## PSD projekt Dawid Bartosiak Zuzanna Godek

### Projekt rozwiązania

W bazie Redis przechowywane są dane w formacie JSON per karta nt.:
id karty, id użytkownika, lokalizacji (długości i szerokości geograficznej), wartości średniej, odchylenia standardowego i limitu odnawialnego.

Generator tworzy transakcje, w których zawarte jest: card_id, user_id, value, lat, lon, timestamp, available_limit.

Anomalie uwzględnione w systemie [10]:

1. TransactionTenTimesTheAverage – wartość 10 razy większa niż średnia historyczna w oknie (SlidingWindow) per karta

```
class TenTimesTheAverage(ProcessWindowFunction):
    def open(self, runtime_context):
        self.avg_state = runtime_context.get_state(
            ValueStateDescriptor("avg", Types.DOUBLE())
        )
        self.redis_client = redis.Redis(host=REDIS_HOST, port=REDIS_PORT, decode_responses=True)

    def process(self, key, context, elements):
        card_id = key
        results = []

        avg = self.avg_state.value() or None
        if avg is None:
            redis_key = f"card:{card_id}"
            try:
                avg = float(self.redis_client.hget(redis_key, "avg_value"))
                self.avg_state.update(avg)
            except Exception as e:
                logger.error(f"Error parsing Redis avg_value for card {card_id}: {e}")

        values = []
        for e in elements:
            if avg is not None and e[1] >= 10 * avg:
                alarm = {
                    "alarm_time": e[0],
                    "card_id": card_id,
                    "value": e[1],
                    "average": avg,
                    "alarm_type": "TransactionTenTimesTheAverage"
                }
                results.append(json.dumps(alarm))
            if e[1] >= 0:
                values.append(e[1])

        new_avg = sum(values) / len(values) if len(values) > 0 else None
        if new_avg is not None:
            redis_key = f"card:{card_id}"
            try:
                self.redis_client.hset(redis_key, "avg_value", new_avg)
                self.avg_state.update(new_avg)
            except Exception as e:
                logger.error(f"Error writing to Redis for card {card_id}: {e}")

        return results
```

2. NegativeTransaction – ujemna transakcja

3. LimitExceeded – przekroczenie limitu karty
4. VeryHighValue – wartość większa niż 10000zł

```
class Negative_Above10k_Limit(KeyedProcessFunction):
    def process_element(self, value, ctx):
        results = []
        if value[1] < 0:
            results.append(json.dumps({
                'alarm_time': value[0],
                'card_id': value[2],
                'value': value[1],
                'alarm_type': 'NegativeTransaction'
            }))
        elif value[1] >= 10000:
            results.append(json.dumps({
                'alarm_time': value[0],
                'card_id': value[2],
                'value': value[1],
                'alarm_type': 'VeryHighValue'
            }))
        if value[4] <= 0.001:
            results.append(json.dumps({
                'alarm_time': value[0],
                'card_id': value[2],
                'value': value[1],
                'card_limit': value[4],
                'alarm_type': 'LimitExceeded'
            }))
        return results
```

5. ImpossibleTravel – bliskie transakcje daleko od siebie (np. z przemieszczeniem o prędkości ponad 900km/h) per karta

```
class ImpossibleTravel(KeyedProcessFunction):
    def open(self, runtime_context):
        self.last_tx_state = runtime_context.get_state(
            ValueStateDescriptor("last_tx", Types.PICKLED_BYTE_ARRAY())
        )

    def process_element(self, value, ctx):
        timestamp = value[0]
        card_id = value[2]
        lat = value[5]
        lon = value[6]

        last_tx = self.last_tx_state.value()

        if last_tx is not None:
            last_time, last_lat, last_lon = last_tx

            time_diff_sec = timestamp - last_time
            if time_diff_sec <= 0:
                self.last_tx_state.update((timestamp, lat, lon))
                return []

            time_diff_hr = time_diff_sec / 3600.0
            distance_km = calculate_distance(lat, lon, last_lat, last_lon)
            speed = distance_km / time_diff_hr
            if speed >= 900:
                alarm = {
                    "alarm_time": timestamp,
                    "card_id": card_id,
                    "current_location": {"lat": lat, "lon": lon},
                    "previous_location": {"lat": last_lat, "lon": last_lon},
                    "distance_km": round(distance_km, 2),
                    "time_diff_sec": time_diff_sec,
                    "estimated_speed_kmh": round(speed, 2),
                    "alarm_type": "ImpossibleTravel"
                }
                self.last_tx_state.update((timestamp, lat, lon))
                return [json.dumps(alarm)]

        self.last_tx_state.update((timestamp, lat, lon))
        return []
```

