewność, że urządzenie, aplikacja i dane żądania nie zostały skompromitowane.

Drugim polem do innowacji jest wysoka niezależność infrastrukturalna. Współczesny rynek dojrzałych rozwiązań zdalnej konfiguracji zdominowany jest przez rozwiązania SaaS, co nie tylko wymusza przesyłanie potencjalnie wrażliwych danych do podmiotów trzecich – ale jest też znaczącym narzutem inżynierskim przy implementacji rozwiązań w kodzie źródłowym aplikacji klienckich. Ryzyko powodowane tymi czynnikami może zostać zmitigowane poprzez adopcję elastycznego rozwiązania w architekturze *self-hosted* – nieposiadającego zewnętrznych zależności.

Trzecim obszarem eksploracji jest implementacja wyłącznika awaryjnego – czyli możliwość dystrybucji konfiguracji w czasie rzeczywistym. Większość rozwiązań opiera się na zasobochłonnym i podatnym na błędy odpytywaniu serwera. Innowacją może być wdrożenie architektury *Server-Sent Events* lub *WebSocket* w celu wykonania aktualizacji flag tylko w momencie, kiedy jest to potrzebne – bez nadmiernego drenażu baterii.

Czwartym, istotnym z perspektywy inżynierskiej aspektem jest zapewnienie w pełni natywnej integracji i ergonomii pracy programisty. Znacząca część komercyjnych pakietów SDK wymaga od programistów odwoływania się do flag funkcji za pomocą surowych ciągów znaków. Jest to podejście podatne na błędy (literówki) i trudne w utrzymaniu. Przestrzenią na innowację jest stworzenie głęboko natywnych bibliotek dla systemów iOS i Android, które poprzez silne typowanie dostarczą pełne wsparcie dla mechanizmów autouzupełniania kodu oraz automatycznej propagacji flag na serwery deweloperskie – aby nie musiały być konfigurowane w dwóch miejscach jednocześnie.

Podsumowując, zaadresowanie wymienionych aspektów – od suwerenności danych, przez kryptografię sprzętową, aż po ergonomię API – stanowi wyzwanie wymierne wyzwanie inżynierskie. Proces projektowania i implementacji środowiska odpowiadającego na te postulaty opisany został w następnym rozdziale.

3. Propozycja bezpiecznego systemu zdalnej konfiguracji

Rozdział ten prezentuje architekturę oraz projekt funkcjonalny prototypu autorskiego systemu zdalnej konfiguracji i monitorowania aplikacji mobilnych. Zakłada on zdefiniowanie głównych aktorów, przypadków użycia oraz modelowanie przepływu danych między serwerem a klientami mobilnymi ze szczególnym uwzględnieniem mechanizmów bezpieczeństwa.

3.1. Projekt funkcjonalny i przypadki użycia

Celem poprawnego zamodelowania zachowania systemu oraz jego przypadków użycia, jednym z pierwszych kroków jest definicja aktorów, którzy będą wchodzić w interakcję z systemem. W przypadku systemu zdalnej konfiguracji, właściwym podejściem jest wprowadzenie rozróżnienia między aktorem (rzeczywistym), a aktorem systemowym (działającego jako odrębne urządzenie lub system). Według tych pryncypiów, dla systemu konfiguracji zdalnej dedykowanego aplikacjom mobilnym można wyróżnić następujących aktorów:

- **Właściciel produktu** – funkcjonalny administrator zestawu flag konfiguracji zdalnej. Jego głównym celem jest możliwość dynamicznego udostępniania poszczególnych elementów systemu poprzez flagi funkcji. Jako cel dodatkowy, należy wziąć pod uwagę potrzebę aktora o dowiadrywaniu się większej ilości rzeczy o produkcie – przeprowadzanie eksperymentów, walidacja pomysłów wobec użytkowników,
- **Programista** – częściowy administrator zestawu flag funkcji. Z racji roli, skupia się na tworzeniu nowych flag z szerokim spektrum dostępnych parametrów – tak aby mogły być przydatne przy sterowaniu wieloma rodzajami wartości programistycznych. Poza samą akcją stworzenia flag, wchodzi w interakcję z systemem, gdy chce przetestować konkretną konfigurację na środowisku testowym (włączając flagę funkcji) lub wymusić natychmiastową aktualizację konfiguracji na środowisku produkcyjnym – np. w przypadku incydentu wymagającego ukrycia problematycznej funkcjonalności.

W przypadku w pełni produkcyjnego systemu należy rozważyć wszelkie role kontrybuujące do bądź wchodzące w interakcje z projektem IT (np. pracownicy działu marketingu) – po pozytywnej ewaluacji, mogą stać się interesariuszami i rzeczywistymi aktorami w modelu systemu. Ze względu na prototypowy charakter tworzonego systemu, lista aktorów zostaje ograniczona do właściciela produktu oraz programisty – gdyż agregują oni wszystkie najważniejsze cele systemu zdalnej konfiguracji.

Aby poprawnie realizować założone cele systemu, konieczne jest również rozpatrzenie aplikacji mobilnych jako aktorów – aplikacje dedykowane platformom iOS i Android mogą być na tym etapie zagregowane do jednego aktora:

- **Aplikacja Mobilna (klient systemu zdalnej konfiguracji)** – jej podstawowe interakcje opierają się na pobieraniu (otrzymywaniu) dedykowanego zestawu flag funkcji. Należy założyć ewentualność, że to nie będzie jedyna interakcja systemu aplikacji mobilnej – mogą być wymagane dodatkowe funkcje, wraz z eksploracją konkretnych przypadków użycia.

Będąc świadomym zestawu aktorów który należy obsłużyć, możliwa jest praca nad przypadkami użycia przez nich wykonywanymi.

3.1.1 Modelowanie procesu zarządzania aplikacjami

Mając na uwadze główne cele systemu, czyli dostarczenie możliwości zarządzania funkcjonalnościami poszczególnych aplikacji oraz przeprowadzania testów A/B, uprzednio należy zdefiniować wszelkie przypadki użycia będące wymaganymi krokami przed ich faktyczną realizacją. Wobec tego – faktem jest, iż zanim zdefiniowane zostaną flagi funkcji dla poszczególnych aplikacji, najpierw powinna zostać zdefiniowana sama aplikacja.

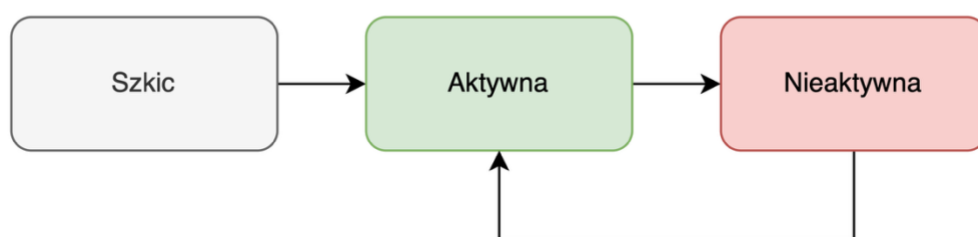
Naturalnie, można potraktować tworzone flagi w ramach systemu globalnie – jednakże wiązałoby się to ze znacznymi niedogodnościami w domenie doświadczeń użytkownika. Aby rozpoznawać flagi powiązane z konkretną aplikacją i tak należałoby wprowadzić pewne konwencje nazewnictwa (np. nazwa-flagi_nazwa-aplikacji_platforma) – lecz operowanie na takim dodatkowym poziomie abstrakcji byłoby dosyć sztuczne i podatne na błędy użytkownika.

Aby uniknąć takich niedogodności, właściwe będzie wprowadzenie mechanizmu definicji i zarządzania aplikacją jako bytem w systemie. Pierwszą operacją do implementacji jest możliwość stworzenia aplikacji. W celu akomodacji wieloplatformowości rozwiązania, aplikację można zdefiniować przez jej:

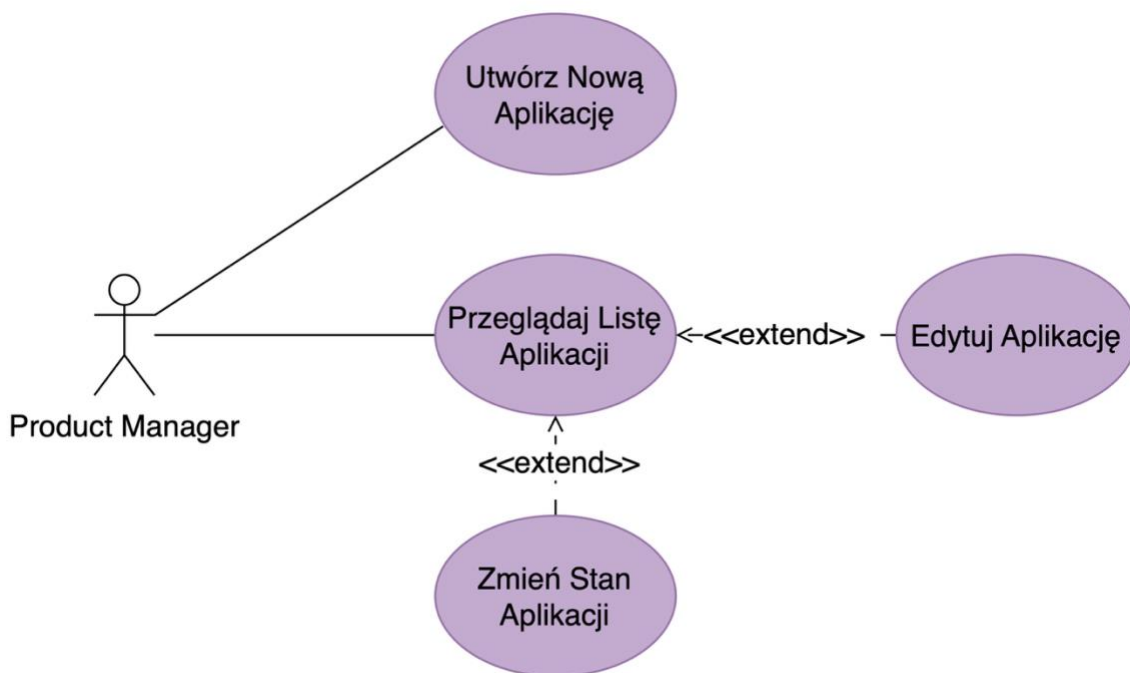
- Nazwę,
- Identyfikator Aplikacji Android,
- Identyfikator Aplikacji iOS.

Przy czym zakłada się, że każda z tych wartości musi być unikalna – a identyfikatory systemowe muszą być wprost powiązane z tymi wymaganymi przez mobilne systemy operacyjne (*package names*).

Oprócz samej logiki tworzenia aplikacji, konieczne jest również udostępnienie widoku wyświetlającego wszystkie dotychczas stworzone aplikacje (lub te, do których konkretny użytkownik ma dostęp). Warto również uwzględnić możliwość zarządzania stanem aplikacji – tym samym udostępniając możliwość „miękkiego usunięcia” poprzez zmianę jej stanu. Przypadki te, ujęte zostały na diagramie przypadków użycia (Rysunek 11). W celu pokrycia jak największej liczby przypadków użycia, można zdefiniować trzy ogólne stany obiektu w całym systemie – szkic, aktywny oraz dezaktywowany, przedstawiane przez Rysunek 10.



Rysunek 10. Schemat blokowy ilustrujący logikę stanów flagi funkcji



Rysunek 11. Diagram przypadków użycia powiązanych z procesem zarządzania aplikacjami

3.1.2 Modelowanie procesu zarządzania funkcjonalnościami

Po utworzeniu danej aplikacji, możliwe jest przystąpienie do konfiguracji jej funkcjonalności. Jednocześnie, można założyć, że wszelkie akcje, które będą wykonywane w systemie, zawsze będą odnosiły się do pewnej aplikacji – a więc konieczne będzie umożliwienie wyboru kontekstu działań, a następnie zachowanie go między poszczególnymi stronami panelu administracyjnego. Wszystkie opisywane akcje ilustrowane są na diagramie przypadków użycia – Rysunek 12.

Wobec tego, przypadek użycia konfiguracji flagi rodzi się z samego przypadku wyboru aplikacji. Następnie, należy określić minimalny zestaw wartości pozwalający na sprawne zarządzanie flagą. W tym przypadku proponowane są następujące parametry:

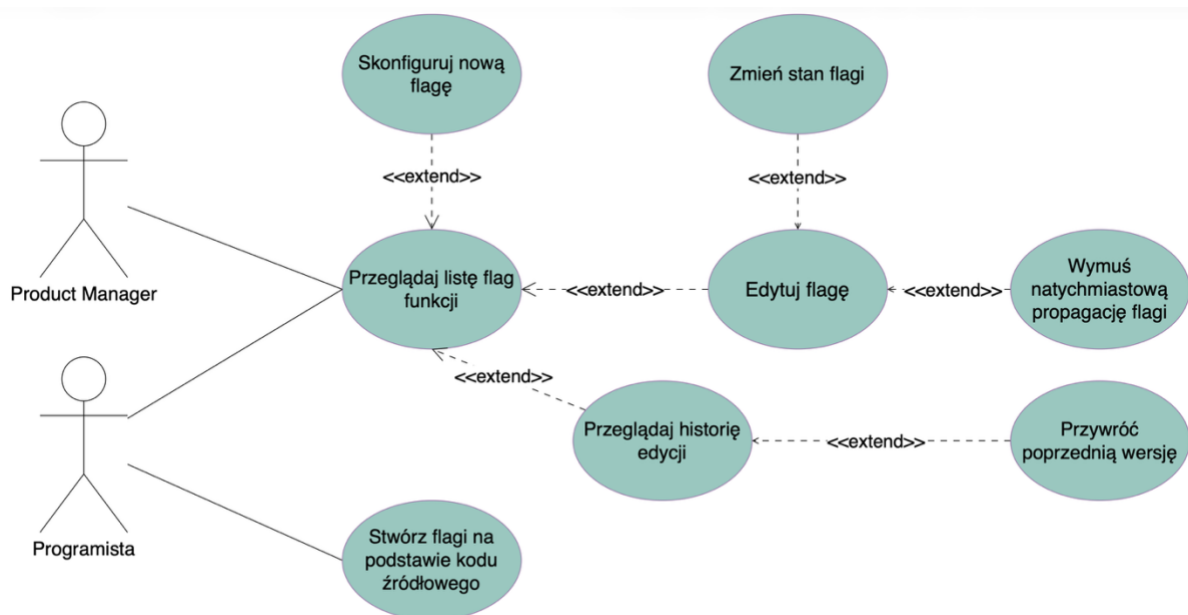
- **Klucz** (Identyfikator o znaczeniu technologicznym),
- **Opis** (Stricte biznesowy / operacyjny),
- **Typ wartości** (Również stricte techniczny parametr, potencjalnie ułatwiający obsługę wartości w bibliotece mobilnej) – np. łańcuch znaków, liczba, wartość Boola,
- **Wartość**,
- **Platforma mobilna**, do której koresponduje flaga – iOS lub Android,
- **Stan** – włączona / wyłączona – z potencjalnym reużyciem maszyny stanu szkic-aktywny-dezaktywowany.

W celu zachowania jak najwyższego poziomu bezpieczeństwa operacyjnego, każda nowo powstająca flaga powinna być w stanie szkicu / dezaktywacji. Włączenie powinno być intencjonalną akcją, wykonaną przez interesariusza ze strony biznesu.

Jako poprawę doświadczenia programisty w korzystaniu z systemu, warte rozważenia jest również automatyczne tworzenie flag na podstawie zawartości kodu – w tym scenariuszu, interesariusz biznesowy musi jedynie zamieścić informację w kryteriach akceptacji danej funkcjonalności bądź parametru, iż powinna być ona konfigurowalna za pośrednictwem panelu administracyjnego.

Programista, implementując funkcjonalność, zamieści odpowiednie znaczniki w kodzie, tworząc niejako szkic flagi funkcji. Następnie, flaga ta może być odczytana i rozpropagowana do panelu administracyjnego w czasie kompilacji lub wykonania. Zdejmuje to z użytkownika biznesowego oraz programisty obowiązek konfiguracji flag w zupełnej separacji – proces zostaje w dużej mierze zautomatyzowany, niwelując potencjalne problemy wywołane przez np. niezgodność klucza flagi między panelem administracyjnym a kodem źródłowym aplikacji mobilnej.

Ponadto, warte rozważenia jest również umożliwienie rewizji historii edycji flag. W przypadku wdrożenia produkcyjnego, funkcjonalność ta jest wręcz niezbędna. W przypadku realizacji prototypu, potraktować ją można jako mile widzianą, lecz nie wymaganą do demonstracji pomysłu.



Rysunek 12. Diagram przypadków użycia powiązanych z procesem zarządzania flagami funkcji

3.1.3 Modelowanie sterowania testami A/B

Testy porównawcze funkcjonalności są ostatnią warstwą abstrakcji systemu – zawsze muszą być powiązane z istniejącą aplikacją oraz pewnym zestawem flag funkcji – sugerując jednocześnie potencjalną chronologię implementacji oraz kolejność akcji użytkownika w interfejsie graficznym.

Aby poprawnie zdefiniować test A/B, konieczne jest określenie jego parametrów – są to:

- **Nazwa** – do łatwiej identyfikacji w panelu administracyjnym,
- **Opis** – służący zrozumieniu warunków i celów testu,
- **Data rozpoczęcia**,
- **Data zakończenia**,
- **Testowane warianty**.

