



Wydział Finansów i Zarządzania w Bydgoszczy

Kierunek: Informatyka w Biznesie

Promotor: dr Krystian Jędrzejczak

**Praca magisterska**

# **REAKTYWNY SYSTEM STRUMIENIUJĄCY FAST DATA Z ANALIZĄ PREDYKCYJNĄ MACHINE LEARNING**

Autor

Marcin Gromadzki

nr albumu 41772

Akceptacja Promotora

.....

podpis Promotora

Bydgoszcz 2020



## SPIS TREŚCI

### Wstęp 5

1. Operacyjne Big Data oraz Machine Learning .....	9
1.1 Ewolucja Big Data w dane wysokiej prędkości Fast Data .....	9
1.2 SQL, NoSQL a NewSQL – wykorzystanie shardingu oraz replikacji .....	14
1.3 Programowanie konkurencyjne, nieblokujące i asynchroniczne, wykorzystanie paralelizacji .....	18
1.4 Systemy reaktywne z architekturą mikroservisową w chmurze .....	23
1.5 Actor model – konkurencyjne przetwarzanie rozproszone .....	27
1.6 Sztuczna inteligencja – machine learning oraz deep learning .....	32
1.7 Orkiestracja oraz komunikacja systemu w chmurze, przetwarzanie strumieniowe .....	43
2. Komponenty reaktywnego systemu rozproszonego Big Data .....	48
2.1 Scala – scalable programming language .....	48
2.2 Lagom Framework (Play Framework oparty o Akka) .....	52
2.3 Cassandra – nierelacyjna baza danych big data (NoSQL) .....	54
2.4 Spark – strumieniowe przetwarzanie danych big data .....	57
2.5 Spark ML – analiza danych machine learning .....	61
2.6 Kafka – asynchroniczny broker wiadomości .....	63
2.7 Wzorce architektury mikroservisowej: CQRS, ES, DDD .....	67
3. Implementacja systemu reaktywnego strumieniującego dane Fast Data .....	73
3.1 Architektura rozwiązania .....	73
3.2 Implementacja systemu reaktywnego .....	76
3.3 Interpolator – symulacja zdarzeń użytkowników .....	88
3.4 Konstrukcja modelu predykcyjnego .....	91
4. Analiza predykcyjna czasu rzeczywistego regresji ceny transakcji .....	97

4.1	Przebieg procesu predykcji .....	97
4.2	Trening modelu uczenia maszynowego .....	99
4.3	Ewaluacja z obserwacją błędów predykcji oraz kontekstowym re- treningiem .....	101
4.4	Wnioski, znaczenie biznesowe oraz rekomendacje dla adaptujących .....	103
Zakończenie .....		108
Bibliografia.....		111
Spis rysunków .....		118
Spis kodów źródłowych .....		119

## Wstęp

Współczesny świat branży IT staje przed dwoma wyzwaniami – skalą oraz danymi. Wraz z nadejściem ery Internetu dawne podejście do poszerzania skali działania poprzez dokładanie coraz większych zasobów do maszyny fizycznej, uruchamiającej główne oprogramowanie, przestało być wystarczające. Osiągnięcie jeszcze większej skali działania wymagało zrezygnowania z przerostu maszyn w superkomputery na korzyść rozwiązań horyzontalnych, gdzie maszyny były słabsze, ale wspólnie stanowiły sumę mocy obliczeniowej pod oprogramowanie działające w paradygmacie rozproszonym. Następstwem tego podejścia był rozwój technologii cloud computing (pol. chmurowych), które udostępniały wirtualne zasoby dostosowane do konkretnego oprogramowania w formie infrastruktury i/lub platformy. Tradycyjne narzędzia przechowywania danych stanowiły uporządkowaną i bezpieczną formę, w postaci RDBMS. Przestały być one wystarczające z dynamicznym wzrostem wolumenów w czasie rozwoju Internetu, pomimo intensywnej optymalizacji zapytań oraz indeksowania. Aby sprostać nowym wyzwaniom ponownie rozwinęły się technologie NoSQL, które pomimo luźniejszego podejścia do bezpieczeństwa, zapewniały możliwość skalowania horyzontalnego, tak niezbędnego w obecnej skali operacyjnej naszych czasów. Następstwem wzrostu ilości danych jest rozwój podejścia data-driven (pol. napędzanego danymi), szczególnie rozwinął się sektor analityki, a w tym również tzw. data science ze szczególnym naciskiem na zastosowania algorytmów Machine Learning, czy też ogólnie pojętego użycia sztucznej inteligencji. Obydwie fale rozwoju uzupełniają się wzajemnie, infrastruktura danych potrzebuje skalowalności, biznes oczekuje dostępności oprogramowania dla wszystkich niezależnie jak wielka to będzie skala, algorytmy uczenia maszynowego wymagają dużej ilości danych do przeprowadzania predykcji wysokiej dokładności – stoją za tym eksperci dziedziny oraz infrastruktura mogąca wprowadzić w życie najambitniejsze idee naszych czasów.