6. RapidTransactions – czasowe odstępy między transakcjami poniżej 10s per karta

```
class RapidTransactions(KeyedProcessFunction):
    def open(self, runtime_context):
        self.last_time_state = runtime_context.get_state(
            ValueStateDescriptor("last_tx_time", Types.DOUBLE())
        )

    def process_element(self, value, ctx):
        current_time = value[0]
        card_id = value[2]

        last_time = self.last_time_state.value()

        if last_time is not None and (current_time - last_time) < 10:
            alarm = {
                "alarm_time": current_time,
                "card_id": card_id,
                "previous_transaction_time": last_time,
                "current_transaction_time": current_time,
                "alarm_type": "RapidTransactions"
            }
            self.last_time_state.update(current_time)
            return [json.dumps(alarm)]

        self.last_time_state.update(current_time)
        return []
```

7. PinAvoidance – kilka transakcji pod rząd poniżej kwoty wymagającej potwierdzenia (100zł) per karta

```
class CloseTransactionsNoPin(KeyedProcessFunction):
    def open(self, runtime_context):
        self.last_transaction_state = runtime_context.get_state(
            ValueStateDescriptor("last_transaction", Types.PICKLED_BYTE_ARRAY())
        )

    def process_element(self, value, ctx):
        current_time = value[0]
        current_value = value[1]
        card_id = value[2]

        last_tx = self.last_transaction_state.value()

        if last_tx is not None:
            last_time, last_value = last_tx

            if current_value < 100 and last_value < 100 and (current_time - last_time) < 300:
                alarm = {
                    "alarm_time": current_time,
                    "card_id": card_id,
                    "previous_transaction_time": last_time,
                    "current_transaction_time": current_time,
                    "previous_value": last_value,
                    "current_value": current_value,
                    "alarm_type": "PinAvoidance"
                }
                self.last_transaction_state.update((current_time, current_value))
                return [json.dumps(alarm)]

        self.last_transaction_state.update((current_time, current_value))
        return []
```

8. ManyTransactionsNoPin – przynajmniej 5 uniknięć PINu w oknie (TumblingWindow) per karta

```
class ManyTransactionsNoPin(ProcessWindowFunction):
    def process(self, key, context, elements):
        results = []
        low_value_count = 0
        for e in elements:
            amount = e[1]
            if amount < 100:
                low_value_count += 1

        if low_value_count >= 5:
            alarm = {
                "alarm_time": elements[-1][0],
                "card_id": key,
                "count": low_value_count,
                "alarm_type": "ManyTransactionsNoPin"
            }
            results.append(json.dumps(alarm))

        return results
```

9. DormantCardActivity – długa przerwa, a potem dużo transakcji per karta

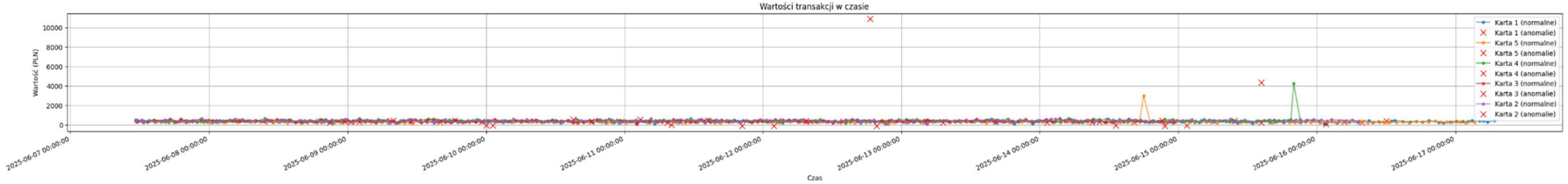
```
class BurstAfterInactivity(KeyedProcessFunction):
    def open(self, runtime_context):
        self.timestamps_state = runtime_context.get_list_state(
            ListStateDescriptor("recent_timestamps", Types.PICKLED_BYTE_ARRAY())
        )
        self.redis_client = redis.Redis(host=REDIS_HOST, port=REDIS_PORT, decode_responses=True)

    def process_element(self, value, context):
        tx_time = value[0]
        card_id = value[2]

        recent_timestamps = list(self.timestamps_state.get()) or []
        if not recent_timestamps:
            redis_key = f"card:{card_id}"
            last_transaction_time = self.redis_client.hget(redis_key, "last_transaction_time")
            self.redis_client.hset(redis_key, "last_transaction_time", tx_time)
            try:
                recent_timestamps = [float(last_transaction_time)]
            except Exception as e:
                logger.error(f"Error parsing Redis timestamps for card {card_id}: {e}")
                recent_timestamps = []

        recent_timestamps.append(tx_time)
        recent_timestamps = sorted(recent_timestamps)[-5:]
        self.timestamps_state.update(recent_timestamps)

        long_break_index = None
        for i in range(1, len(recent_timestamps)):
            gap = recent_timestamps[i] - recent_timestamps[i - 1]
            if gap > 30 * 24 * 60 * 60:  # 30 days
                # print("Gap bigger than 30 days!!!")
                long_break_index = i
                break

        if long_break_index is not None:
            burst_count = 1
            for i in range(long_break_index + 1, len(recent_timestamps)):
                prev = recent_timestamps[i - 1]
                curr = recent_timestamps[i]
                if curr - prev <= 3600:  # 60 minutes
                    burst_count += 1
                else:
                    break
            if burst_count >= 2:
                alarm = {
                    "alarm_time": recent_timestamps[long_break_index],
                    "card_id": card_id,
                    "burst_start": recent_timestamps[long_break_index],
                    "transactions_in_burst": burst_count,
                    "alarm_type": "DormantCardActivity"
                }
                return [json.dumps(alarm)]
        return []
```