Wobec przytoczonych parametrów, przydatne jest również określenie struktury danych formułującej wariant. W minimalnej wersji, można założyć, że wariant składa się z:

- **Nazwy** – przydatnej do wizualizacji danych powiązanych z wariantem,
- **Flag funkcji** – w relacji jeden-do-wielu, pozwalając na nadpisanie wartości dostarczanych za pośrednictwem danej flagi,

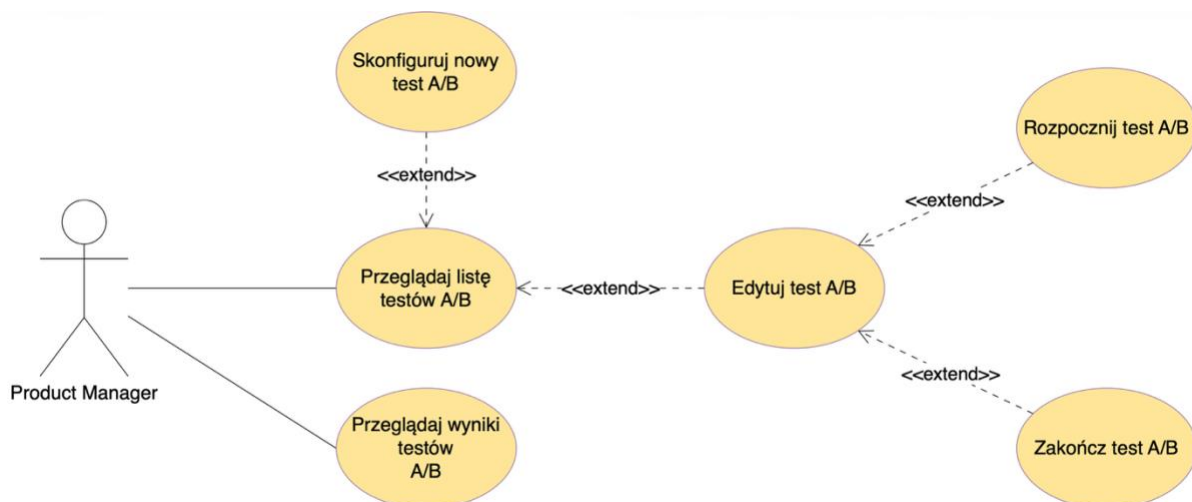
- **Wagi** – parametru sterującego przydziałem wariantu do użytkownika. Suma wag musi wynosić 100%.

Po uprzednim wyborze parametrów i spełnienia warunków brzegowych – tj. daty rozpoczęcia testu, aplikacje muszą zacząć otrzymywać nowe, nadpisane wartości.

Dopełnieniem przypadku użycia jest procedura ekstrakcji danych z testu. Każdy test A/B powinien być podsumowywany w przyjaznej użytkownikowi postaci w ramach konsoli administracyjnej. Podążając za tym wymaganiem, konieczna jest również możliwość definiowania impresji oraz wyświetlania ich statystyk w zagregowanej formie. Jako trzy podstawowe wartości, które mogą być wartościowe do biznesu, zgodnie z analizą stanu sztuki, definiuje się:

- Absolutną liczbę impresji,
- Współczynnik Konwersji,
- Wskaźnik Przyrostu.

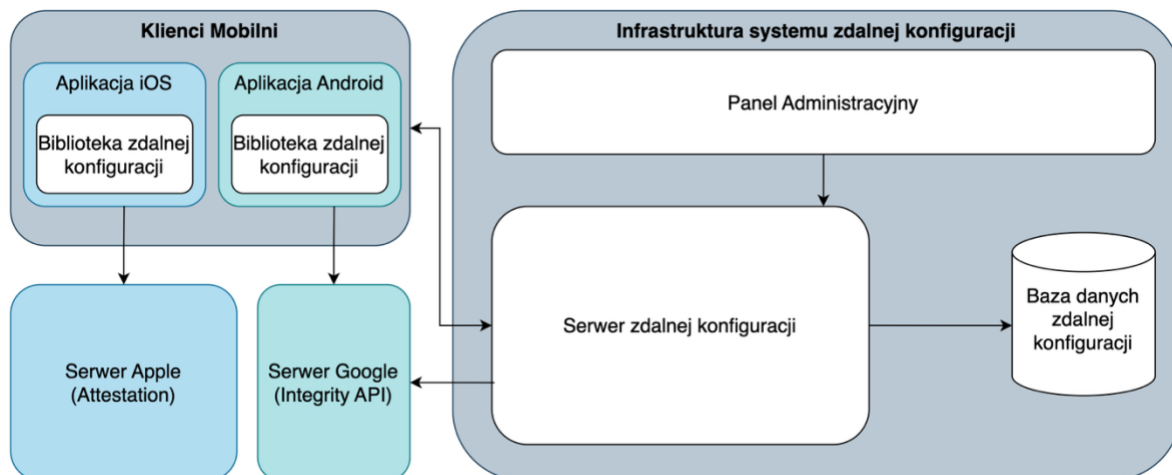
Każdy z tych wykresów powinien klarownie prezentować dane korespondujące z każdym wariantem – pozwalając na porównywanie ich ze sobą. Pełen krajobraz opisanych przypadków użycia zilustrowany został przez Rysunek 13.



Rysunek 13. Diagram przypadków użycia powiązanych z przeprowadzaniem testów A/B

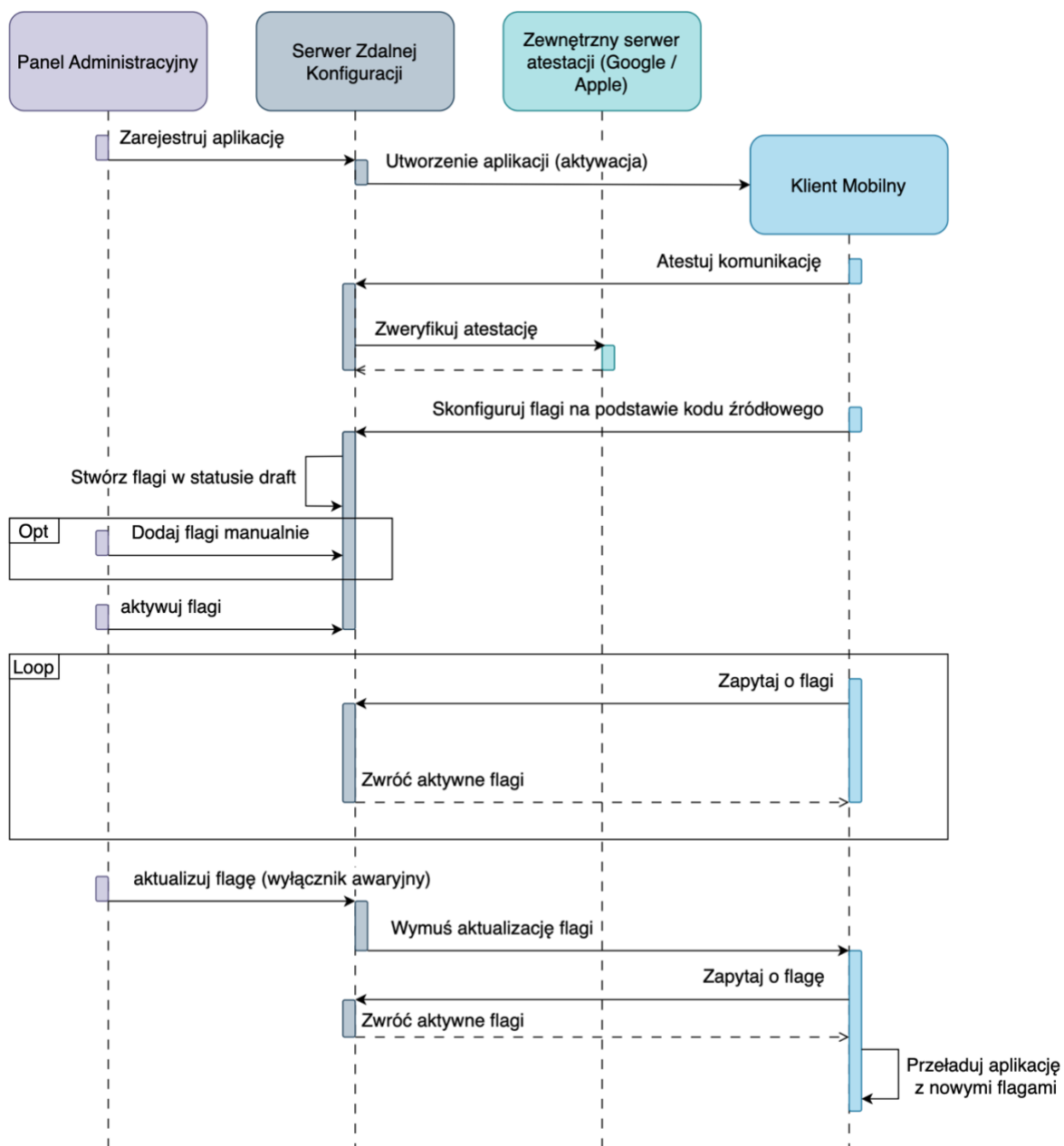
3.2. Architektura systemu i przepływ danych

Prawidłowa i sprawna integracja wieloskładnikowego systemu wymaga nie tylko klaryfikacji i przygotowania przypadków użycia – jednym z kluczowych kroków sterujących późniejszymi wysiłkami programistycznymi są przepływy danych, modelowane w formie diagramów sekwencji UML. Celem sformułowania poprawnego modelu, wpraw należy zilustrować ogólną architekturę rozwiązania (Rysunek 14).

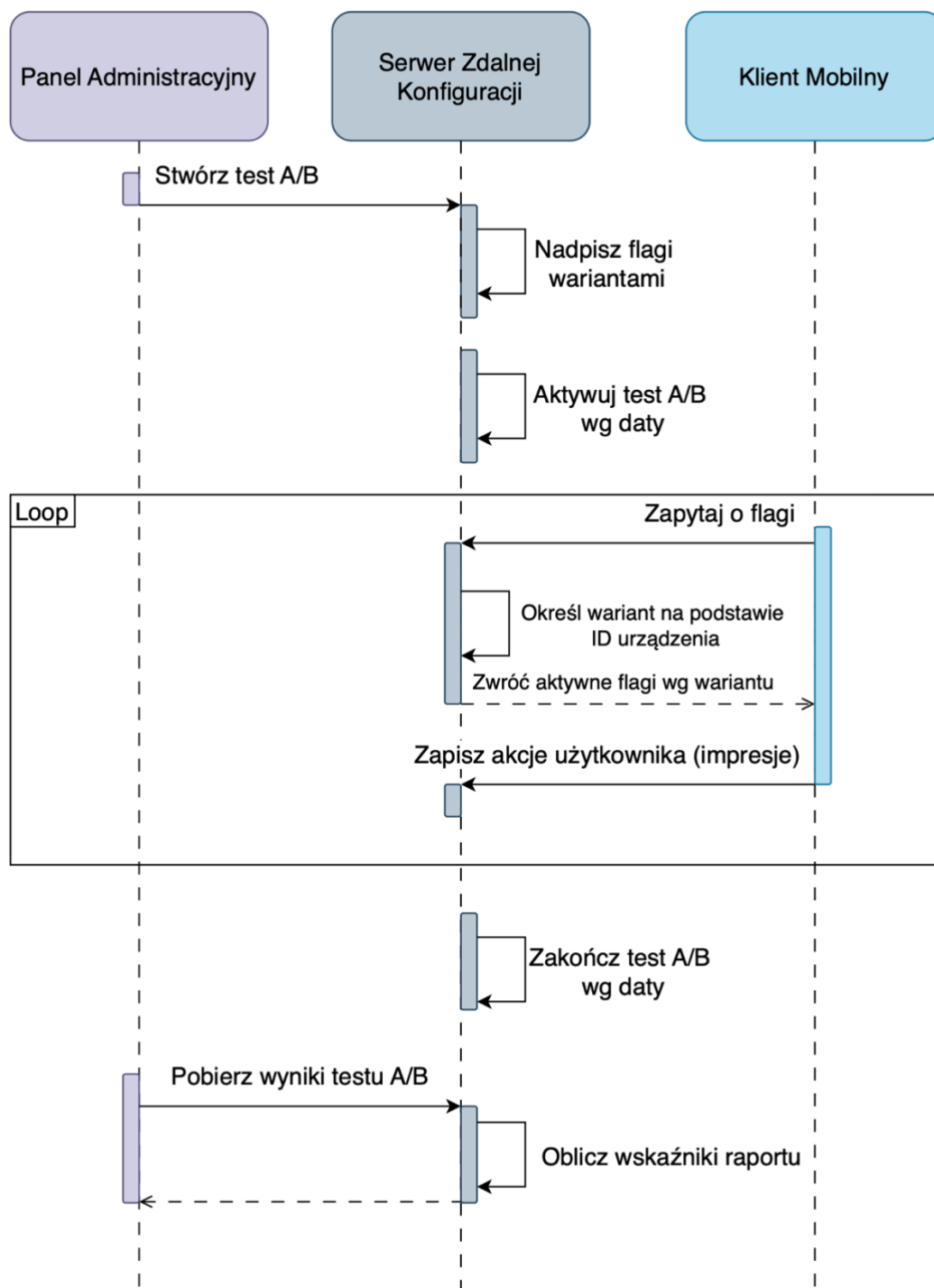


Rysunek 14. Diagram komponentów systemu zdalnej konfiguracji

Zgodnie z architekturą, działanie systemu opiera się na interakcjach między kilkoma partiami – przepływy te można zilustrować w podziale na kategorie przypadków użycia flag funkcji (z przepływem zilustrowanym przez Rysunek 15) oraz testów A/B (Rysunek 16). Proces tworzenia aplikacji jako trywialny nie wymaga dodatkowej dokumentacji. Diagramy, mając na uwadze wysoki poziom abstrakcji oraz zachowanie czytelności przepływu logiki biznesowej między oddalonymi węzłami systemu, na rysunkach 15 i 16 pominięto techniczne komunikaty powrotne, o ile nie niosą one ze sobą nowych danych niezbędnych do kontynuacji procesu.



Rysunek 15. Diagram sekwencji ilustrujący zależności między komponentami w realizacji przepływów danych dotyczących kluczowych przypadków użycia w domenie flag funkcji



Rysunek 16. Diagram sekwencji ilustrujący proces przeprowadzania testów A/B – ilustracja pełnego drzewa przypadków użycia

3.3. Mechanizmy bezpieczeństwa

Aby zapewnić najwyższy możliwy poziom bezpieczeństwa rozwiązania – odróżniający je od rozwiązań dostępnych na rynku, konieczne jest uwzględnienie szeregu funkcji zapewniających należyty poziom bezpieczeństwa, zgodny z wytycznymi omówionymi w rozdziale o stanie sztuki.

3.3.1 Uwierzytelnienie - systemy weryfikacji integralności sprzętowej

Celem realizacji najwyższego możliwego poziomu uwierzytelnienia urządzeń mobilnych bez uwzględnienia sesji użytkownika, właściwe jest zastosowanie natywnych funkcji weryfikacji integralności sprzętowej – Apple App Attest oraz Google Play Integrity API. Implementacja wymagana jest w zakresie dwóch komponentów systemu: natywnej biblioteki mobilnej (używającej sprzętowych modułów kryptograficznych do przygotowania ładunku uwierzytelniającego) oraz serwera (weryfikującego zapytania klienta mobilnego w oparciu o zewnętrzne serwisy lub certyfikaty Root).

3.3.2 Autoryzacja

Proponowany system w sposób naturalny kreuje role użytkowników, które powinny być weryfikowane we właściwym dla nich miejscu – w tym przypadku, w panelu administracyjnym. Docelowo, konieczne by było stworzenie przynajmniej trzech ról:

- **Programista** (z możliwością tworzenia flag wraz ze zdolnością zarządzania nimi),
- **Interesariusz biznesowy** (z możliwością modyfikacji flag oraz tworzenia i ewaluacji testów A/B),
- **Administrator** (posiadający możliwość nadawania ról oraz tworzenia aplikacji).

Naturalnie, możliwa jest bardziej szczegółowa definicja ról, powiązana z konkretnymi stanowiskami i ich poziomem odpowiedzialności, na przykład Analityk Biznesowy lub Właściciel Produktu.

Z racji powtarzalności implementacji oraz wtórności struktury rozwiązań autoryzacyjnych w produktach już obecnych na rynku, wymagania te muszą zostać zrealizowane tylko w przypadku produkcyjnego uruchomienia systemu. Na potrzeby prototypu, implementacja kontroli dostępu opartej o role pozostaje poza zakresem realizacji.

3.3.3 Kryptografia

Z racji zastosowania natywnych technik atestacji, w sposób zaufany realizowany jest kluczowy postulat specyfikacji mTLS, zakładający nie tylko uwierzytelnienie autentyczności serwera wobec klienta, ale również klienta wobec serwera. Wobec tego, standardowe szyfrowanie komunikacji odbywające się na podstawie negocjacji będących częścią implementacji protokołu HTTPS jest wystarczające do zapewnienia odpowiedniego poziomu bezpieczeństwa.

Celem eksploracji maksymalnego możliwego poziomu zabezpieczenia (szczególnie przydatnego w przypadku dostarczania wartości sekretnych), rozpatrzyć należy opcję asymetrycznego szyfrowania end-to-end, np. z użyciem klucza publicznego urządzenia użytego do poświadczenia App Attestation. Środek ten znany jest m.in. z architektury aplikacji bankowych – i w przypadku systemu zdalnej konfiguracji może być uznany za nadmiarowy.

4. Opis narzędzi stosowanych w pracy

Rozdział ten zawiera omówienie kluczowych technologii wykorzystanych do realizacji prototypu autorskiego, bezpiecznego i elastycznego systemu zdalnej konfiguracji. Dobór narzędzi podyktowany był przede wszystkim dotychczasowym poziomem zaznajomienia z nimi, elastycznością zastosowania, wygodą integracji oraz poziomem bezpieczeństwa. Następujące podrozdziały charakteryzują stos technologiczny całości rozwiązania z podziałem na komponenty serwerowe (*backendowe*), implementację panelu administracyjnego (*frontend* CMS), technologie mobilne oraz sposoby zapewnienia bezpieczeństwa aplikacji.