Biznes największych z innowacyjnych instytucji branży IT rozwija się niezwykle dynamicznie oraz wzrasta w zasięgu docierając do miliardów ludzi. Poza ogromnymi wymaganiami skali, posiadają one dostęp do masywnych wolumenów danych generowanych przez użytkowników. Firmy tzw. *modern enterprise* jako pierwsze stanęły przed omówionymi problemami, osiągając rewolucyjne innowacje na polu cloudowych systemów rozproszonych, Big Data oraz Machine Learning. Wprowadzają

one w swojej ofercie usługi takie, jak automatyczni asystenci, translatory językowe, synteza mowy, rozpoznawanie obiektów na obrazach, autonomiczne pojazdy i inne. Prócz wykorzystania wartości z posiadanych danych istotnym stało się działanie operacyjne czasu rzeczywistego, w wiecznie podłączonym Internecie, zapoczątkowało to tranzycję z dotychczasowego podziału systemów na analityczne oraz operacyjne, dążąc do zunifikowanego rozwiązania operacyjnego, z natychmiastowym wykorzystaniem wartości z danych w jednoczesnej analizie. Aby było to możliwe coraz szerzej korzysta się ze strumieniowania danych w postaci przetwarzania Fast Data.

Celem niniejszej pracy jest badanie rozwiązań umożliwiających transformację tradycyjnej analizy ETL / Big Data na korzyść natychmiastowej analizy operacyjnej Fast Data, z wykorzystaniem algorytmów Machine Learning w czasie rzeczywistym:

- w konsekwencji opracowanie kompletnej infrastruktury skalowalnego reaktywnego systemu rozproszonego strumieniującego Fast Data do ewaluacji czasu rzeczywistego modeli uczenia maszynowego
- jak również uzyskanie wysokiego poziomu dokładności predykcji w stworzonym modelu regresji uczenia maszynowego
- ponadto identyfikacja technologicznych czynników sukcesu firm wykorzystujących wartość z danych do osiągnięcia przełomowych innowacji.

W pracy wykorzystano szereg metod badawczych, m. in. w rozdziałach poświęconych omawianiu technologii Big Data oraz Machine Learning, a także w architekturze systemu reaktywnego oraz w stosowanych narzędziach posłużono się metodą analizy literatury. Praca została napisana głównie w oparciu o zagraniczne źródła internetowe, poradniki stosowanych narzędzi, artykuły technologiczne, opinie i punkty widzenia innych badaczy oraz opracowania tematyki z konkluzjami doświadczeń, a także na podstawie wiedzy własnej autora. Na potrzeby rozdziałów przedstawiających implementację rozwiązania oraz jego funkcjonowanie wykorzystano metody badania empirycznego i analizy przypadku. Natomiast wykonanie ewaluacji modelu wraz z re-treningiem przeprowadzono w formie eksperymentu.

Praca składa się z 4 rozdziałów oraz wstępu i zakończenia.

W rozdziale pierwszym omówiono czym jest Big Data oraz Machine Learning, jaka była ich droga ewolucji, jakie koncepcje i pojęcia za nimi stoją, dlaczego są istotne, a także jakie są ich najistotniejsze przypadki użycia. Przedstawiono paradygmaty inżynierii oprogramowania używane przy systemach rozproszonych, takie jak system reaktywny oraz Actor model, objaśniono również koncepcje asynchroniczności, konkurencyjności, czy paralelizacji. Sekcje poświęcone narzędziom bazodanowym omawiają NoSQL oraz NewSQL i przyjęte w nich strategie zapewniania skalowalności, czyli sharding oraz replikacja. Opisano również czym jest sztuczna inteligencja oraz jej rozwiązania. Poświęcono również uwagę komunikacji komponentów w chmurze wraz z przetwarzaniem strumieniowym Fast Data.

Rozdział drugi opisuje zestaw narzędzi wykorzystywanych w aplikacjach skalowalnych, czasu rzeczywistego oraz napędzanych danymi, tzw. SMACK stack. Omówiono język programowania Scala, framework Lagom oparty o Akka oraz Actor model. Przedstawiono nierelacyjną bazę danych Cassandra, a w części analitycznej Fast Data ukazano Apache Spark oraz Spark ML dla uczenia maszynowego. W postaci Apache Kafka przedstawiono komunikację między komponentami poprzez event bus. Skupiono się również na wzorcach projektowych, które wspomagają pracę nad dużymi, skalowalnymi oprogramowaniami, w postaci DDD, CQRS, ES. Omówione narzędzia zostają następnie wykorzystane w implementacji badanego przypadku systemu reaktywnego Fast Data z analizą Machine Learning.