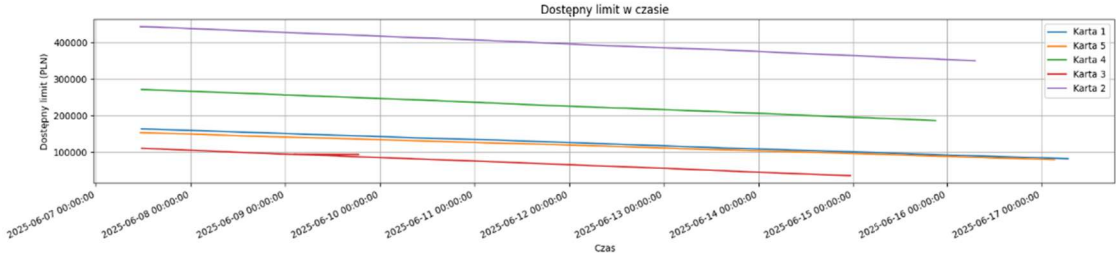
10. MultiCardDistance – transakcje różnymi kartami tego samego użytkownika w małym odstępie czasu a w dużej odległości (prędkość > 900km/h) per użytkownik

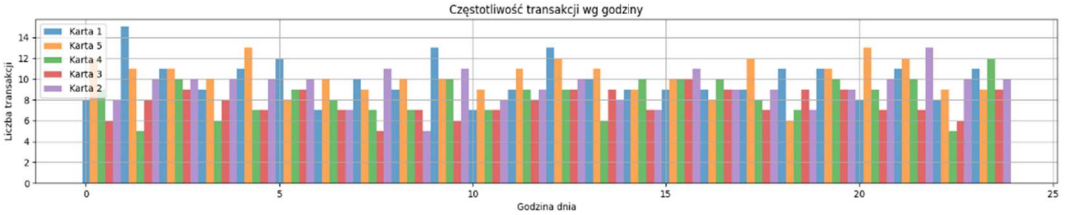
```
class MultiCardDistance(KeyedProcessFunction):
    def open(self, runtime_context):
        self.transactions_state = runtime_context.get_list_state(
            ListStateDescriptor("transactions", Types.PICKLED_BYTE_ARRAY())
        )

    def process_element(self, value, ctx):
        results = []
        user_id = ctx.get_current_key()

        transactions = list(self.transactions_state.get()) or []

        current_tx = value
        transactions.append(current_tx)

        transactions.sort(key=lambda x: x[0])

        current_card_id = current_tx[2]
        current_timestamp = current_tx[0]
        current_lat, current_lon = current_tx[5], current_tx[6]

        for prev_tx in transactions[:-1]:  # Exclude current transaction
            prev_card_id = prev_tx[2]
            if current_card_id == prev_card_id:
                continue


            prev_timestamp = prev_tx[0]
            prev_lat, prev_lon = prev_tx[5], prev_tx[6]

            time_diff = abs(current_timestamp - prev_timestamp)

            # Skip if no time difference to avoid division by zero
            if time_diff <= 0:
                continue


            dist_km = calculate_distance(current_lat, current_lon, prev_lat, prev_lon)
            time_diff_hr = time_diff / 3600.0
            speed = dist_km / time_diff_hr

            if speed >= 900:
                print(f"user_id: {user_id} card_id_1: {prev_card_id} card_id_2: {current_card_id} speed: {speed:.2f} km/h")
                alarm = {
                    "alarm_time": current_timestamp,
                    "user_id": user_id,
                    "card_id": current_card_id,
                    "card_id_2": prev_card_id,
                    "lat1": current_lat,
                    "lat2": prev_lat,
                    "lon1": current_lon,
                    "lon2": prev_lon,
                    "time_diff_seconds": time_diff,
                    "distance_km": round(dist_km, 2),
                    "estimated_speed_kmh": round(speed, 2),
                    "alarm_type": "MultiCardDistance"
                }
                results.append(json.dumps(alarm))
                transactions = []
                break

        if len(transactions) > 5:
            transactions = transactions[-5:]


        self.transactions_state.update(transactions)

        return results
```