4.1. Narzędzia backendowe

Warstwa serwerowa stanowi centralny punkt systemu zdalnej konfiguracji – to ona odpowiada za realizację logiki zarządzania i dostarczania konfiguracji, tworzenie i agregację telemetrii z testów A/B oraz weryfikację dowodów atestacyjnych jako zabezpieczenia komunikacji z klientami mobilnymi.

4.1.1 Frameworki serwerowe

Językiem programowania wykorzystanym do implementacji serwera jest Java w wersji 17 [63], będąca jedną z wersji pokrytych długoterminowym wsparciem – stanowiąc jednocześnie popularny wybór przedsiębiorstw przy realizacji nowych systemów. Język ten nie tylko utrzymuje się w czołówce najpopularniejszych języków świata – pomimo długiej historii jest również wysoce popularnym językiem oferującym sprawne mechanizmy zarządzania pamięcią, wsparcie wielowątkowości oraz rozbudowany ekosystem bibliotek. Ponadto, dzięki działaniu w ramach JVM, może być uruchamiany na dowolnej architekturze serwerowej.

Sam serwer został zbudowany w oparciu o środowisko Spring Boot w wersji 3.5.0 (najnowszej w momencie rozpoczęcia pracy) [64]. Oferuje on gotowe implementacje najbardziej przydatnych komponentów w przypadku realizacji serwerów aplikacji webowych – wbudowany serwer web Tomcat [65], mechanizmy budowania projektu i zarządzania zależnościami poprzez m.in. wstrzykiwanie ich do obiektów. W ramach projektu wykorzystano następujące moduły – rozszerzenia Spring Boot:

- **Spring Web:** pozwalający na sprawne wystawienie interfejsu REST z pomocą użycia odpowiednich anotacji w kodzie – pozwalających na faktyczną obsługę żądań płynących od klientów [64],
- **Spring Security:** ułatwia on zarządzanie kontrolą dostępu do aplikacji – pozwalający między innymi na szybką konfigurację chronionych zasobów, kontrolę parametrów CORS oraz implementację ochrony przed CSRF [64],
- **Lombok:** biblioteka znacząco redukująca tzw. *boilerplate code*, np. poprzez dostarczanie konstrukcji umożliwiających manipulację danymi obiektu oraz ukrycie konstruktorów [66].

Całość procesu budowania aplikacji, zarządzania zależnościami i uruchamiania testów jednostkowych jest orkiestrowana przy pomocy systemu Gradle [67].

4.1.2 Technologie bazodanowe

Jako że komponent serwerowy do poprawnego działania wymaga możliwości przechowywania danych o zdefiniowanych flagach funkcji, danych analitycznych, bądź pokrewnych danych aplikacji, konieczne było wdrożenie relacyjnej bazy danych.

W celu sprawnego rozwoju systemu, użyto bazę danych H2 [68], działającą całkowicie w ramach pamięci operacyjnej urządzenia, na którym uruchamiany jest projekt. Choć jest ona wystarczająca do celów programistycznych i demonstracyjnych, w przypadku produkcyjnego uruchomienia systemu należy rozpatrzyć zastąpienie jej inną bazą o trwałej pamięci. W tym przypadku, pełna zgodność ze standardem SQL otwiera szeroki wachlarz rozwiązań produkcyjnych – ze szczególnym wskazaniem na PostgreSQL [69] ze względu na jej uniwersalność i otwarte środowisko.

4.1.3 Protokoły komunikacyjne

Aby zapewnić wysoki poziom interoperacyjności między stroną serwera a klientami web i mobile, architekturę komunikacji oparto na standardach wykorzystujących protokoły HTTP/HTTPS.

Podstawową metodą wymiany danych jest API REST [70], realizując możliwość klasycznej, asynchronicznej komunikacji. W ramach tej metody realizowane są wszystkie operacje dostępne z poziomu webowego panelu administracyjnego oraz część komunikacji z klientami mobilnymi. Jako uniwersalną strukturę danych ładunku zapytań ustanowiono obiekty JSON [71] – zachowując znaczną elastyczność i prostotę mapowania obiektów między środowiskami.

Nie jest to jedyna przyjęta metoda komunikacji – w celu realizacji wymogu natychmiastowej propagacji zmian (wyłącznik awaryjny) dla klientów mobilnych, konieczne było zrezygnowanie z zapytań asynchronicznych. Zamiast tego, zaimplementowano standard *Server-Sent Events* [72]. Pozwala on na utrzymywanie długotrwałego, jednokierunkowego strumienia danych między serwerem a aplikacją mobilną. Dzięki temu, możliwe jest natychmiastowe wypchanie konfiguracji do zarejestrowanych klientów – bez opóźnień, narzutu zapytań i drenażu baterii.

4.2. Technologie frontendowe

Graficzny interfejs użytkownika pełniący rolę konsoli administracyjnej systemu został zrealizowany w architekturze *Single Page Application*, z użyciem zmodyfikowanej biblioteki komponentów TailAdmin [73]. Zastosowanie tego wzorca pozwala na osiągnięcie wysokiej responsywności interfejsu, redukując konieczność przeładowywania strony w trakcie nawigacji.

Warstwa prezentacyjna implementowana jest przez framework React w wersji 19 [74] – wprost łącząc się z językiem logiki aplikacyjnej, która została napisana w języku TypeScript [75]. Wybór języka typowanego motywowany był ułatwionym mapowaniem struktur danych oraz ogólną wygodą pracy – pozwalając na uniknięcie części błędów już na etapie kompilacji.

Proces budowania aplikacji zarządzany jest z kolei przez środowisko Vite [76] – minimalizując czas uruchamiania i przebudowywania aplikacji w trakcie procesu wytwórczego. Oprócz tego, poszczególne aspekty inżynieryjne aplikacji zostały zrealizowane następującymi narzędziami:

- **Warstwa komunikacyjna** – Axios [77],
- **Stylowanie** – Tailwind CSS [78] (jako część biblioteki TailAdmin),
- **Zarządzanie stanem nawigacji** – React Router [79],
- **Wizualizacja Danych Analitycznych (testów A/B)** – ApexCharts [80].

Ponadto, całość projektu posiada skonfigurowany mechanizm statycznej analizy jakości kodu ESLint [81].

4.3. Technologie mobilne

W ramach projektu realizowane są nie tylko autorskie biblioteki umożliwiające sprawną integrację klienta mobilnego z systemem zdalnej konfiguracji – na potrzeby ich ewaluacji powstały przykładowe aplikacje mobilne opierające się w całości o technologie natywne. Metody te przeważały nad alternatywami wieloplatformowymi ze względu na prostotę i pełne odwzorowanie integracji ze sprzętowymi modułami kryptograficznymi, bez potrzeby użycia bibliotek utrzymywanych przez społeczność.

4.3.1 Technologie iOS

Do przygotowania biblioteki oraz przykładowej aplikacji działającej na systemie iOS użyty został język Swift [82] – będący w momencie pisania pracy standardem realizacji aplikacji na tę platformę. Warstwa prezentacyjna aplikacji testowej została w całości zbudowana z użyciem frameworka SwiftUI [83] – umożliwiając bardzo dynamiczne, deklaratywne prototypowanie interfejsów graficznych, jednocześnie wiążąc widoki ze stanem aplikacji. Podejście to oznacza, że każda zmiana wartości flagi konfiguracyjnej automatycznie przebuduje docelowe komponenty ekranu bez konieczności dodatkowego odświeżania.

W domenie bezpieczeństwa zastosowano systemowe biblioteki DeviceCheck oraz App Attest API, które generują pary kluczy asymetrycznych wewnątrz *Secure Enclave* i uwierzytelniają się wobec serwera z użyciem obiektów asercji podpisanych tymże kluczami.

Oprócz tego, do realizacji strumieniowania danych przy wyłączniku awaryjnym została użyta natywna klasa `URLSession` po odpowiedniej konfiguracji.

4.3.2 Technologie Android

Aplikacja kliencka wraz z dedykowaną biblioteką dla platformy Android została zaimplementowana w języku Kotlin [84] – również stanowiącego standard współcześnie powstających natywnych aplikacji mobilnych, zajmując miejsce uprzednio dominującej Javie.

Warstwa prezentacji opiera się na bibliotece Jetpack Compose [85], pozwalającej na zwinne, deklaratywne prototypowanie interfejsów użytkownika i łączenie ich ze stanem aplikacji (włączając w to stan flag funkcji). Podobnie jak w przypadku iOS, pozwala to na warunkowe wyświetlanie i animowanie widoków w czasie rzeczywistym, zawsze biorąc pod uwagę dane płynące z serwera konfiguracyjnego – bez potrzeby pełnej aktualizacji drzewa, dawniej wymuszanej przez architektury oparte o pliki XML.

W celu propagacji flag w czasie rzeczywistym, aplikacja zawiera mechanizm nawiązywania połączenia SSE, używając standardowych bibliotek do komunikacji sieciowej (okhttp) oraz Kotlin Coroutines i reaktywnych strumieni danych (StateFlow) w celu płynnego przekazywania danych do interfejsu użytkownika. Ponadto, wsparciem dla komunikacji jest biblioteka Google Gson [86], wykorzystywana do serializacji i deserializacji przesyłanych struktur danych.

Funkcje bezpieczeństwa realizowane są w oparciu o lokalne implementacje logiki Play Integrity API, wykorzystując komponenty Android Keystore do przygotowania obiektów atestacji.

5. Prototyp systemu zdalnej konfiguracji i monitorowania

Niniejszy rozdział poświęcony jest szczegółowemu opisowi zrealizowanego prototypu autorskiego systemu zdalnej konfiguracji który materializuje uprzednio zdefiniowane założenia projektowe. Głównym celem tego etapu pracy jest weryfikacja paradygmatu tworzenia dedykowanego systemu konfiguracji dla aplikacji mobilnych oraz jego poszczególnych elementów, takich jak natychmiastowa propagacja flag lub bezpieczeństwo przez atestację.

Prezentowany opis celowo unika całościowego przytaczania kodu źródłowego, skupiając się na kluczowych komponentach systemu, jego logice biznesowej oraz mechanizmach integracji między aktorami systemu.

Z racji prototypowego charakteru implementacji, w architekturze zaakceptowano niektóre świadome kompromisy technologiczne, mające na celu minimalizację złożoności środowiska przy jednoczesnym zachowaniu pełnej wierności funkcjonalnej w obszarze bezpieczeństwa, zarządzania flagami funkcji oraz testami A/B. Wymagania oraz kierunki transformacji obecnego prototypu w system w pełni produkcyjny zostały szczegółowo zaadresowane w rozdziale 6.

5.1. Implementacja serwera oraz mechanizmu natychmiastowej propagacji zmian

Centralnym punktem całego ekosystemu zdalnej konfiguracji jest serwer dystrybucyjny – silnik logiczny, pośredniczący między danymi wpływającymi przez panel administratora a bibliotekami zintegrowanymi w aplikacjach mobilnych. Architektura serwera, zgodnie z klasycznymi schematami systemów wytwarzanych przy pomocy języka Java wraz z środowiskiem Spring Boot, charakteryzuje się warstwowym podziałem. Wydzielone zostały następujące pakiety:

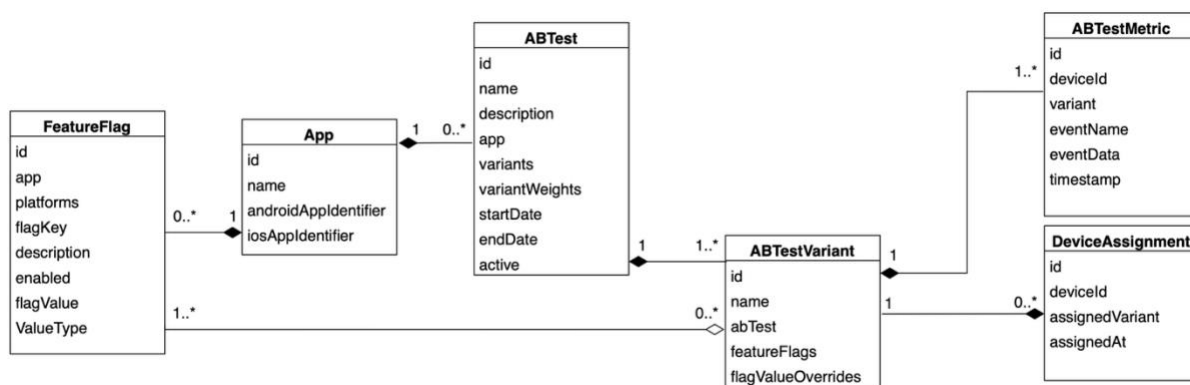
- **Config** – odpowiadający za pliki konfiguracyjne biblioteki Spring Security, ograniczające komunikację z API serwera,
- **Controller** – przechowujący wszystkie definicje końcówek REST API serwera, skupiając się jedynie na kontrakcie komunikacji,
- **Service** – gdzie znajdują się definicje serwisów, implementujących zarówno szczegóły logiki biznesowej (takiej jak tworzenie i zarządzanie flagami), jak i aspekty bezpieczeństwa (mechanizmy atestacji po stronie serwera),
- **Repository** – zachowujący ostatnią warstwę pośredniczącą między serwisami a bazą danych – definicje i implementacje repozytoriów ORM,
- **Model** – zawierający definicję wszystkich encji baz danych zgodnie z logiką mapowania obiektowo-relacyjnego,
- **Dto** – przechowujący uproszczony model danych, adekwatny do ich ekspozycji poprzez końców REST API.

Podsumowanie całości krajobrazu implementacji przepływu danych, wraz z rozróżnieniem między jej warstwami i poszczególnymi klasami, zilustrowane zostało na diagramie architektury ilustrowanym przez Rysunek 18.

5.1.1 Warstwa przechowywania danych

Na potrzeby prototypu, całość danych przechowywana jest w bazie H2, natomiast komunikacja z nią obsługiwana jest przez interfejsy dziedziczące po klasach implementujących Jakarta Persistence API. Część z funkcjonalności wymagała rozszerzenia klas o dodatkowe metody napisane zgodnie z dialektem Hibernate. W przypadku większości interfejsów, pominięto przygotowywanie natywnych kwerend w SQL – było to nadmiarowe w przypadku prototypu.

Struktura bazy danych jest narzucana przez mapowanie obiektowo-relacyjnie zgodnie ze schematem określonym przez Rysunek 17.

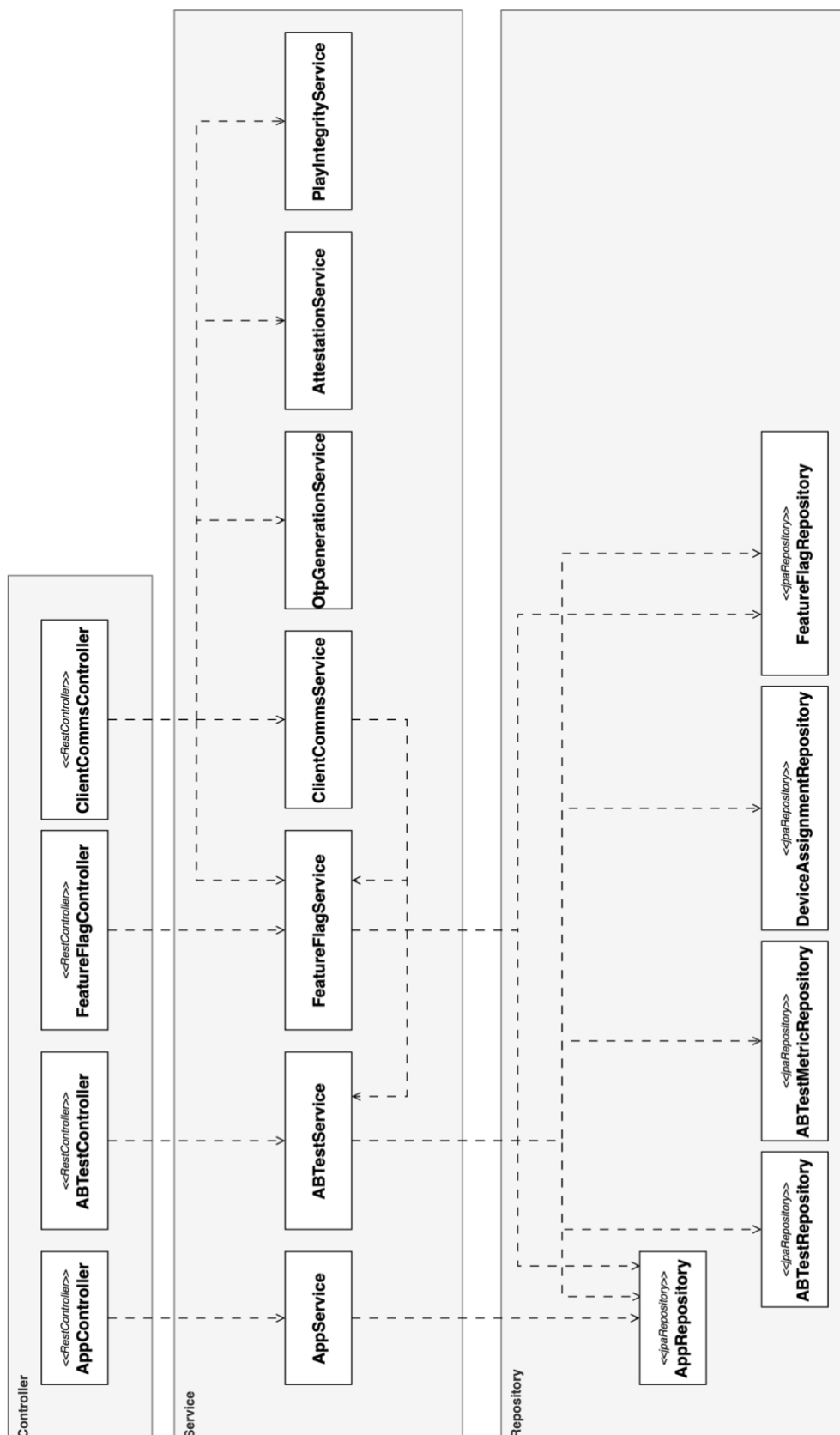


Rysunek 17. Diagram klas przedstawiający model danych systemu

Na poziomie definicji klas encji, zastosowana została bezpośrednia walidacja danych z użyciem biblioteki Jakarta Validation. Dodatkowo, część parametrów implementuje logikę ładowania *Lazy*, co pozwala na uniknięcie narzutu wydajnościowego powodowanego przez problem N+1, na przykład w przypadku ładowania tabeli flag funkcji. Wdrożona została również kaskadowa polityka czyszczenia danych w relacjach, które tego wymagają – przykładowo, usunięcie całego testu A/B wymusza również usunięcie wszystkich powiązanych z nim wariantów.

Pośród rzeczy, które nie zawierają się bezpośrednio w kodzie modelu, lecz wpływają na politykę zarządzania danymi – wszelkie akcje wpływające na zmianę danych płynących do klientów mobilnych zaopatrzone zostały w adnotację *Transactional*, implementującą reguły ACID w ramach środowiska Spring Boot.

Z racji drugorzędnej istoty wydajności systemu na etapie prototypu, nie założono potrzeby indeksowania poszczególnych kolumn – jednakże w przypadku potencjalnego uruchomienia produkcyjnego jest to niezbędne.



Rysunek 18. Diagram architektury serwera Spring Boot

5.1.2 Warstwa logiki biznesowej

Logika biznesowa aplikacji implementowana została przez klasy oznaczone adnotacją `@service`, które następnie stają się zależnościami wstrzykiwanymi do kontrolerów lub pozostałych serwisów, co ułatwia testowanie i podmiany implementacji. Rozkładzie serwisów przyświecała zasada pojedynczej odpowiedzialności, co pozwala na uniknięcie zbędnych zależności między poszczególnymi komponentami. W ramach serwisów wydzielono:

- ***App Service*** – odpowiedzialny za zarządzanie cyklem życia aplikacji klienckiej,
- ***Feature Flag Service*** – realizujący całość logiki powiązanej z tworzeniem i zarządzaniem flagami funkcji,
- ***ABTestService*** – separujący mechanizm tworzenia eksperymentów oraz przydzielania ich do poszczególnych urządzeń użytkowników,
- ***Client Comms Service*** – łączący pozostałe funkcjonalności w akcje uruchamiane bezpośrednio przez aplikacje klienckie,
- ***OtpGeneration, PlayIntegrity, Attestation Service*** – wszystkie będące odpowiedzialne za implementację poszczególnych etapów mechanizmów bezpieczeństwa komunikacji z klientem.

Jak można zaobserwować z powyższego opisu, pomimo ogólnego założenia zasady pojedynczej odpowiedzialności wobec serwisów, poczyniono jednocześnie pewne uproszczenia – na przykład, serwis odpowiedzialny za komunikację z klientem enkapsuluje w sobie odpowiedzialności wykorzystujące zdolności kilku pozostałych serwisów. Podobne uproszczenia wprowadzono również w niektórych procedurach *FeatureFlagService* oraz *ABTestService*, rozszerzając zależność tych serwisów na więcej niż jedno repozytorium. Niniejsze podejście z perspektywy systemu produkcyjnego jest suboptymalne – ogranicza zdolności sprawnej rozbudowy systemu. Jednocześnie, jest ono świadomym kompromisem, przyjętym na etapie prototypu – gdy struktura zależności między serwisami i repozytoriami nie jest jeszcze zbyt rozbudowana. Proponowana docelowa struktura klas w przypadku dalszego rozwoju projektu zaprezentowana została w rozdziale 6.

5.1.3 Warstwa kontrolerów

Klasy kontrolerów określone przez Rysunek 18 charakteryzują się najmniejszym stopniem fragmentacji, co wynika z ich wyjątkowo prostej funkcji narzuconej przez pryncypia środowiska Spring Boot. Ich podział sterowany był dwoma czynnikami:

- Wygodą integracji panelu administratora,
- Wygodą integracji aplikacji klienckich.

W przypadku definicji końcówek REST, elementy odpowiedzialne za dostarczanie danych do frontendu zostały zorganizowane w ścieżkach inspirowanych podejściem RESTful. W rzeczywistości, projekt API podąża za większością z zasad tego podejścia: ścieżki są warstwowe, samo-opisujące się, a akcje użytkownika pozostają bezstanowe. Niektóre z rekomendacji, takie jak HATEOAS, zostały pominięte ze względu na ich znaczny narzut implementacyjny na wczesnych etapach wytwarzania systemu – i jednocześnie nie wymagany do prawidłowego działania prototypu. Przykładowe ścieżki implementowane przez kontrolery to:

Listing 1. Struktura ścieżek API panelu administracyjnego

```
/api/app/{appId}/feature-flag/{featureFlagId}  
/api/app/{appId}/ab-tests/{abTestId}
```

Lub w przypadku komunikacji z klientem w wariantcie strumieniowym – gdzie ścieżka jest uproszczona:

Listing 2. Struktura ścieżki końcówki strumieniującej

```
/api/client/{platform}/{nativeAppId}/feature-flags/stream
```

Całość komunikacji opakowana jest w obiekty *ResponseEntity*, pozwalające na sprawną obsługę odpowiedzi w pełni zgodnych z protokołem HTTP. Dodatkowo, kontrolery nie opierają się na obiektach danych czerpanych bezpośrednio z modelu. W każdym przypadku, wykorzystywane są dedykowane obiekty transferu danych, dostosowane do potrzeb przypadków użycia realizowanych przez konkretne kontrolery. Listing 3 pokazuje przykładowe DTO o kompozytowej naturze – łączące jednocześnie informacje o aplikacji wraz z informacjami o flagach funkcji, które do nich przynależą:

Listing 3. Przykład agregatu DTO

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class AppWithFeatureFlagsDto {
    private Long appId;
    private String appName;
    private List<FeatureFlagDto> featureFlags;
}
```

Listing 4. Przykład bazowego DTO

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class FeatureFlagDto {
    private Long id;
    private Long appId;

    @NotBlank(message = "Feature flag key is required")
    @Size(min = 1, max = 255)
    private String flagKey;

    @Size(min = 1, max = 255)
    private String description;
    private boolean enabled;

    @NotNull(message = "Feature flag value type is required")
    private FeatureFlag.ValueType valueType;

    @NotBlank(message = "Feature flag key is required")
    @Size(min = 1, max = 255)
    private String flagValue;

    @NotNull(message = "At least one platform is required")
    private Set<Platform> platforms;

    public static FeatureFlagDto fromEntity(FeatureFlag featureFlag) {
        FeatureFlagDto dto = new FeatureFlagDto();
        dto.setId(featureFlag.getId());
        dto.setFlagKey(featureFlag.getFlagKey());
        dto.setFlagValue(featureFlag.getFlagValue());
        dto.setEnabled(featureFlag.isEnabled());
        dto.setDescription(featureFlag.getDescription());
        return dto;
    }
}
```

5.2. Panel administratora

Komponentem pozostającym blisko centrum całego rozwiązania – i odpowiadającym większości funkcjonalności implementowanym przez serwer – jest graficzny panel administratora. Został on wykonany zgodnie ze strukturą popularnych systemów CMS: składa się z segmentów ustawień, operujących na uprzednio ustawionym kontekście użytkownika – administratora. Choć biblioteka React pozwala na duży stopień swobody w podziale kodu – jego początkowa struktura została narzucona przez bibliotekę komponentów TailAdmin, która pozwoliła na szybkie wytwarzanie nowych interfejsów graficznych, zachowując pełną spójność wyglądu aplikacji oraz jej responsywności. Aplikacja React została podzielona na następujące moduły:

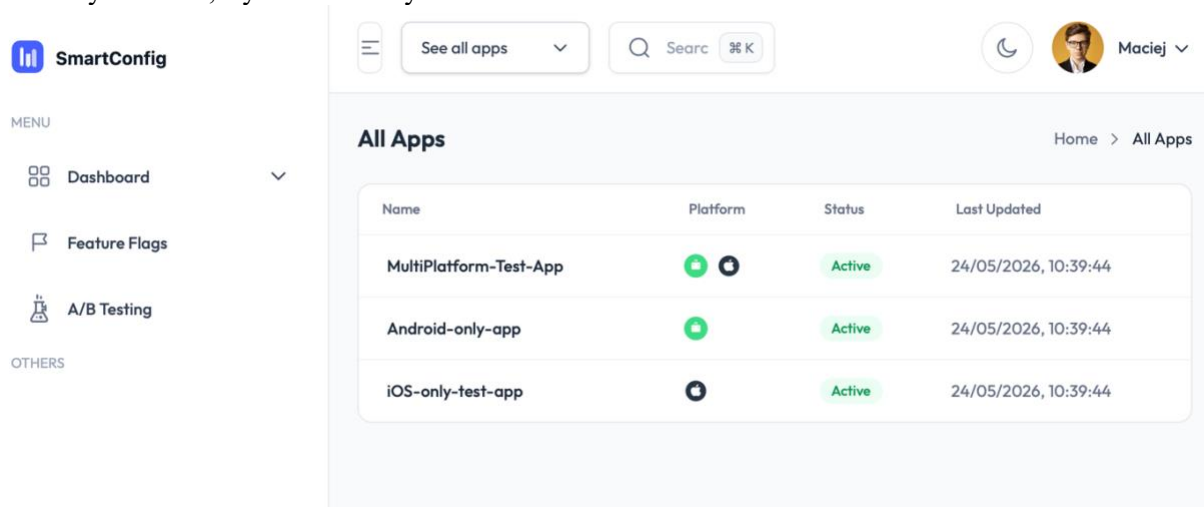
- **Components** – przechowujący reużywalne elementy interfejsu graficznego, ograniczony jedynie do warstwy prezentacji,
- **Context** – odpowiedzialny za przechowywanie stanu aplikacji, ułatwiający dodawanie funkcjonalności zachowując paradygmaty *Singe Page Application* – wykorzystano go m.in. do zarządzania informacją o aktualnie wybranej aplikacji, która używana jest przez poszczególne widoki z pakietu *Pages*,
- **Hooks** – zawierający definicję prostych, reaktywnych akcji, wykorzystywane wielokrotnie w różnych miejscach interfejsu, m.in. do obsługi nawigacji lub zarządzania cyklem życia okien dialogowych,
- **Layout** – definiujący komponenty wyznaczające ramy architektoniczne i wizualne aplikacji – na przykład, stale widoczny komponent nagłówek, menu bocznego lub tła, które opakowują buforowane widoki,
- **Pages** – zawierający definicję pod-stron, odpowiadających ścieżkom implementowanym przez serwer. Łączy ze sobą definicje wizualne oraz realizację logiki poszczególnych stron, orkiestrując wykonaniem logiki biznesowej – pobieraniem danych i przekazywaniem ich do komponentów warstwy prezentacji,
- **Services** – odpowiedzialny za reużywalne serwisy komunikacji sieciowej, łączące logikę React z zadaniami http przesyłanymi na serwer. Serwisy zdefiniowane tam odpowiadają również za podstawową walidację danych, implementując lustrzane odbicie encji DTO serwera.

Ogólna architektura i schemat komunikacji między poszczególnymi komponentami i pakietami, został przedstawiony przez Rysunek 24 – część elementów została na nim pominięta mając na uwadze czytelność i zachowanie wysokiego poziomu abstrakcji. Kluczowym elementem przytoczonego rysunku są relacje między poszczególnymi komponentami, ilustrujące w praktyce niektóre wzorce projektowe które zostały zastosowane przy wykonaniu portalu administracyjnego. Wzorzec Separacji Zagadnień podyktował wydzielenie modułu *Services*, implementacji komunikacji bezpośrednio w komponentach *Pages*. Rozdział na kontenery (*Pages*) i komponenty prezentacyjne (*Components*) zapewnił wysoką reużywalność i czytelność kodu źródłowego – jednocześnie implementując zasadę DRY. Tym samym, rozbudowana biblioteka prostych, bezstanowych komponentów pozwoliła na tworzenie interfejsów w modelu kompozycji, realizując jeden z kluczowych paradygmatów React – „kompozycji ponad dziedziczenie”.

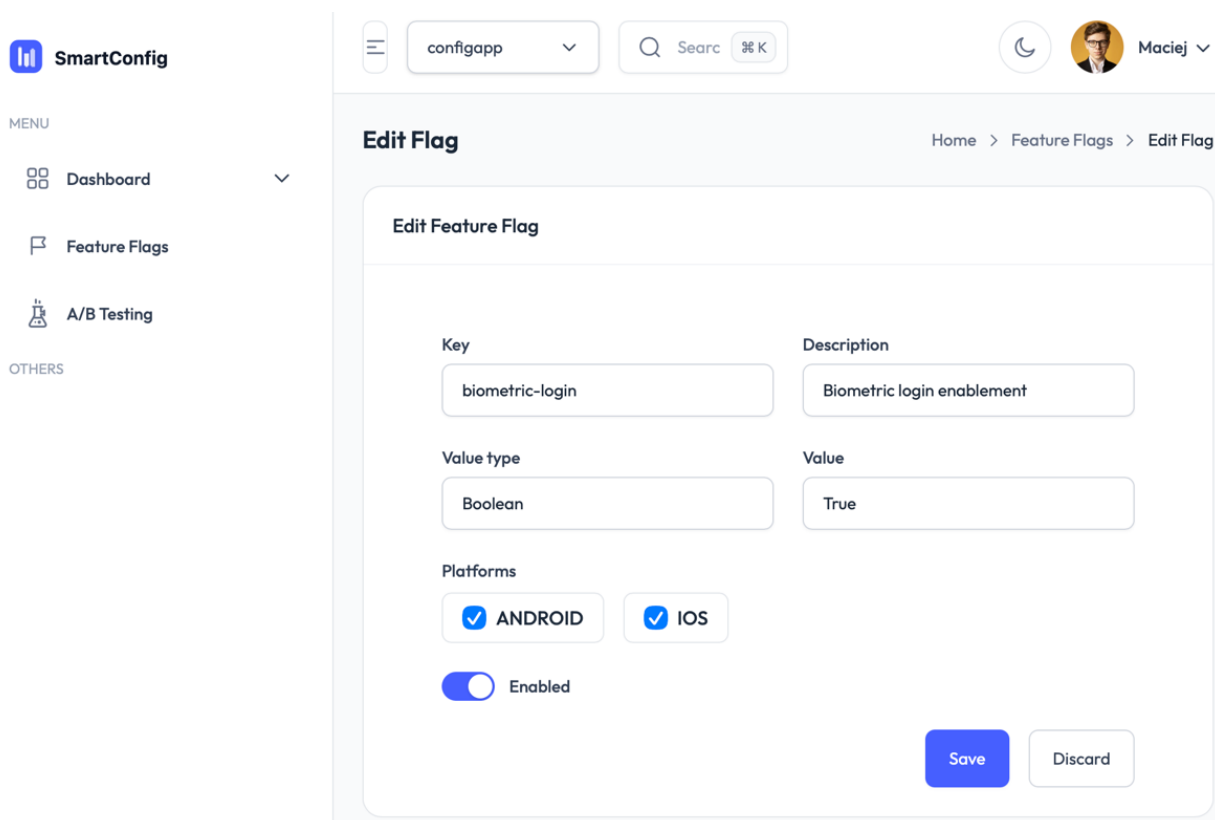
Istotne z architektonicznego punktu widzenia jest również rozpatrzenie wzorca Separacji Zagadnień w ujęciu nie tylko technicznym (tzw. horyzontalnym), który miał zastosowanie przy widokach i serwisach – ale również w ujęciu biznesowym, to jest wertykalnym: komponenty odpowiedzialne za realizację poszczególnych przypadków użycia są od siebie w pełni odseparowane, opierając się miejscami tylko na kompozycji prostszych, generycznych elementów. Dzięki temu, zachowano separację przypadków użycia podobną do tej z implementacji serwera – komponenty nazwane jako:

- **App** – odpowiadają tylko za zarządzanie aktywnymi aplikacjami klienckimi – Rysunek 19,

- **FeatureFlag** – realizują tylko przypadki użycia powiązane z zarządzaniem flagami funkcji – Rysunek 20,
- **ABTest** – używane są tylko do zarządzania eksperymentami oraz wyświetlania ich wyników – Rysunek 21, Rysunek 22 i Rysunek 23.