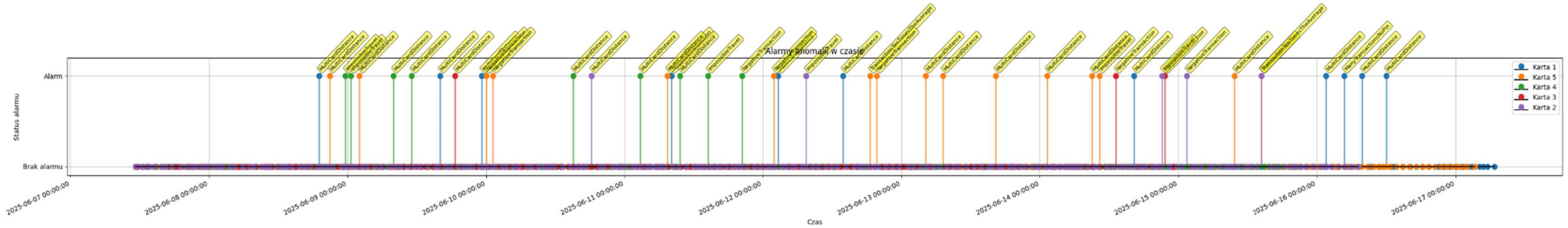
- Aplikacje – wszystkie zostały napisane w Pythonie. Uruchamiane w kolejności: inicjalizator -> generator -> plan przetwarzania Flinka -> wizualizator:
- Skrypt uruchomieniowy na inicjalizację do bazy wtedy jest wywoływany na początku, gdy nie ma wolumenu w Redisie. Dodatkowo daje możliwość nadpisania danych z Redisa.
- Generator – wszystkie transakcje generowane na 1 topic – generator sczytuje średnie z bazy danych a potem wokół nich generuje nowe dane. Dodatkowo z pewnym prawdopodobieństwem generowane są transakcje anomalne które zostały odpowiednio przygotowane, tak aby była pewność że w trakcie prezentacji one zajdą.
- Flink – przetwarza dane strumieniowo i generuje alarmy używając także SlidingWindow i TumblingWindow. Oparliśmy się o programowanie obiektowe i wyznaczenie klas dla każdego z opisywanych alarmów. Przesyłane są także dodatkowe informacje do Kafki, które posłużyć mogą debugowaniu programu, dostarczając znacznie bardziej precyzyjnych informacji o działaniu programu i występujących w nim ostrzeżeniach.
- Wizualizator – 4 różne tryby pracy – jest możliwość wyboru czy działamy per użytkownik, czy per karta oraz które alarmy chcemy wyświetlić. Dodatkowo generowana jest mapa z widokiem wszystkich transakcji. Łączy dwa strumienie danych prezentując na bieżąco przetwarzane dane.
- Redis, Flink (task manager, job manager), Kafka, Kafka-UI, redis-commander – wszystko w osobnych kontenerach skomponowanych z użyciem docker-compose. Dodatkowo kontener z Flinkiem został wzbogacony i zbudowany indywidualnie z Pythonem oraz odpowiednimi konektorami Kafki w osobnym Dockerfile.
- Dodatkowo – skrypty uruchomieniowe napisane w bashu:
    - scripts/t_setup.sh – odpowiadający za przygotowanie środowiska (uruchamiany tylko raz po pobraniu repozytorium),
    - scripts/t_run.sh – odpowiadający za uruchomienie całego projektu do momentu wizualizacji (gdyż wizualizacja dotyczy wybranych kart lub użytkowników),
    - scripts/t_submit_flink_job.sh – jako skrypt pomocniczy do przekazania zadania Flinkowi,
    - scripts/create-topics.sh – celem tworzenia topic'ów kafkowych z większą liczbą partycji,
    - scripts/debug.sh – pomagający rozwiązywać problemy z samą konfiguracją kontenerów.