Rysunek 19. Widok zarządzania aplikacjami w panelu administratora



Rysunek 20. Widok manualnego tworzenia flagi funkcji

MENU

Dashboard

Feature Flags

A/B Testing

OTHERS

ConfigApp

Search

Maciej

Create A/B Test

Name

App Name Experiment

Description

Test influence of app name on users

Start Date

2026-05-19

End Date

2026-05-30

Variants

Variant Name

Control

Feature Flags

name x

Weight (%)

50

name override

Gakko

Variant Name

Experiment

Feature Flags

name x

Weight (%)

50

name override

Gekko

Add Variant

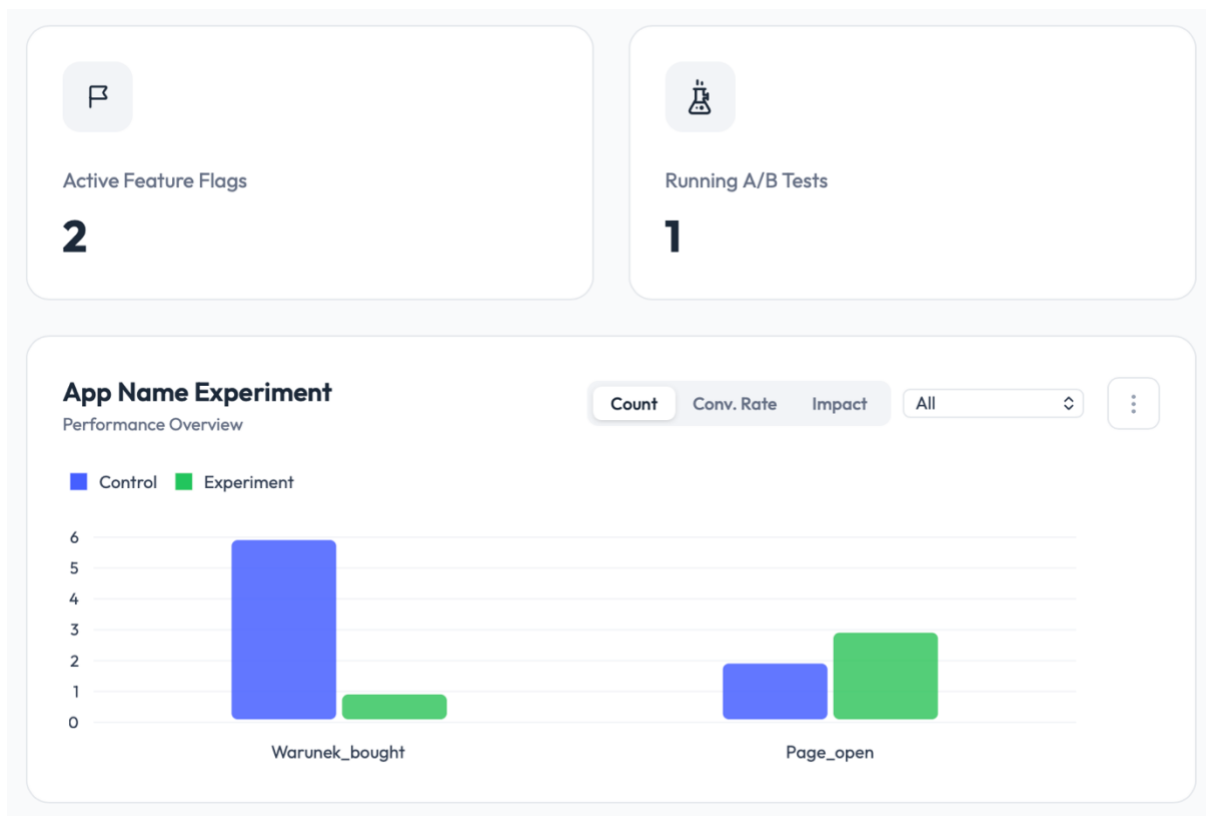
Total weight: 100%

Save

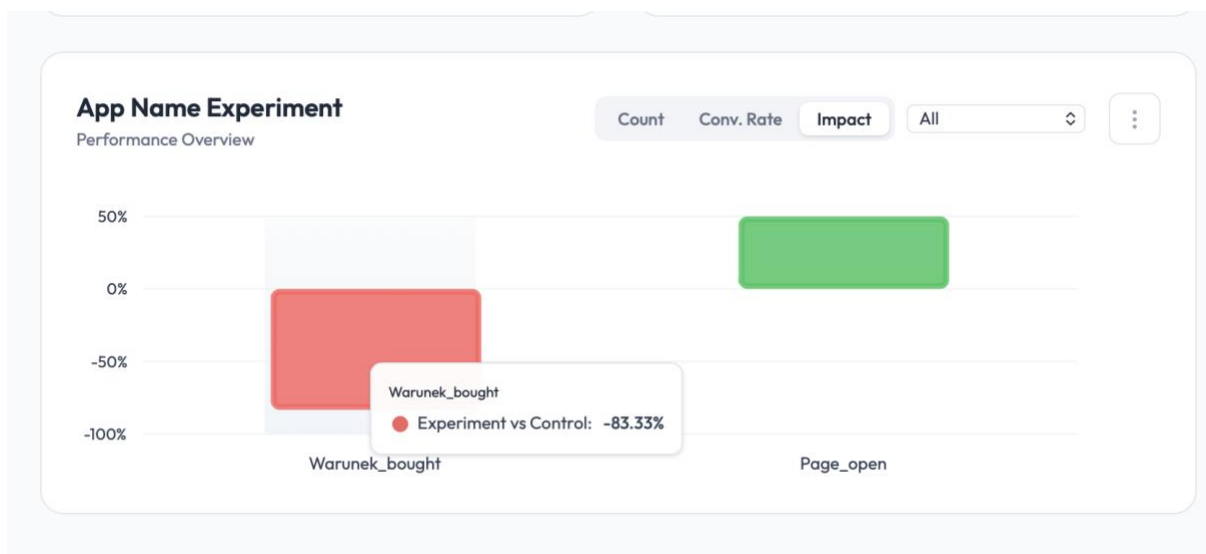
Discard

Rysunek 21. Widok tworzenia nowego testu AB

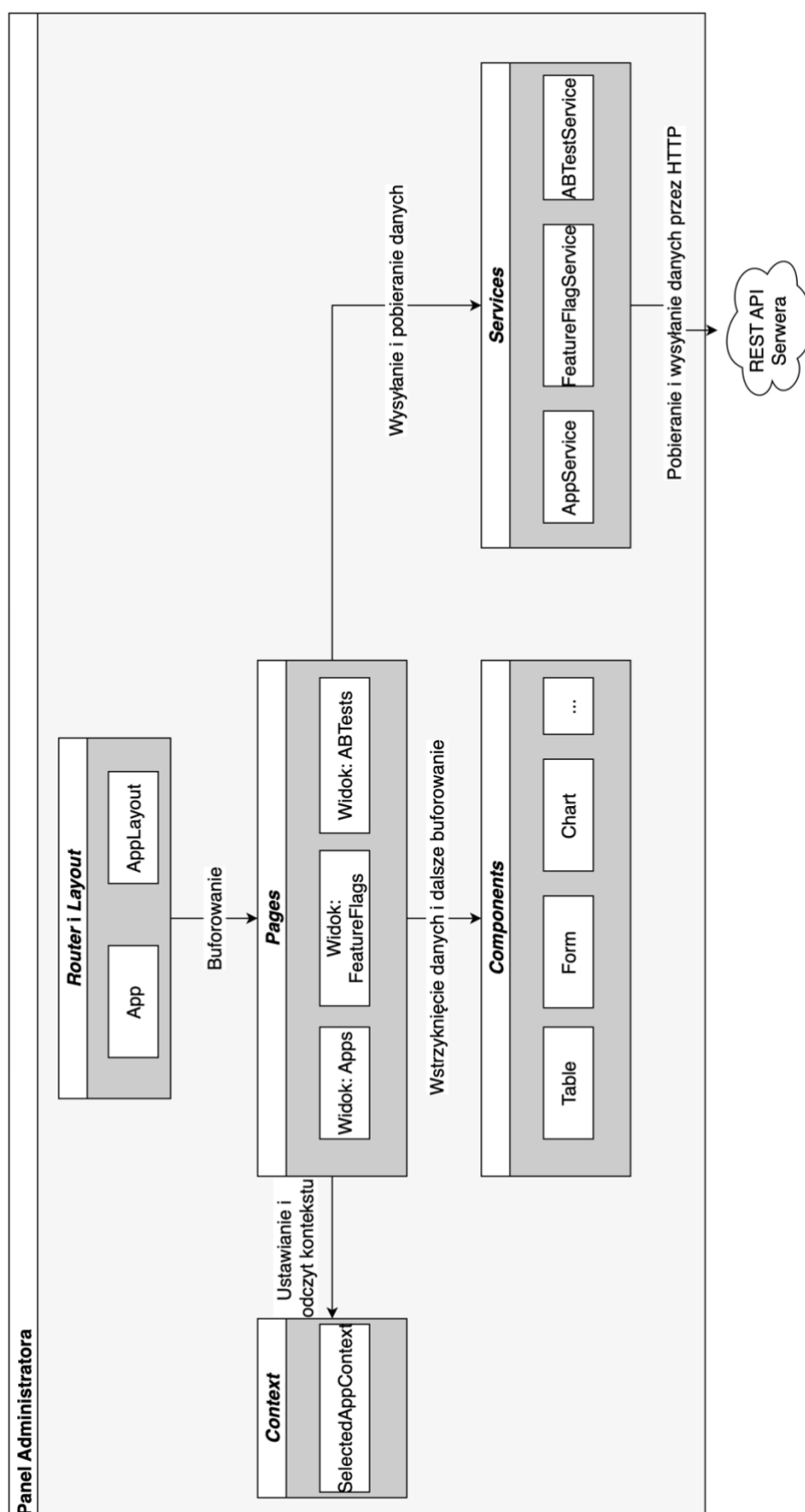
54



Rysunek 22. Widok podsumowania testu AB na podstawie liczby zarejestrowanych zdarzeń



Rysunek 23. Widok podsumowania testu AB na podstawie wpływu eksperymentu na licznosc zdarzenia



Rysunek 24. Wysokopoziomowy diagram architektury panelu administratora

5.3. Integracja na platformie iOS

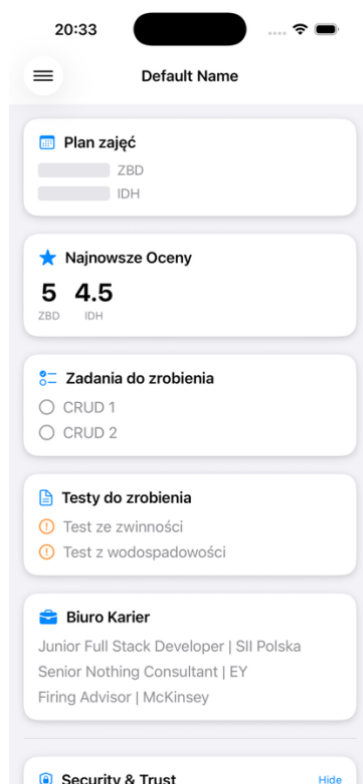
Aplikacja kliencka wraz z biblioteką integrującą na platformie mobilnej była ostatnim elementem do urzeczywistnienia pełnego ekosystemu zdalnej konfiguracji. Aplikacja iOS wraz z biblioteką została zaprojektowana zgodnie z paradygmatem czystej architektury – separując logikę aplikacji od logiki biblioteki zdalnej konfiguracji w jak największym stopniu.

5.3.1 Aplikacja testowa

Moduł pełniący rolę aplikacji testowej zrealizowany został w modelu „cienkiego klienta” – zawiera on jedynie warstwę prezentacji, pozbawioną jakiejkolwiek logiki biznesowej. Jest to minimalny zakres aplikacji pozwalający na przetestowanie założonych przypadków użycia – stan warstwy prezentacyjnej jest zależny od parametrów konfiguracyjnych dostarczanych przez bibliotekę łączącą się z serwerem.

Aplikacja składa się z jedynie dwóch zintegrowanych widoków, skomponowanych przy pomocy natywnych komponentów SwiftUI – *ContentView* oraz *MenuView*. Oba widoki jako przykład, ilustrują elementy interfejsu inspirowane aplikacjami realizującymi zadania wirtualnego dziekanatu – w tym konkretnym przypadku główną inspiracją była platforma Gakko Polsko-Japońskiej Akademii Technik Komputerowych – realizacja przedstawiona została przez Rysunek 25.

Przy technicznej realizacji interfejsu i wiązania go z podawanymi zdalnie parametrami konfiguracyjnymi, zastosowano paradygmat programowania reaktywnego – poszczególne elementy (np. banery) subskrybują konkretne instancje flag konfiguracji. Gdy zachodzi zmiana danej wartości na serwerze, interfejs odświeża się automatycznie, bez potrzeby imperatywnego przerysowania widoków.



Rysunek 25. Interfejs graficzny aplikacji testowej iOS

5.3.2 Implementacja dedykowanego SDK

Dedykowana biblioteka integrująca aplikację testową z serwerem konfiguracyjnym została enkapsulowana w wydzielonym, wielowarstwowym module SmartConfig. Skupiając się tylko na procesach powiązanych z bezpiecznym pozyskaniem i szybkim propagowaniem konfiguracji, jego architektura opiera się na trzech komponentach:

- **Rejstr** (Fasada i buforowanie) – stanowi on punkt styku między samą aplikacją kliencką a mechanizmami biblioteki SmartConfig. Aby zachować pełną spójność danych w ramach aplikacji, komponent zbudowany jest na zasadzie wzorca Singleton: przechowuje stan wszystkich zarejestrowanych i pobranych flag w jednym miejscu, jednocześnie zarządzając ich cyklem życia z perspektywy klienta. Umożliwia on auto-propagację flag w stanie *draft* do serwera, gdy są one odpowiednio oznaczone w kodzie. Same domyślne oraz pobrane z serwera wartości flag przechowywane są w pamięci operacyjnej, umożliwiając szybki dostęp oraz zabezpieczenie na wypadek braku połączenia sieciowego. Jednocześnie, interfejs pilnuje spójności obiektu flagi (inicjalizowany przez Listing 5), zawierając definicje przekazywanych do serwera DTO.

Listing 5. Definicja flagi funkcji w kodzie Swift

```
@FeatureFlag("name", default: "Default Name") var name: String
```

- **Warstwa komunikacji** (natychmiastowa propagacja zdarzeń) – element architektury podyktowany wymogiem obsługi funkcjonalności wyłączników awaryjnych. Poprzez wykorzystanie połączenia *Server-Sent Events*, inicjalizuje on długotrwałą sesję nasłuchującą na zdarzenia przychodzące z serwera. Otrzymane dane wstrzykiwane są do Registry, efektywnie i szybko aktualizując stan aplikacji.
- **Warstwa atestacji sprzętowej** – najbardziej nietypowy, a zarazem krytyczny aspekt projektu stanowi warstwa bezpieczeństwa oparta na sprzętowej weryfikacji tożsamości i integralności klienta. Implementacja tego mechanizmu znajduje się w wydzielonym pakiecie *Security*. Realizacja logiki bezpieczeństwa została rozbita na dwa tory – *DeviceCheckManager*, odpowiedzialny za komunikację w ramach urządzenia z bezpieczną enklawą oraz *DeviceCheckService*, odpowiedzialny za obsługę żądań sieciowych potrzebnych do realizacji atestowanego uścisku dłoni między smartfonem a serwerem – cała procedura uwidoczniła jest przez Listing 6.

Listing 6. Procedura tworzenia obiektów atestacji po stronie kodu iOS

```
func validateAttestation(attestationObject: String,
                        keyId: String,
                        challenge: String) async throws ->
AttestationResponse {
    log("--- Starting validateAttestation ---")
    guard let url = URL(string:
"http://localhost:8080/api/client/attestation") else {
        log("Error: Invalid URL")
        throw URLError(.badURL)
    }

    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("application/json", forHTTPHeaderField: "Content-
Type")
}
```

```

let payload = [
    "challenge": challenge,
    "keyId": keyId,
    "attestationObject": attestationObject
]

let requestData = try JSONEncoder().encode(payload)
request.httpBody = requestData

// Attestation objects can be large, so we might truncate the log or
print it all.
// Printing abbreviated version for readability, full version for
debugging.
if let requestString = String(data: requestData, encoding: .utf8) {
    log("Request [\(url.absoluteString)] Body: \(requestString)")
}

do {
    let (data, response) = try await URLSession.shared.data(for:
request)

    if let httpResponse = response as? HTTPURLResponse {
        log("Response Status Code: \(httpResponse.statusCode)")
    }

    if let responseString = String(data: data, encoding: .utf8) {
        log("Response Body: \(responseString)")
    }

    guard let httpResponse = response as? HTTPURLResponse, 200..<300
~= httpResponse.statusCode else {
        log("Error: Bad Server Response")
        throw URLError(.badServerResponse)
    }

    log("validateAttestation completed successfully")
    return try JSONDecoder().decode(AttestationResponse.self, from:
data)
} catch {
    log("Error in validateAttestation:
\(error.localizedDescription)")
    throw error
}
}

```

5.4. Integracja na platformie Android

Aby proponowane mechanizmy atestacji miały sens przy wdrożeniu produkcyjnym, muszą pokrywać w całości rynek urządzeń mobilnych – wobec tego, konieczne jest również zaimplementowanie oraz sprawdzenie ekwiwalentu rozwiązania atestacyjnego na platformie Android. W tym celu, zrealizowane zostało lustrzane odbicie aplikacji iOS, o podobnej strukturze – z minimalną warstwą prezentacyjną oraz w pełni separowaną biblioteką integracyjną, napisaną w języku Kotlin.

5.4.1 Aplikacja testowa

Przykładowy widok aplikacji testowej zrealizowany został w oparciu o deklaratywne widoki Jetpack Compose pełniące rolę pasywnego, cienkiego klienta.

Widoki są zupełnie odseparowane od logiki powiązanej z realizacją przypadków użycia systemu zdalnej konfiguracji – ich jedyna odpowiedzialność i interakcja to reaktywne przeładowywanie widoku wobec zmian wystawianych przez ViewModel połączony bezpośrednio z biblioteką SmartConfig. Mechanizm wiązania danych, automatycznie wymusza przerysowanie ekranu w momencie aktualizacji, odświeżając wygląd komponentów ilustrowanych przez Rysunek 26.



Rysunek 26. Interfejs graficzny aplikacji testowej Android

5.4.2 Implementacja dedykowanego SDK

Moduł SmartConfig w przypadku systemu Android został zrealizowany jako zbiór odizolowanych komponentów, będących lustrzanym odbiciem implementacji na platformie iOS – używając interfejsów specyficznych dla danego ekosystemu. Architektura biblioteki w przypadku Androida również opiera się na trzech warstwach:

- **Rejestru** (fasady) – wykorzystującej technologię *Coroutines* do przechowywania informacji o pobranej konfiguracji w sposób bezpieczny z perspektywy wielowątkowości systemu,
- **Komunikacji** – podobnie jak w iOS, odpowiadającej za wykonywanie żądań http oraz nawiązywania stałego połączenia z serwerem w celu natychmiastowego pobierania aktualizacji,
- **Atestacji sprzętowej** – wykorzystującej integrację z Play Integrity API, która została zencapsulowana w pakiecie *Security*. Podobnie jak w przypadku iOS, implementacja procesu atestacji została rozbita na dwa komponenty – *PlayIntegrityService*, odpowiadający za połączenia sieciowe (m.in. pobranie *nonce* z serwera) oraz *PlayIntegrityManager*, który orkiestruje całym procesem atestacji poprzez użycie natywnych metod oraz wstrzykiwanie danych atestacyjnych tam, gdzie są wymagane, którego kod zawiera Listing 7. Oba moduły są wykorzystywane w procedurze wykonywania zabezpieczonej akcji, którego kod ewidencjonuje Listing 8.

Listing 7. Klasa orkiestrująca atestacją po stronie aplikacji Android

```
object PlayIntegrityManager {

    private var integrityTokenProvider:
StandardIntegrityManager.StandardIntegrityTokenProvider? = null

    suspend fun prepareIntegrityProvider(context: Context,
cloudProjectNumber: Long) {
        val integrityManager =
IntegrityManagerFactory.createStandard(context)

        val request =
StandardIntegrityManager.PrepareIntegrityTokenRequest.builder()
            .setCloudProjectNumber(cloudProjectNumber)
            .build()

        integrityTokenProvider =
integrityManager.prepareIntegrityToken(request).await()
    }

    suspend fun getToken(nonce: String): String {
        val provider = integrityTokenProvider ?: throw
IllegalStateException("Intgrity provider not initialized")

        val request = StandardIntegrityTokenRequest.builder()
            .setRequestHash(nonce)
            .build()

        val response = provider.request(request).await()
        return response.token()
    }
}
```

Listing 8. Procedura odpowiedzialna za atestowaną komunikację z serwerem

```
suspend fun performSecureAction(): Boolean {
    return try {
        Log.d(TAG, "Step 1: Requesting challenge from backend")
        val challengeRequest = ChallengeRequestDTO(keyId =
"testdeviceid")
        val challengeResponse = apiService.getChallenge(challengeRequest)
        val nonce = challengeResponse.challenge
        Log.d(TAG, "Got nonce: $nonce")

        Log.d(TAG, "Step 2: Requesting integrity token")
        val token = PlayIntegrityManager.getToken(nonce)
        Log.d(TAG, "Got integrity token. ${token.length}")

        Log.d(TAG, "Step 3: Verifying token with backend")
        val verificationRequest = IntegrityVerificationRequest(
            integrityToken = token,
            nonce = nonce
        )

        val verifyRes = apiService.verifyToken(verificationRequest)

        if (verifyRes.success) {
            Log.d(TAG, "Integrity verification succeeded")
        }
    }
}
```

```

        true
    } else {
        Log.w(TAG, "Integrity verification failed:
${verifyRes.message}")
        false
    }
}
} catch (e: retrofit2.HttpException) {
    if (e.code() == 401) {
        val errorBody = e.response()?.errorBody()?.string()
        if (errorBody != null) {
            try {
                val gson = com.google.gson.Gson()
                val errorResponse = gson.fromJson(errorBody,
VerificationResponse::class.java)
                Log.w(TAG, "Verification failed (401):
${errorResponse.message}")
            } catch (jsonException: Exception) {
                Log.e(TAG, "HTTP 401: $errorBody")
            }
        } else {
            Log.e(TAG, "HTTP 401: Verification failed")
        }
        false
    } else {
        Log.e(TAG, "HTTP error ${e.code()}: ${e.message()}", e)
        false
    }
}
} catch (e: Exception) {
    Log.e(TAG, "Error during secure action flow", e)
    false
}
}
}

```

5.5. Walidacja rozwiązania

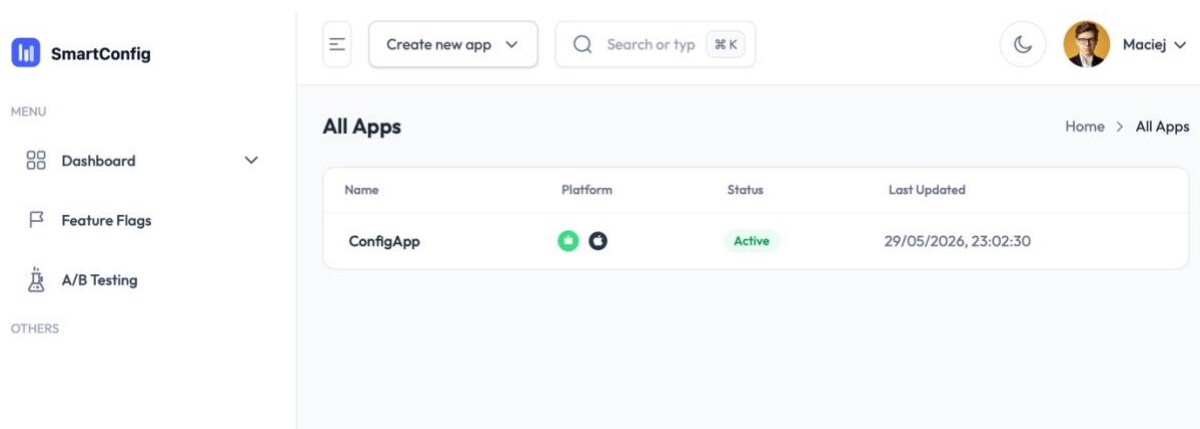
Celem zapewnienia jakości i udowodnienia prawidłowego działania rozwiązania, konieczne było przeprowadzenie testów end-to-end, angażujących webowy panel administracyjny, serwer zdalnej konfiguracji oraz aplikacje klienckie wraz z zintegrowanymi bibliotekami. Celem minimalnego udowodnienia działania conceptów, zdefiniowano trzy scenariusze testowe, dotyczące aspektu flag funkcji, mechanizmów bezpieczeństwa oraz spójności deterministycznej testów A/B.

5.5.1 Weryfikacja wyłącznika awaryjnego (SSE)

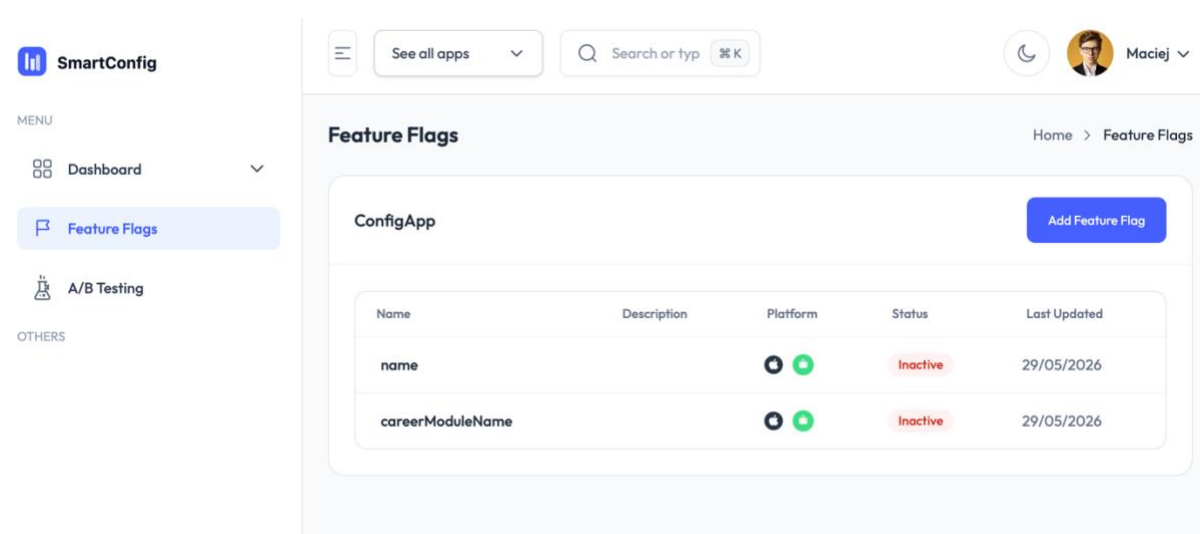
Mając na celu weryfikację danego przypadku użycia, najpierw należy zdefiniować stosowny scenariusz testowy. Przed realizacją konkretnego scenariusza, zdefiniować należy warunki początkowe testu – należą do nich:

- Zarejestrowanie aplikacji klienckiej w panelu administratora (Rysunek 27),
- Uruchomienie aplikacji klienckiej – automatyczne utworzenie flag funkcji (Rysunek 28),
- Włączenie uprzednio utworzonych flag funkcji.

Przed wykonaniem niniejszego testu, wszystkie akcje zostały wykonane, zgodnie z załączonymi rysunkami.



Rysunek 27. Zarejestrowana aplikacja testowa



Rysunek 28. Automatycznie stworzone flagi przed aktywacją

Następnie, należy wykonać zmianę wartości aktywnych flag, tak aby zostały one natychmiastowo rozpropagowane do aktualnie podłączonych klientów. Akcję aktualizacji flagi wykonano w momencie oznaczonym stemplem czasowym **23:37:37.044**. Rezultatem akcji stała się kaskada aktualizacji w aplikacjach klienckich o następujących godzinach:

23:37:37.299 – dostarczenie informacji o aktualizacji flag przez SSE

23:37:37.371 – wysłanie żądania o pobranie nowego zestawu flag funkcji

23:37:37.385 – otrzymanie nowego zestawu flag funkcji

23:37:37.399 – zakończenie przerysowywania interfejsu, zmieniona flaga jest widoczna w aplikacji

Dokładne przetwarzanie rejestrowanych informacji zostało zilustrowane przez Listing 9. Wobec przeprowadzonego testu, udało się uzyskać czas aktualizacji flagi funkcji znacznie mniejszy niż pół sekundy – efektywnie, z perspektywy użytkownika, wręcz natychmiastowy. Należy wziąć pod uwagę, że w przypadku testu sesja między aplikacją kliencką a serwerem była już nawiązana, a sama komunikacja odbywała się w ramach tej samej sieci lokalnej, co znacząco redukuje narzuty wydajnościowe.

Listing 9. Zrzut logów z aplikacji klienckiej

```
2026-05-29 23:37:37.299 21574-21660 FeatureFlagManager dev.maciejniszczota.configapp
D SSE event received - type: feature-flags-updated, data: refresh

2026-05-29 23:37:37.300 21574-21574 HomeViewModel dev.maciejniszczota.configapp
D Feature flags update detected, refreshing...

2026-05-29 23:37:37.371 21574-21738 SmartConfig dev.maciejniszczota.configapp
D Fetching from:
http://10.0.2.2:8080/api/client/android/dev.maciejniszczota.configapp/feature-flags

2026-05-29 23:37:37.384 21574-21738 SmartConfig dev.maciejniszczota.configapp
D Response code: 200

2026-05-29 23:37:37.385 21574-21738 SmartConfig dev.maciejniszczota.configapp
D Response body: {"careerModuleName":"Biuro Karier","name":"Gakko-test"}

2026-05-29 23:37:37.385 21574-21738 SmartConfig dev.maciejniszczota.configapp
D Parsed flag: careerModuleName = Biuro Karier

2026-05-29 23:37:37.385 21574-21738 SmartConfig dev.maciejniszczota.configapp
D Parsed flag: name = Gakko-test

2026-05-29 23:37:37.399 21574-21738 SmartConfig dev.maciejniszczota.configapp
D Updated flags: SnapshotStateMap(value={name=Gakko-test, careerModuleName=Biuro
Karier})@92576333
```

5.5.2 Walidacja mechanizmów bezpieczeństwa

Celem weryfikacji ochrony oferowanej przez mechanizmy atestacyjne, należy przyjąć tzw. scenariusz negatywny – podjąć próbę komunikacji z serwerem poprzez urządzenie nieakceptowalne. W przypadku rygorystycznych wymogów atestacyjnych, przygotowanie takiego środowiska nie jest skomplikowane – wystarczające jest użycie symulatora, który jest traktowany jako urządzenie niebezpieczne, niezdolne do przejścia atestacji.

W obu przypadkach, zarówno Android jak i iOS, efektem są logi wskazujące na odrzucenie połączenia przez serwer – widoczne szczególnie w ostatniej linijce ilustrowanej przez Listing 10.

Listing 10. Zrzut logów z symulatora Android w momencie atestacji

```
2026-05-30 00:02:48.771 21574-21694 HomeViewModel dev.maciejniszczota.configapp
D Step 1: Requesting challenge from backend

2026-05-30 00:02:48.784 21574-21694 HomeViewModel dev.maciejniszczota.configapp
D Got nonce: 656144

2026-05-30 00:02:48.784 21574-21694 HomeViewModel dev.maciejniszczota.configapp
D Step 2: Requesting integrity token

2026-05-30 00:02:48.784 21574-21694 PlayCore dev.maciejniszczota.configapp
I UID: [10214] PID: [21574] StandardIntegrity : requestExpressIntegrityToken(-
924916230229346826)

2026-05-30 00:02:48.784 21574-21661 PlayCore dev.maciejniszczota.configapp
I UID: [10214] PID: [21574] StandardIntegrity : Initiate binding to the service.

2026-05-30 00:02:48.811 21574-21574 PlayCore dev.maciejniszczota.configapp
I UID: [10214] PID: [21574] StandardIntegrity :
ServiceConnectionImpl.onServiceConnected(ComponentInfo{com.android.vending/com.google.android.finsky.expressintegrityservice.ExpressIntegrityService})

2026-05-30 00:02:48.811 21574-21661 PlayCore dev.maciejniszczota.configapp
I UID: [10214] PID: [21574] StandardIntegrity : linkToDeath

2026-05-30 00:02:48.828 21574-21614 PlayCore dev.maciejniszczota.configapp
I UID: [10214] PID: [21574] OnRequestIntegrityTokenCallback :
onRequestExpressIntegrityToken
```

```

2026-05-30 00:02:48.828 21574-21661 PlayCore dev.maciejniszczota.configapp
I UID: [10214] PID: [21574] StandardIntegrity : Unbind from service.
2026-05-30 00:02:48.828 21574-21694 HomeViewModel dev.maciejniszczota.configapp
D Got integrity token. 404
2026-05-30 00:02:48.828 21574-21694 HomeViewModel dev.maciejniszczota.configapp
D Step 3: Verifying token with backend
2026-05-30 00:02:49.287 21574-21694 HomeViewModel dev.maciejniszczota.configapp
W Verification failed (401): Integrity verification failed

```

Dokładnie ten sam efekt, choć przykryty logami o wyższym poziomie abstrakcji otrzymywany jest w przypadku iOS (Listing 11).

Listing 11. Zrzut logów z symulatora iOS w momencie przeprowadzania atestacji

```

[DeviceCheck] Initializing Security Flow...
[DeviceCheck] Registering Device (KeyGen)...
[DeviceCheck] ERROR: The operation couldn't be completed. (configapp_ios.DeviceCheckError
error 3.)

```

5.5.3 Test spójności deterministycznej konfiguracji

Celem testu jest weryfikacja, czy ten sam użytkownik (o tym samym ID urządzenia) zawsze otrzyma tę samą konfigurację, realizując techniczną część przypadku użycia testów A/B. Aby zrealizować ten test, konieczne jest spełnienie warunków:

- Aplikacja kliencka zarejestrowana w panelu administracyjnym
- Flagi funkcji zarejestrowane w panelu administracyjnym

Następnie, należy przygotować test A/B, który dostarczy obejścia wartości na skonfigurowanych flagach funkcji – w tym celu nadpisano nazwę aplikacji – w grupie kontrolnej przybiera ona wartość „Gakko”, natomiast w grupie eksperymentalnej „Gago”. W momencie, gdy test jest stworzony, przystąpić można do właściwego testu. Test zostanie wykonany za pomocą narzędzia Postman, które pozwoli na sprawne nadpisanie parametru identyfikatora urządzenia. Jako że test odbywa się zupełnie poza ekosystemem aplikacji mobilnych, konieczne jest również przełączenie serwera w tryb deweloperski – aby akceptował połączenia bez atestacji. Następnie, wysyłane będą kolejne żądania o zdalną konfigurację, z delikatnie zmienionym identyfikatorem, tak aby wymusić na logice przydzielającej wariant pewną entropię. Dla każdego identyfikatora test zostanie powtórzony trzy razy, aby upewnić się co do konsekwentności przydziału. Wyniki plasują się następująco:

- Identyfikator 1234: Gago
- Identyfikator 12345: Gakko
- Identyfikator 123456: Gakko
- Identyfikator 1234567: Gago

Przytoczone wyniki są jedynie przyjazną czytelnikowi ilustracją konceptu przydziału wariantów – rzeczywisty test zostanie wykonany za pomocą skryptu Python, który dokona generacji 1000 losowych identyfikatorów oraz wykona zapytania do API, zbierając statystyki, jak plasuje się podział między grupą eksperymentalną a kontrolną. Zgodnie z założeniami testu A/B, podział powinien dążyć do 50% dla każdego wariantu. Wyniki po 1000 iteracji są następujące:

- **Grupa eksperymentalna** – 529 wyników, 52,9%,

- **Grupa kontrolna** – 471 wyników, 47,1%.