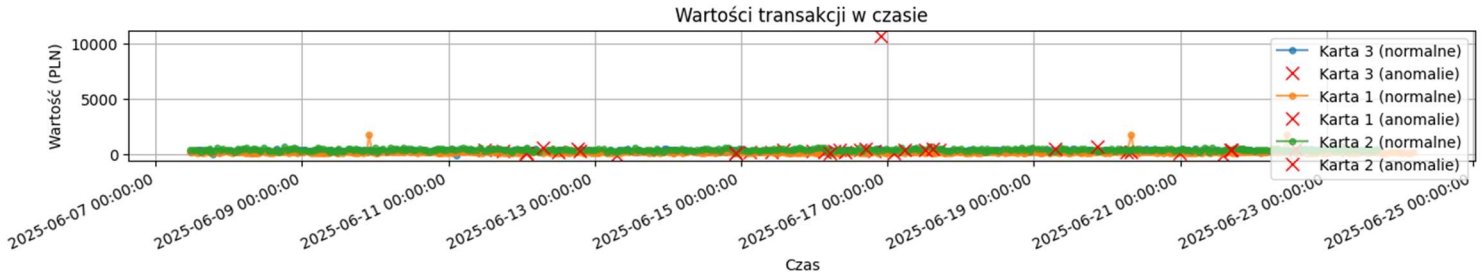
Działanie prezentujemy na zrzutach ekranu na następnych stronach:

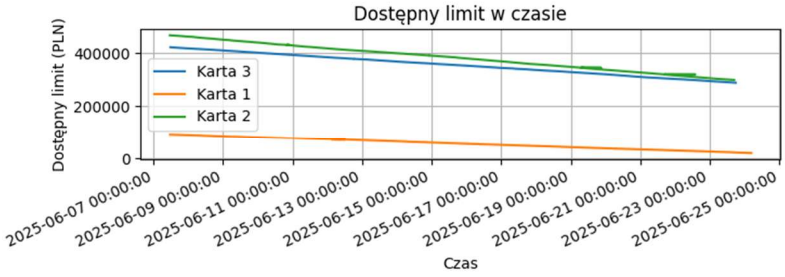

Wartości transakcji w czasie
Wartość (PLN)
Czas
Karta 1 (normalne)
Karta 1 (anomalie)
Karta 5 (normalne)
Karta 5 (anomalie)
Karta 4 (normalne)
Karta 4 (anomalie)
Karta 3 (normalne)
Karta 3 (anomalie)
Karta 2 (normalne)
Karta 2 (anomalie)

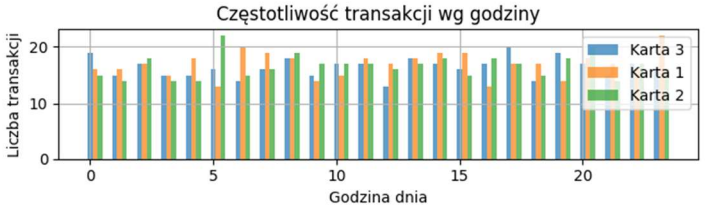

Dostępny limit w czasie
Dostępny limit (PLN)
Czas
Karta 1
Karta 5
Karta 4
Karta 3
Karta 2


Częstotliwość transakcji wg godziny
Liczba transakcji
Godzina dnia
Karta 1
Karta 5
Karta 4
Karta 3
Karta 2