Wyniki te wskazują na to, iż algorytm przydziału wariantów w większej skali sprawdza się zgodnie z oczekiwaniami. Ewentualne odchylenia od oczekiwanej wartości 50% wynikają z losowości skryptu wobec potencjalnych wyników funkcji skrótu odpowiedzialnej za przydział.

6. Problemy i wyzwania związane z realizowanym systemem

System, którego projekt i implementacja zostały opisane w poprzedzających rozdziałach udawania wykonalność integracji mechanizmów atestacji sprzętowej w środowiskach wymagających uwierzytelnienia domyślnie niezaufanych, mobilnych klientów wobec kluczowych komponentów infrastruktury. Jednakże, pomimo osiągnięcia głównych założeń badawczych, realia systemów produkcyjnych systemów zdolnych do uruchomienia w ramach przedsiębiorstwa stawiają wobec proponowanego systemu dodatkowe, rygorystyczne ograniczenia w wymiarze technicznym jak i biznesowym.

Niniejszy rozdział stanowi krytyczną analizę zbudowanego prototypu – skupiając się na przyjętych kompromisach architektonicznych, uproszczeniach domenowych i potencjalnych przyszłych problemach – wskazując wektory problemów, które powinny zostać rozwiązane zanim system uzyskałby zdolność do działania w środowisku produkcyjnym.

6.1. Ograniczenia komunikacji w tle a funkcjonalność natychmiastowej propagacji zmian

Jedną z kluczowych propozycji systemu było zastosowanie mechanizmu *Server-Sent Events* do dystrybucji flag funkcji o minimalnej latencji. Z technicznego punktu widzenia, rozwiązanie to funkcjonuje bezbłędnie w warunkach testowych, gdy aplikacja kliencka niemal zawsze znajduje się na pierwszym planie interfejsu urządzenia mobilnego. W scenariuszu rzeczywistego użytkowania, należy wziąć pod uwagę, że przez większość czasu będzie wręcz odwrotnie – wtedy, jednym z wyzwań architektonicznych stanie się podtrzymanie połączenia sieciowego między aplikacją a serwerem zdalnej konfiguracji.

Mobilne systemy operacyjne takie jak Android lub iOS, implementują zaawansowane mechanizmy oszczędzania energii, powodując redukcję zasobów dostępnych do użytku przez aplikacje uruchomione w tle. W przypadku sesji SSE, należy założyć, że system operacyjny z biegiem czasu wymusi zamknięcie gniazd sieciowych, efektywnie zrywając połączenie gdy tylko użytkownik zminimalizuje aplikację lub zablokuje ekran urządzenia.

Wobec tego, należy zachować w pamięci, iż system SSE będzie funkcjonował prawidłowo tylko dla aplikacji o aktywnej sesji, jednocześnie upewniając się, że każde wybudzenie aplikacji wymusi ponowne nawiązanie połączenia. W przypadku, gdyby wdrożenie zmian konfiguracyjnych było niezbędne nawet w momencie uśpienia aplikacji, rekomendowanym podejściem jest integracja systemu z natywnymi usługami powiadomień *push*. Wykorzystując możliwość wysyłania przez serwer niewidzialnego dla użytkownika, cichego powiadomienia, pozwala to na krótkotrwale wybudzenie aplikacji i natychmiastowe pobranie zaktualizowanego stanu konfiguracji z serwera – zanim użytkownik wróci do aplikacji.

6.2. Aspekty prywatności i regulacje prawne w testach A/B

Aby mechanika eksperymentów A/B była realizowana w prawidłowy sposób, konieczne jest zapewnienie determinizmu w przydziale użytkownika – tak, aby ten sam użytkownik raz przydzielony do danego wariantu testowego pozostał w nim do końca kampanii ewaluacyjnej. Istota tej kwestii

dyktowana jest nie tylko wartością analityczną zbieranych danych, ale również spójnością doświadczeń użytkownika. W przypadku prototypu, realizacja przydziału wariantu została oparta o unikalny identyfikator urządzenia, przesyłany w momencie przydzielenia wariantu A/B.

Z perspektywy wdrożenia takiego systemu w warunkach komercyjnych, podejście to może rodzić wyzwania w kontekście regulacji prawnych dotyczących ochrony danych osobowych, regulowanych na terenie Unii Europejskiej przez przepisy RODO, Wykorzystanie unikalnych i niezmiennych identyfikatorów sprzętowych do śledzenia zachowań użytkowników i przydzielania im pewnych profili behawioralnych wymaga uprzedniej, wyraźnej zgody.

Docelowa implementacja w najlepszym przypadku powinna całkowicie zrezygnować z przesyłania jawnych identyfikatorów sprzętowych do serwera dystrybucyjnego i zastąpić je surogatami dowolnego pochodzenia – mogą to być cyklicznie rotowane anonimowe identyfikatory instalacji, lub rozważenie przeniesienia ewaluacji przydzielonych wariantów na stronę klienta, gdzie serwer dostarcza jedynie wagi, a wydarzenia analityczne zawierają jedynie identyfikatory użytkowanego wariantu, pozwalając na zachowanie pełnej anonimowości.

6.3. Zależność wobec zewnętrznych usług

Implementacja architektury bezpieczeństwa oparte o usługi atestacyjne Apple oraz Google niewątpliwie podnosi poziom bezpieczeństwa systemu. Jednakże, wprowadza również ze sobą pewne zagrożenia w pozostałych domenach. Użycie centralnych systemów, wprowadza pojedynczy punkt awarii w przypadku, jeśli jest to główne zabezpieczenie systemu. Do potencjalnych problemów może dojść w następujących scenariuszach:

- Awaria serwerów atestacyjnych
- Niepoprawnie przeprowadzona rotacja certyfikatów
- Wycofanie wsparcia dla starszych wersji urządzeń klienckich

W tym scenariuszu, każde żądanie aplikacji klienckiej zostałoby odrzucone wobec braku pomyślnej weryfikacji. Podobne efekty może mieć wprowadzenie aplikacji na rynki, które ograniczają dostępność wymienionych zależności ze względów regulacyjnych: przykładem jest blokada usług Google Play na terenie Chińskiej Republiki Ludowej lub w przypadku urządzeń objętych amerykańskim embargiem technologicznym (np. smartfony Huawei). Brak mechanizmu alternatywnego automatycznie pozbawia potencjalnych użytkowników od możliwości korzystania z aplikacji.

Wobec tego, konieczne jest rozważenie dopuszczenia planowanej, bezpiecznej degradacji usługi w określonych przypadkach. Tam, gdzie nie jest to absolutnie niezbędne, możliwe staje się częściowe obniżenie wymagań bezpieczeństwa lub przełączenie się na inne metody autoryzacji – potencjalnie wspierane przez heurystyczne metody analizy ruchu na linii klient-serwer.

6.4. Refaktoryzacja kodu serwera

Obecna implementacja warstwy serwerowej, choć z powodzeniem realizuje postawione przed nią cele badawcze, wymaga znaczącej transformacji przed potencjalnym wdrożeniem tego rozwiązania do komercyjnego systemu o dużej skali i dostępności.

6.4.1 Optymalizacja warstwy transportowej

Aktualna implementacja mechanizmu *Server-Sent Events* opiera się na domyślnym serwerze aplikacyjnym Apache Tomcat i tradycyjnym modelu przydzielającym jeden wątek na żądanie – co stałoby się znaczącym wyzwaniem wydajnościowym wobec znaczącej ilości jednoczesnych użytkowników. Optymalizacja tego elementu systemu wymagałaby adopcji mechanizmu wirtualnych wątków, które pojawiły się w ramach edycji 21 języka Java. Dzięki innemu podejściu do zarządzania cyklem życia wątków, serwer uzyska zdolność usypiania i wybudzania lekkich wątków – otwierając drogę do obsługi tysięcy połączeń bez znaczącego narzutu wydajnościowego.

6.4.2 Implementacja szyny zdarzeń (*event bus*)

W ramach prototypu, logika subskrypcji zdarzeń opiera się o obiekty istniejące w ramach czasu wykonania poszczególnej instancji aplikacji, co może przynieść znaczące problemy przy horyzontalnym skalowaniu serwera, np. w ramach środowiska Kubernetes. Aby zapewnić pełną kompatybilność implementowanej logiki biznesowej z popularnymi mechanizmami skalowania, konieczne staje się zastosowanie brokera wiadomości (Redis Pub/Sub lub Apache Kafka), który będzie pełnił rolę uprzednio stosowanych obiektów. Zapewniłoby to, że każda instancja aplikacji korzystałaby z tego samego źródła danych, zapewniając spójność rozgłaszania zdarzeń do klientów, niezależnie od węzła, z którym akurat są połączeni.

6.4.3 Dekompozycja połączeń Serwis-Repozytorium

Obecna struktura prototypu wykazuje się silnym sprzęgnięciem między serwisami biznesowymi a wieloma repozytoriami – często z innych obszarów domenowych. W przypadku dalszego utrzymania systemu, stanowi to ryzyko powstawania cyklicznych zależności oraz nieintencjonalnych błędów w zarządzaniu danymi całego systemu.

Konieczne jest rygorystyczne wydzielenie pasów integracji między serwisami a repozytoriami zgodnie z pryncypiami Domain-Driven Design. Serwisy powinny operować wyłącznie na danych, do których domenowo się odnoszą, natomiast obecne zależności powinny zostać zamienione na komunikację między poszczególnymi serwisami.

6.4.4 Eliminacja narzutu ORM

Wykorzystanie środowiska Hibernate jako warstwy mapowania obiektowo-relacyjnego znacząco przyspiesza fazę wytwórczą – jednakże w systemach o wysokiej przepustowości staje się on wąskim gardłem na drodze dostępu do bazy danych.

Docelowy kształt systemu powinien zakładać migrację auto-generowanych zapytań do natywnych kwerend SQL lub wykorzystania bibliotek takich jak jOOQ, pozwalając na wysoce zoptymalizowaną obsługę najczęstszych zapytań.

6.5. Ograniczenia procesu programistycznego i zapewnienia jakości

Zastosowanie sprzętowych mechanizmów atestacji i poziom bezpieczeństwa który wnoszą do aplikacji wpływa nie tylko na samych użytkowników – jest to również istotny aspekt dyktujący podejście do procesu wytwórczego aplikacji, wprowadzając do niego głębokie komplikacje.

Pierwszą, fundamentalną zmianą jest wykluczenie możliwości korzystania z emulatorów urządzeń mobilnych przez programistów i testerów. Jako, że wirtualne środowiska nie posiadają dostępu do fizycznych komponentów kryptograficznych, nie mogą one otrzymać pozytywnego werdyktu atestacji, co zostało dowiedzione w fazie testów rozwiązania. Z perspektywy codziennej pracy programistycznej, konieczność kompilacji kodu i testowania go wyłącznie na fizycznych urządzeniach, podłączonych kablem do maszyny programisty znacząco utrudnia proces wytwórczy. Aby zmniejszyć ten narzut, konieczne staje się zróżnicowanie zabezpieczeń między środowiskami poprzez wyłączenie mechanizmów atestacji – np. w przypadku uruchomienia w środowisku lokalnym lub deweloperskim. Podejście to, choć konieczne, narusza najlepsze praktyki zarządzania środowiskami, sprawiając, że programista nigdy nie testuje faktycznego, docelowego zachowania aplikacji.

Ograniczenia te wpływają również na koszt zapewnienia jakości aplikacji – absolutnie niezbędne jest zapewnienie testerom urządzeń fizycznych – co nieraz jest problemem operacyjnym. Alternatywą o akceptowalnym koszcie jest adopcja platform dostarczających dostęp do rzeczywistych urządzeń, na które można wgrać aplikację i przeprowadzić zestaw testów, takich jak Kobiton, TestMU lub Browserstack.

7. Podsumowanie

Głównym celem niniejszej pracy była ocena rynku i zabezpieczeń rozwiązań na nim obecnych i następne zaprojektowanie, zaimplementowanie i zweryfikowanie systemu zarządzania zdalną konfiguracją, dedykowanego urządzeniom mobilnym. Zrealizowany prototyp w pełni udowadnia wykonalność założenia wykorzystania sprzętowych metod uwierzytelnienia wraz z integracją towarzyszących funkcjonalności. Z sukcesem stworzono wielowarstwowe środowisko, składające się z serwera dystrybucyjnego, graficznego panelu administracyjnego oraz dwóch hermetycznych bibliotek na platformy iOS i Android, które zostały zintegrowane w ramach autorskich aplikacji testowych. Największym osiągnięciem badawczym jest zaproponowanie alternatywnego podejścia do użycia kluczy API w ramach aplikacji mobilnych, pozwalając na uniknięcie kompromitacji połączonych systemów drogą dekompilacji kodu źródłowego. Pozwoliło to na stworzenie bezpiecznego kanału dystrybucji konfiguracji, implementującego tzw. wyłączniki awaryjne oraz testy A/B – odpornego nie tylko na statyczną analizę kodu, ale również na potencjalne ataki typu man-in-the-middle lub nawiązywanie połączeń przez farmy botów.

Mimo pozytywnej weryfikacji systemu wobec pierwotnych założeń projektowych, zrealizowany prototyp systemu eksponuje pewne ograniczenia architektoniczne, wyznaczające możliwe kierunki dalszych prac rozwojowych. Należą do nich przede wszystkim podatność utrzymywania długotrwałych połączeń sieciowych, narzut operacyjny adopcji sugerowanego poziomu zabezpieczeń (utrzymywanie środowisk testowych) oraz kwestie wydajnościowe serwera – dotyczące głównie zasobów komunikacyjnych.

Zasadniczym wnioskiem płynącym z przeprowadzonego procesu badawczego i twórczego jest stwierdzenie, że współczesne, natywne interfejsy atestacyjne dostarczane przez twórców systemów operacyjnych urządzeń mobilnych pozwalają na bezprecedensowe zabezpieczanie infrastruktury przed nieautoryzowanym ruchem. Wniosek ten sugeruje również, że wartościowym pomysłem jest tworzenie systemów dopasowanych do platformy, w ramach której mają być używane – usługi generyczne, próbujące połączyć platformy webowe i mobilne posiadają zestaw narzędzi znacząco ograniczony przez międzyplatformowe wymogi kompatybilności. Należy mieć jednak na uwadze, że przyjęte podejście okupione jest kompromisem w postaci utrudnionego procesu twórczego i rosnącej zależności względem pojedynczych, globalnych dostawców.

Stworzony system stanowi solidny i innowacyjny fundament, ilustrujący wagę użycia natywnie dostępnych narzędzi w domenie bezpiecznego zarządzania cyklem życia aplikacji mobilnych – posiadając potencjał do modularnego reużycia komponentów we wszelkich przypadkach użycia o podobnych wymaganiach w domenie bezpieczeństwa i latencji.

Prace cytowane

- [1] G. Kim, J. Humble, P. Debois, J. Willis i N. Forsgren, DevOps. Światowej klasy zwinnosc, niezawodnosc i bezpieczenstwo w Twojej organizacji. Wydanie II., Onepress, 2023, ISBN: 978-83-283-9686-9.
- [2] J. Dremann, R. Mahdavi i L. Williams, „Software Development with Feature Toggles: Practices used by Practitioners,” *Empirical Software Engineering*, tom 26, nr 1, p. 1, 2021.
- [3] R. Fedytskyi, „Dynamic Frontend Architecture for Runtime Component Versioning and Feature Flag Resolution in Regulated Applications,” *Software*, tom 4, nr 4, p. 32, 2025.
- [4] M. Fowler, „Feature toggle,” 29 October 2010. [Online]. Available: <https://martinfowler.com/bliki/FeatureFlag.html>. [Data uzyskania dostępu: 15 March 2026].
- [5] Google LLC, „Firebase Remote Config,” 10 March 2026. [Online]. Available: <https://firebase.google.com/docs/remote-config>. [Data uzyskania dostępu: 15 March 2026].
- [6] LaunchDarkly, „Feature Management for the Modern Enterprise,” 2026. [Online]. Available: <https://launchdarkly.com/>. [Data uzyskania dostępu: 15 March 2026].
- [7] Netflix, Inc., „Archaius: Library for configuration management API,” [Online]. Available: <https://github.com/Netflix/archaius>. [Data uzyskania dostępu: 15 March 2026].
- [8] OWASP Foundation, „OWASP Mobile Application Security Verification Standard (MASVS),” [Online]. Available: <https://mas.owasp.org/MASVS/>. [Data uzyskania dostępu: 15 March 2026].
- [9] Google LLC, „Understand Firebase Projects: Publicly identifying your project,” [Online]. Available: <https://firebase.google.com/docs/projects/learn-more>. [Data uzyskania dostępu: 15 March 2026].
- [10] Amazon Web Services, Inc., „Authenticating Requests (AWS Signature Version 4),” [Online]. Available: <https://docs.aws.amazon.com/general/latest/gr/signature-version-4.html>. [Data uzyskania dostępu: 15 March 2026].
- [11] Amazon Web Services, Inc., „What is AWS AppConfig?,” [Online]. Available: <https://docs.aws.amazon.com/appconfig/latest/userguide/what-is-appconfig.html>. [Data uzyskania dostępu: 15 March 2026].
- [12] Microsoft Corporation, „Azure App Configuration documentation,” [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-app-configuration/>. [Data uzyskania dostępu: 16 March 2026].
- [13] ConfigCat, „ConfigCat Docs,” [Online]. Available: <https://configcat.com/docs/>. [Data uzyskania dostępu: 23 March 2026].
- [14] Optimizely, „Optimizely,” [Online]. Available: <https://www.optimizely.com>. [Data uzyskania dostępu: 2 April 2026].