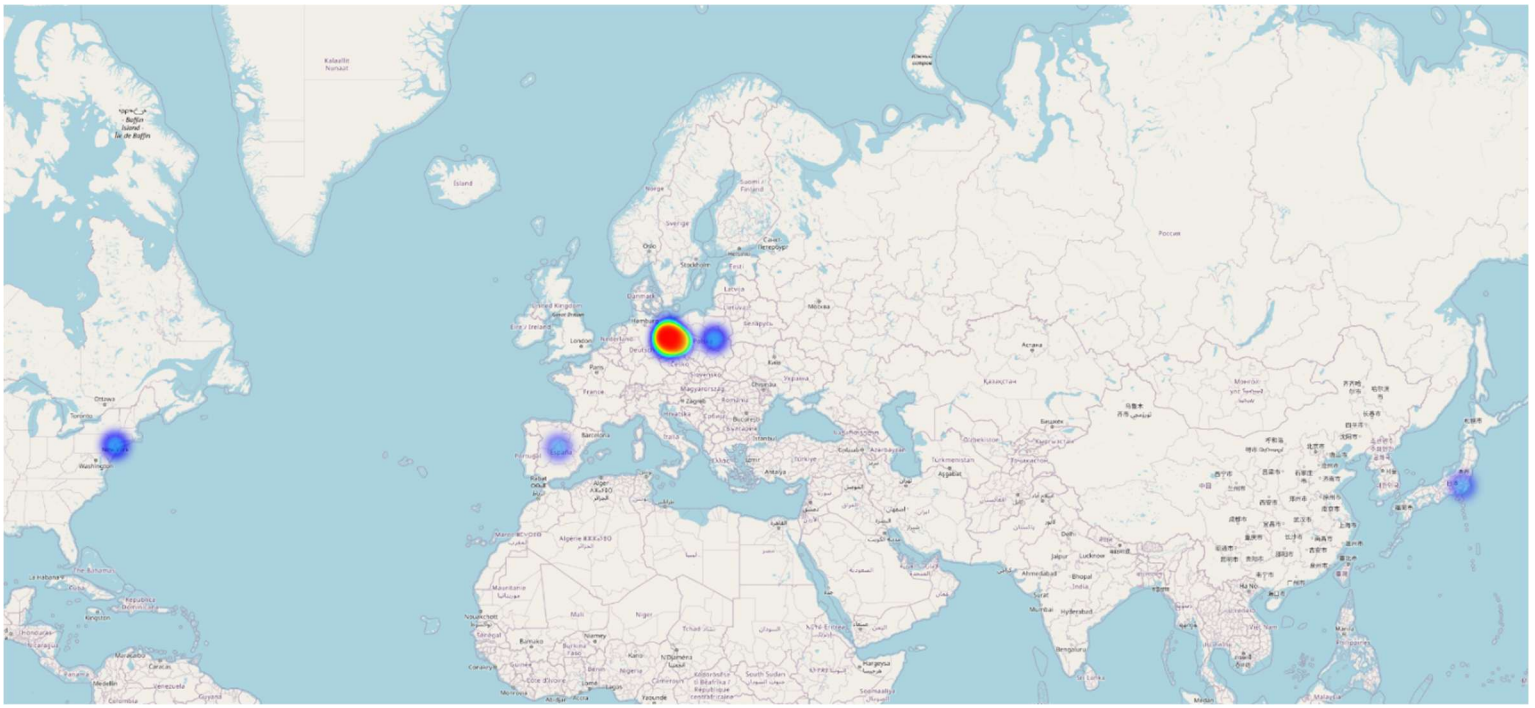

Alarmy anomalii w czasie
Status alarmu
Alarm
Brak alarmu
Czas
Karta 1
Karta 5
Karta 4
Karta 3
Karta 2


Wartości transakcji w czasie
Wartość (PLN)
10000
5000
0
Karta 3 (normalne)
Karta 3 (anomalie)
Karta 1 (normalne)
Karta 1 (anomalie)
Karta 2 (normalne)
Karta 2 (anomalie)
2025-06-07 00:00:00
2025-06-09 00:00:00
2025-06-11 00:00:00
2025-06-13 00:00:00
2025-06-15 00:00:00
2025-06-17 00:00:00
2025-06-19 00:00:00
2025-06-21 00:00:00
2025-06-23 00:00:00
2025-06-25 00:00:00
Czas


Dostępny limit w czasie
Dostępny limit (PLN)
400000
200000
0
Karta 3
Karta 1
Karta 2
2025-06-07 00:00:00
2025-06-09 00:00:00
2025-06-11 00:00:00
2025-06-13 00:00:00
2025-06-15 00:00:00
2025-06-17 00:00:00
2025-06-19 00:00:00
2025-06-21 00:00:00
2025-06-23 00:00:00
2025-06-25 00:00:00
Czas