- [15] Optimizely, „Optimizely Developer Documentation,” [Online]. Available: <https://docs.developers.optimizely.com>. [Data uzyskania dostępu: 2 April 2026].
- [16] Unleash, „Unleash Documentation,” [Online]. Available: <https://docs.getunleash.io>. [Data uzyskania dostępu: 2 April 2026].
- [17] Flagsmith, „Flagsmith Documentation,” [Online]. Available: <https://docs.flagsmith.com>. [Data uzyskania dostępu: 3 April 2026].
- [18] Featbit, „Featbit Documentation,” [Online]. Available: <https://docs.featbit.co>.
- [19] OWASP Foundation, „Secrets Management Cheat Sheet,” [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html. [Data uzyskania dostępu: 5 April 2026].
- [20] Google LLC, „Common Security Risks: Hardcoded Cryptographic Secrets,” [Online]. Available: <https://developer.android.com/privacy-and-security/risks/hardcoded-cryptographic-secrets>. [Data uzyskania dostępu: 6 April 2026].
- [21] Guardsquare, „Proguard Manual,” [Online]. Available: <https://www.guardsquare.com/manual/home>. [Data uzyskania dostępu: 7 April 2026].
- [22] Google LLC, „Enable app optimization with R8,” [Online]. Available: <https://developer.android.com/topic/performance/app-optimization/enable-app-optimization>. [Data uzyskania dostępu: 7 April 2026].
- [23] Guardsquare, „ProGuard vs DexGuard: A comparison guide,” [Online]. Available: <https://www.guardsquare.com/blog/dexguard-vs-proguard>. [Data uzyskania dostępu: 7 April 2026].
- [24] PreEmptive, „DashO User Guide,” [Online]. Available: <https://support.preemptive.com/hc/en-us/articles/31997880803729-Introduction>. [Data uzyskania dostępu: 8 April 2026].
- [25] Apple Inc., „XCode Build Settings Reference,” [Online]. Available: <https://developer.apple.com/documentation/Xcode/build-settings-reference>. [Data uzyskania dostępu: 7 April 2026].
- [26] M. Gutowski, „Swift Confidential Repository,” [Online]. Available: <https://github.com/securevale/swift-confidential>. [Data uzyskania dostępu: 7 April 2026].
- [27] Information security group of the University of Applied Sciences and Arts Western Switzerland of Yverdon-les-Bains, „OLLVM Wiki,” [Online]. Available: <https://github.com/obfuscator-llvm/obfuscator/wiki>. [Data uzyskania dostępu: 7 April 2026].
- [28] Guardsquare, „iXGuard Product Page,” [Online]. Available: <https://www.guardsquare.com/ixguard>.
- [29] Promon, „Promon Shield for Mobile,” [Online]. Available: <https://promon.io/products/shield-mobile>. [Data uzyskania dostępu: 7 April 2026].
- [30] OWASP Foundation, „TEST-0213: Use of Hardcoded Cryptographic Keys in Code,” [Online]. Available: <https://mas.owasp.org/MASTG/tests/ios/MASVS-CRYPTO/MASTG-TEST-0213/#overview>. [Data uzyskania dostępu: 7 April 2026].
- [31] OWASP Foundation, „TEST-0212: Use Of Hardcoded Cryptographic Keys in Code,” [Online]. Available: <https://mas.owasp.org/MASTG/tests/android/MASVS-CRYPTO/MASTG-TEST-0212/>. [Data uzyskania dostępu: 7 April 2026].

- [32] „Dex to Java decompiler,” [Online]. Available: <https://github.com/skylot/jadx>. [Data uzyskania dostępu: 7 April 2026].
- [33] OWASP Foundation, „MASTG-TECH-0026: Dynamic Analysis on Non-Rooted Devices,” [Online]. Available: <https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0026/>. [Data uzyskania dostępu: 7 April 2026].
- [34] OWASP Foundation, „MASTG-TECH-0068: Disassembling Native Code,” [Online]. Available: <https://mas.owasp.org/MASTG/techniques/ios/MASTG-TECH-0068/>. [Data uzyskania dostępu: 7 April 2026].
- [35] OWASP Foundation, „MASTG-TECH-0018: Disassembling Native Code,” [Online]. Available: <https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0018/>. [Data uzyskania dostępu: 7 April 2026].
- [36] Apple Inc., „The Secure Enclave,” [Online]. Available: <https://support.apple.com/pl-pl/guide/security/sec59b0b31ff/web>. [Data uzyskania dostępu: 8 April 2026].
- [37] Apple Inc., „Protecting keys with the Secure Enclave,” [Online]. Available: <https://developer.apple.com/documentation/security/protecting-keys-with-the-secure-enclave>. [Data uzyskania dostępu: 8 April 2026].
- [38] B. Roemmele, „Forbes: What Is Apple's New Secure Enclave And Why Is it Important?,” [Online]. Available: <https://www.forbes.com/sites/quora/2013/09/18/what-is-apples-new-secure-enclave-and-why-is-it-important/>. [Data uzyskania dostępu: 8 April 2026].
- [39] Apple Inc., „Keychain Data Protection,” [Online]. Available: <https://support.apple.com/en-gb/guide/security/secb0694df1a/web>. [Data uzyskania dostępu: 8 April 2026].
- [40] D. Hardt, „The OAuth 2.0 Authorization Framework,” Internet Engineering Task Force, 2012.
- [41] Google LLC, „Trusty TEE,” [Online]. Available: <https://source.android.com/docs/security/features/trusty>. [Data uzyskania dostępu: 10 April 2026].
- [42] ARM Ltd, „ARM TrustZone Technology,” [Online]. Available: <https://developer.arm.com/documentation/100690/0201/Arm-TrustZone-technology?lang=en>. [Data uzyskania dostępu: 10 April 2026].
- [43] Google LLC, „Android Keystore System,” [Online]. Available: <https://developer.android.com/privacy-and-security/keystore>. [Data uzyskania dostępu: 10 April 2026].
- [44] Google LLC, „Pixel 6: Setting a new standard for mobile security,” [Online]. Available: <https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html>. [Data uzyskania dostępu: 11 April 2026].
- [45] OWASP Foundation, „MASTG-TECH-0010: Basic Network Monitoring / Sniffing,” [Online]. Available: <https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0010/>. [Data uzyskania dostępu: 11 April 2026].
- [46] E. Rescorla, „The Transport Layer Security (TLS) Protocol Version 1.3,” Internet Engineering Task Force, [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc8446>. [Data uzyskania dostępu: 11 April 2026].

- [47] XK72, „SSL Proxying,” [Online]. Available: <https://www.charlesproxy.com/documentation/proxying/ssl-proxying/>. [Data uzyskania dostępu: 12 April 2026].
- [48] R. S. Borate, „System CA on Android: How to Install & Work Around Modern Restrictions,” [Online]. Available: <https://medium.com/@RoBoHackermann/system-ca-on-android-how-to-install-work-around-modern-restrictions-c570f000ab9a>. [Data uzyskania dostępu: 11 April 2026].
- [49] Apple Inc., „Identity Pinning: How to configure server certificates for your app,” [Online]. Available: <https://developer.apple.com/news/?id=g9ejcf8y>.
- [50] OWASP Foundation, „Pinning,” [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Pinning_Cheat_Sheet.html. [Data uzyskania dostępu: 11 April 2026].
- [51] OWASP Foundation, „MASTG-TECH-0012: Bypassing Certificate Pinning,” [Online]. Available: <https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0012/>. [Data uzyskania dostępu: 11 April 2026].
- [52] S. Newman, Building Microservices: Designing Fine-Grained systems, 2nd edition, O'Reilly, 2021, ISBN: 978-14-920-3397-4.
- [53] D. Feldman, E. Fox, E. Gilman, I. Haken, F. Katuz, U. Khan, M. Lembrecht, B. Lum, A. Martinez, E. Nesterov, A. Vega i M. Wardrop, Solving the Bottom Turtle — a SPIFFE Way to Establish Trust in Your Infrastructure via Universal Identity, 2020, ISBN: 978-0-578-77737-5.
- [54] Apple Inc., „Establishing your app's integrity,” [Online]. Available: <https://developer.apple.com/documentation/devicecheck/establishing-your-app-s-integrity>. [Data uzyskania dostępu: 12 April 2026].
- [55] Apple Inc., „Validating apps that connect to your server,” [Online]. Available: <https://developer.apple.com/documentation/devicecheck/validating-apps-that-connect-to-your-server>. [Data uzyskania dostępu: 12 April 2026].
- [56] Google LLC, „Overview of the Play Integrity API,” [Online]. Available: <https://developer.android.com/google/play/integrity/overview>. [Data uzyskania dostępu: 12 April 2026].
- [57] Google LLC, „Play Integrity API: Make a standard API request,” [Online]. Available: <https://developer.android.com/google/play/integrity/standard>. [Data uzyskania dostępu: 12 April 2026].
- [58] Apple Inc., „DeviceCheck Documentation,” [Online]. Available: <https://developer.apple.com/documentation/devicecheck>. [Data uzyskania dostępu: 13 April 2026].
- [59] R. Kohavi, D. Tang i Y. Xu, Trustworthy Online Controlled Experiments: A Practical Guide To A/B Testing, Cambridge University Press, 2020, ISBN: 978-1108724265.
- [60] C. Davis, Cloud Native Patterns: Designing Change-Tolerant Software, Manning, 2019, ISBN: 978-16383356851.
- [61] Google LLC, „Minimize the effect of regular updates,” [Online]. Available: <https://developer.android.com/develop/connectivity/minimize-effect-regular-updates>. [Data uzyskania dostępu: 15 April 2026].
- [62] G. Z. Georgiev, Statistical Methods in Online A/B Testing, 2019, ISBN: 978-1694079725.

- [63] Oracle, „Java SE 17 Documentation,” [Online]. Available: <https://docs.oracle.com/en/java/javase/17/>. [Data uzyskania dostępu: 20 5 2026].
- [64] Broadcom, „Spring Boot Reference Documentation,” [Online]. Available: <https://docs.spring.io/spring-boot/documentation.html>. [Data uzyskania dostępu: 20 May 2026].
- [65] Apache Software Foundation, „Apache Tomcat Documentation,” [Online]. Available: <https://tomcat.apache.org>. [Data uzyskania dostępu: 20 May 2026].
- [66] Project Lombok, „Project Lombok Repository,” [Online]. Available: <https://github.com/projectlombok/lombok>. [Data uzyskania dostępu: 20 May 2026].
- [67] Gradle Inc., „Gradle User Manual,” [Online]. Available: <https://docs.gradle.org/current/userguide/userguide.html>. [Data uzyskania dostępu: 20 May 2026].
- [68] H2 Database Engine, „H2 Database Documentation,” [Online]. Available: <https://h2database.com/html/main.html>. [Data uzyskania dostępu: 20 May 2026].
- [69] PostgreSQL Global Development Group, „PostgreSQL Documentation,” [Online]. Available: <https://www.postgresql.org/docs/>. [Data uzyskania dostępu: 20 May 2026].
- [70] R. Fielding, „Architectural Styles and the Design of Network-based Software Architectures,” UNIVERSITY OF CALIFORNIA, IRVINE, 2000. [Online]. Available: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Data uzyskania dostępu: 20 May 2026].
- [71] Internet Engineering Task Force, „RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format,” 2017. [Online]. Available: <https://www.rfc-editor.org/info/rfc8259/>. [Data uzyskania dostępu: 20 May 2026].
- [72] W3C / WHATWG, „Server-Sent Events Specification,” [Online]. Available: <https://html.spec.whatwg.org/multipage/server-sent-events.html>. [Data uzyskania dostępu: 21 May 2026].
- [73] PIMJO, „TailAdmin - Tailwind CSS Admin Dashboard Template,” [Online]. Available: <https://tailadmin.com>. [Data uzyskania dostępu: 21 May 2026].
- [74] Meta Inc., „React Documentation,” [Online]. Available: <https://react.dev>. [Data uzyskania dostępu: 21 May 2026].
- [75] Microsoft, „TypeScript Documentation,” [Online]. Available: <https://www.typescriptlang.org/docs/>. [Data uzyskania dostępu: 21 May 2026].
- [76] VoidZero, „Vite: Next Generation Frontend Tooling,” [Online]. Available: <https://vite.dev/guide/>. [Data uzyskania dostępu: 21 May 2026].
- [77] Axios, „Axios Documentation,” [Online]. Available: <https://axios.rest/pages/getting-started/first-steps>. [Data uzyskania dostępu: 21 May 2026].
- [78] Tailwind Labs Inc., „Tailwind CSS Documentation,” [Online]. Available: <https://tailwindcss.com/docs/installation/using-vite>. [Data uzyskania dostępu: 21 May 2026].
- [79] Shopify Inc., „React Router Documentation,” [Online]. Available: <https://reactrouter.com/home>. [Data uzyskania dostępu: 21 May 2026].
- [80] ApexCharts, „ApexCharts.js Documentation,” [Online]. Available: <https://apexcharts.com/docs/>. [Data uzyskania dostępu: 21 May 2026].

- [81] OpenJS Foundation, „ESLint Documentation,” [Online]. Available: <https://eslint.org/docs/latest/>. [Data uzyskania dostępu: 21 May 2026].
- [82] Apple Inc., „Swift Programming Language Documentation,” [Online]. Available: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/>. [Data uzyskania dostępu: 22 May 2026].
- [83] Apple Inc., „SwiftUI Documentation,” [Online]. Available: <https://developer.apple.com/documentation/swiftui/>. [Data uzyskania dostępu: 22 May 2026].
- [84] JetBrains, „Kotlin Programming Language,” [Online]. Available: <https://kotlinlang.org/docs/home.html>. [Data uzyskania dostępu: 22 May 2026].
- [85] Google LLC, „Jetpack Compose Documentation,” [Online]. Available: <https://developer.android.com/compose>. [Data uzyskania dostępu: 22 May 2026].
- [86] Google LLC, „Gson Documentation,” [Online]. Available: <https://github.com/google/gson>. [Data uzyskania dostępu: 22 May 2026].

Dodatki

Dodatek A: Spis rysunków

Rysunek 1. Sekwencja długiego odpytywania realizująca natychmiastową propagację konfiguracji w Firebase Remote Config	12
Rysunek 2. Panel konfiguracyjny flag funkcji w platformie Firebase	13
Rysunek 3. Panel konfiguracyjny flag funkcji w platformie Optimizely	16
Rysunek 4. Panel administracyjny Unleash	17
Rysunek 5. Architektura Bezpiecznej Enklawy na przykładzie smartfonu Apple iPhone	23
Rysunek 6. Architektura TEE i StrongBox na przykładzie smartfonu Google Pixel	25
Rysunek 7. Architektura Backend-for-Frontend w przypadku wielokanałowej dystrybucji aplikacji	27
Rysunek 8. Mechanizm działania atestacji środowiska wykonawczego przez Play Integrity API	29
Rysunek 9. Mechanizm atestacji środowiska wykonawczego przez Apple App Attest	31
Rysunek 10. Schemat blokowy ilustrujący logikę stanów flagi funkcji	36
Rysunek 11. Diagram przypadków użycia powiązanych z procesem zarządzania aplikacjami	37
Rysunek 12. Diagram przypadków użycia powiązanych z procesem zarządzania flagami funkcji.....	38
Rysunek 13. Diagram przypadków użycia powiązanych z przeprowadzaniem testów A/B ...	39
Rysunek 14. Diagram komponentów systemu zdalnej konfiguracji.....	40
Rysunek 15. Diagram sekwencji ilustrujący zależności między komponentami w realizacji przepływów danych dotyczących kluczowych przypadków użycia w domenie flag funkcji..	41
Rysunek 16. Diagram sekwencji ilustrujący proces przeprowadzania testów A/B – ilustracja pełnego drzewa przypadków użycia	42
Rysunek 17. Diagram klas przedstawiający model danych systemu	48
Rysunek 18. Diagram klas serwera Spring Boot.....	49
Rysunek 19. Widok zarządzania aplikacjami w panelu administratora	53
Rysunek 20. Widok manualnego tworzenia flagi funkcji	53
Rysunek 21. Widok tworzenia nowego testu AB	54
Rysunek 22. Widok podsumowania testu AB na podstawie liczby zarejestrowanych zdarzeń	55
Rysunek 23. Widok podsumowania testu AB na podstawie wpływu eksperymentu na licznosc zdarzenia.....	55
Rysunek 24. Wysokopoziomowy diagram architektury panelu administratora	56
Rysunek 25. Interfejs graficzny aplikacji testowej iOS	57
Rysunek 26. Interfejs graficzny aplikacji testowej Android	60
Rysunek 27. Zarejestrowana aplikacja testowa.....	63
Rysunek 28. Automatycznie stworzone flagi przed aktywacją.....	63

Dodatek B: Spis tabel

Tabela 1. Macierz funkcjonalności usług zdalnej konfiguracji.....	19
---	----

Dodatek C: Spis listingów

Listing 1. Struktura ścieżek API panelu administracyjnego	50
Listing 2. Struktura ścieżki końcówki strumieniującej	51
Listing 3. Przykład agregatu DTO	51
Listing 4. Przykład bazowego DTO	51
Listing 5. Definicja flagi funkcji w kodzie Swift	58
Listing 6. Procedura tworzenia obiektów atestacji po stronie kodu iOS	58
Listing 7. Klasa orkiestrująca atestacją po stronie aplikacji Android	61
Listing 8. Procedura odpowiedzialna za atestowaną komunikację z serwerem	61
Listing 9. Zrzut logów z aplikacji klienckiej	64
Listing 10. Zrzut logów z symulatora Android w momencie atestacji	64
Listing 11. Zrzut logów z symulatora iOS w momencie przeprowadzania atestacji	65