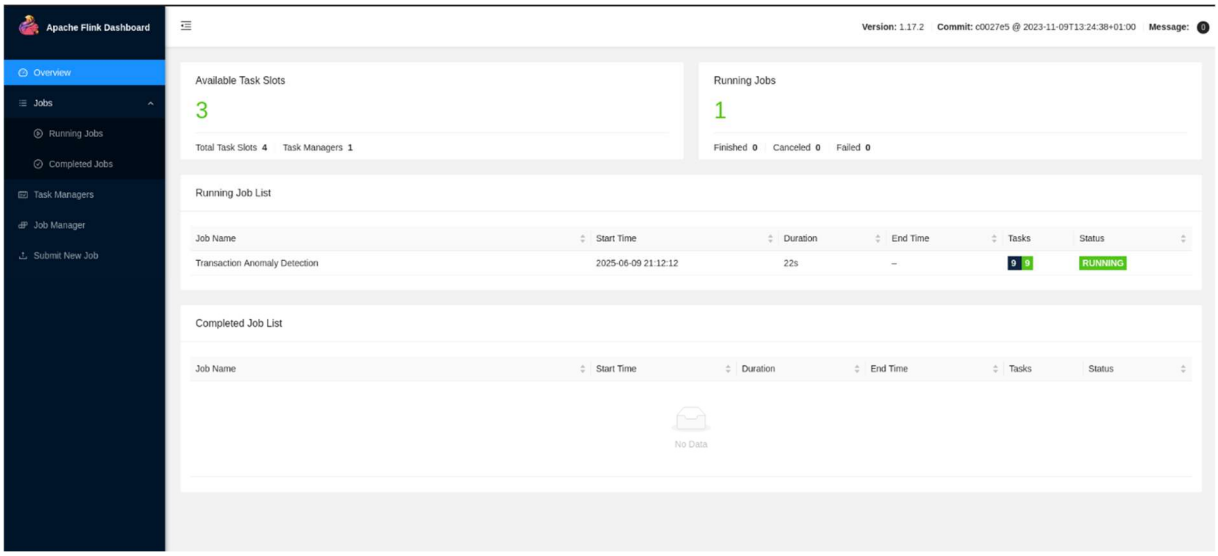

Częstotliwość transakcji wg godziny
Liczba transakcji
20
10
0
Karta 3
Karta 1
Karta 2
0
5
10
15
20
Godzina dnia


Alarmy - anomalii w czasie
Status alarmu
Alarm
Brak alarmu
Karta 3
Karta 1
Karta 2
MultiCardDistance
RapidTransactions
NegativeTransaction
PinAttemptsTimesTheAverage
ImpossibleTravel
2025-06-07 00:00:00
2025-06-09 00:00:00
2025-06-11 00:00:00
2025-06-13 00:00:00
2025-06-15 00:00:00
2025-06-17 00:00:00
2025-06-19 00:00:00
2025-06-21 00:00:00
2025-06-23 00:00:00
2025-06-25 00:00:00
Czas

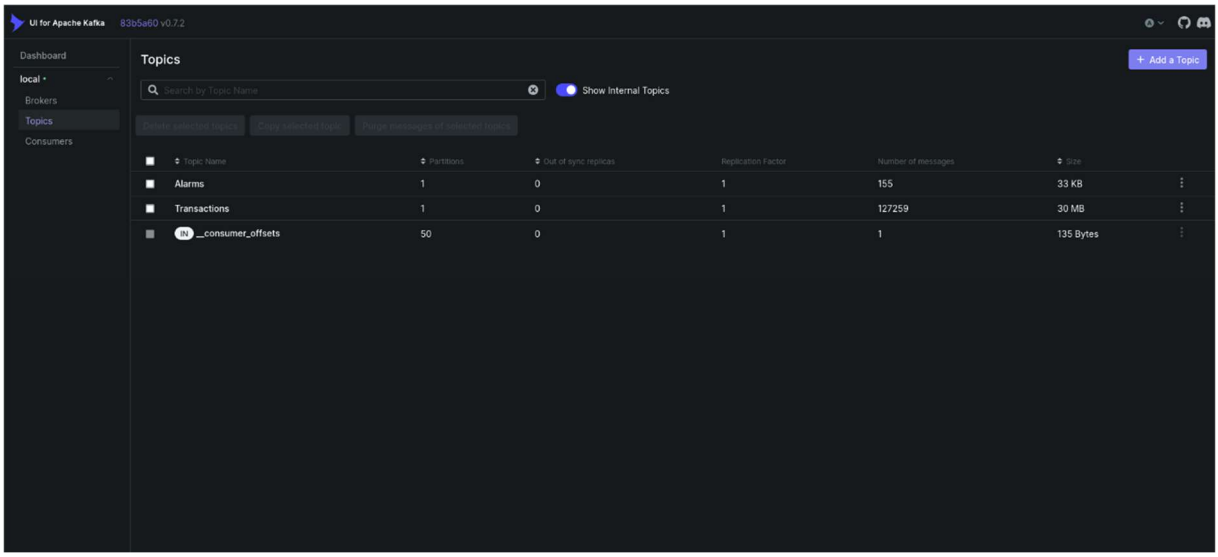


Oczywiście można mapę odpowiednio przybliżać i sprawdzać dokładne lokalizacje transakcji.

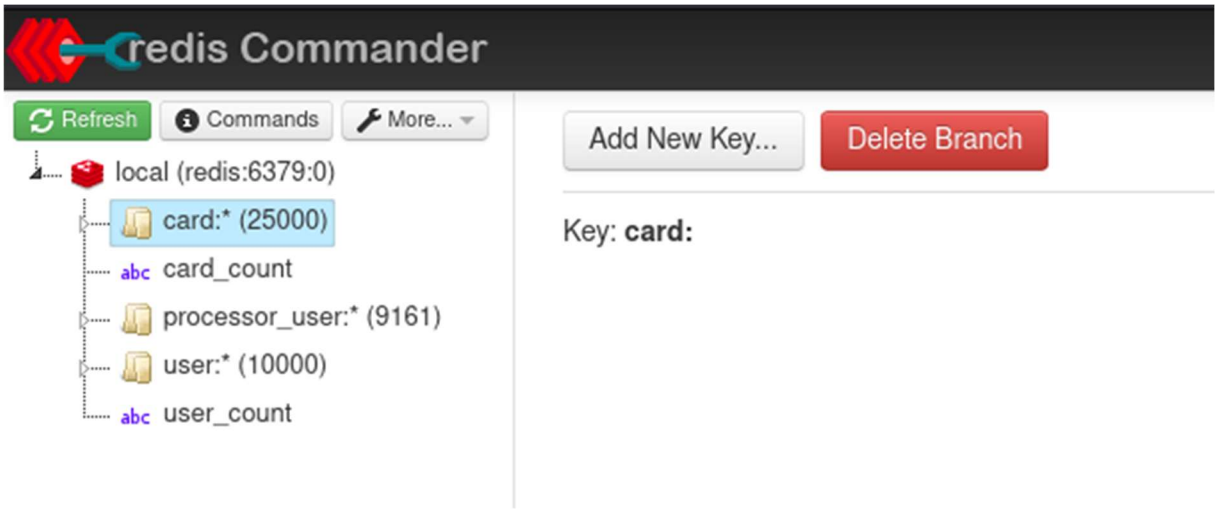
Poniżej przedstawiono narzędzia wykorzystywane podczas tworzenia projektu.




W interfejsie użytkownika wystawianym przez Flinka można zaobserwować, że zostało uruchomione zadanie w postaci running job oraz ile tasków aktualnie jest w użyciu co odpowiada skonfigurowanym przepływom (sinks) w kodzie. Używano tej aplikacji do monitorowania oraz debugowania działania Flinka.




Powyżej widać UI Kafki, na którym można zobaczyć wszystkie aktywne topic'i. Umożliwia ono także podglądanie wiadomości publikowanych na każdym topic'u. To rozwiązanie ułatwiało weryfikację generowanych transakcji i alarmów. Dodatkowo pozwala na szybkie czyszczenie wiadomości na poszczególnych topic'ach, co z kolei przyspiesza proces debugowania.




W interfejsie graficznym Redisa możliwie jest sprawdzenie zawartości bazy danych przechowującej karty i użytkowników. Redis commander pozwala na podglądanie grup jak i pojedynczych wpisów w bazie. Stan bazy widoczny na zdjęciu to 25000 kart i 10000 użytkowników.


W ramach projektu napisano także autorskie skrypty w języku powłoki bash ułatwiające uruchamianie komponentów systemu. Na powyższym zdjęciu widoczne jest wywołanie t_run.sh, który umożliwia uruchomienie całego systemu na raz, czyli wszystkich kontenerów potrzebnych do generowania transakcji oraz alarmów.

Prezentowany system to kompletne i efektywne narzędzie do monitorowania transakcji kartowych w czasie rzeczywistym, które skutecznie wykrywa szeroki zakres anomalii. Dzięki wykorzystaniu technologii strumieniowego przetwarzania danych oraz konteneryzacji, rozwiązanie działa bezproblemowo, jest skalowalne, a jego przejrzysty interfejs wizualny pozwala na szybką identyfikację i reagowanie na potencjalne zagrożenia. Cała implementacja, od generowania danych po wizualizację alarmów, jest spójna i funkcjonalna, co potwierdzają przedstawione zrzuty ekranu i możliwość interakcji z mapą transakcji.