Rozdział trzeci prezentuje opracowaną implementację systemu reaktywnego strumieniującego dane Fast Data wraz z regresyjną analizą predykcyjną Machine Learning. Poświęcono uwagę architekturze rozwiązania oraz wysokopoziomowym opisie jej elementów wraz z relacjami, jakie pomiędzy nimi zachodzą. Opisane zostały najistotniejsze elementy kodowe prototypu w formie listingów, omówiono punkty komunikacji pomiędzy komponentami oraz przedstawiono kompletną drogę przebiegu systemu, od interpolatora, poprzez logikę mikroservisów aż do konstrukcji modelu predykcyjnego.

W ostatnim, czwartym rozdziale poświęcono uwagę analizie predykcyjnej uczenia maszynowego w opracowanej implementacji systemu reaktywnego. Przedstawiono przebieg procesu predykcji, selekcję danych wykorzystanych do symulowania sytuacji

historycznej w eksperymencie poprzez interpolator, proces treningu oraz ewaluacji pod kątem oceny poziomu błędów. Dodatkowo wprowadzono czym jest kontekstowy re-trening i dlaczego jest on istotny w przypadku użycia. Na koniec omówiono finalne wnioski z badania oraz zwrócono uwagę na znaczenie biznesowe, wraz z rekomendacjami dla adaptujących.



# 1. Operacyjne Big Data oraz Machine Learning

W rozdziale omówiono pojęcia, koncepcje, genezę rozwoju oraz aspekty technologiczne rozwiązań Big Data oraz Machine Learning. Za szczególnie istotne przyjęto ich kontekst działania operacyjnego, zamiast skupiać się na analityce w oddzieleniu od ruchu online. Wprowadzono idee inżynierii oprogramowania skupionej na wydajnym wykorzystaniu zasobów, takie jak asynchroniczność, konkurencyjność, paralelizacja i paradygmaty używane przy systemach rozproszonych, takie jak system reaktywny czy Actor model. Omówiono również zastosowanie rozwiązań NoSQL oraz NewSQL w nowym podejściu do skalowalnego zarządzania danymi. Przygotowano również obszerną sekcję definiującą czym jest sztuczna inteligencja oraz jej poszczególne terminologie wraz z przykładami. Skupiono się także na omówieniu koncepcji architektury chmurowej, w tym komunikacji pomiędzy komponentami systemu oraz przetwarzania strumieniowego w myśl Fast Data.

## 1.1 Ewolucja Big Data w dane wysokiej prędkości Fast Data

Big Data (ang. „duże dane”) oznacza duże ilości danych, zarówno ustrukturyzowanych jak i nieustrukturyzowanych, które są zmienne i różnorodne, a także cechują się dużą prędkością, od momentów szczytowych pozyskiwania po wysoką częstość ich przetwarzania – ich analiza stanowi wyzwanie będące nietrywialnym zadaniem, natomiast przeprowadzona z sukcesem prowadzi do uzyskania wartościowej wiedzy<sup>1</sup> oraz <sup>2</sup>. Klasyczne rozumienie technologii Big Data reprezentuje charakterystyka modelu 5V przedstawionego na Rys.1.

Rys.1. Model 5V – pięć obszarów Big Data

---

<sup>1</sup> *What is Big Data?* [online] SAS Institute Inc. [dostęp: 20.08.2019]. Dostępny w Internecie: [https://www.sas.com/en\\_us/insights/big-data/what-is-big-data.html](https://www.sas.com/en_us/insights/big-data/what-is-big-data.html)

<sup>2</sup> *What is Big Data?* [online] Oracle Corporation [dostęp: 20.08.2019]. Dostępny w Internecie: <https://www.oracle.com/big-data/guide/what-is-big-data.html>



Źródło: Charakterystyka Big Data - *Analyzing Big Data in MicroStrategy* [online] MicroStrategy, Inc. [dostęp: 20.08.2019]. Dostępny w Internecie: [https://www2.microstrategy.com/producthelp/archive/10.7/WebUser/WebHelp/Lang\\_1033/Content/mstr\\_big\\_data.htm](https://www2.microstrategy.com/producthelp/archive/10.7/WebUser/WebHelp/Lang_1033/Content/mstr_big_data.htm)

Cechy Big Data wg modelu 5V<sup>3</sup> oraz <sup>4</sup>:

- *Volume* (ang. objętość) – ogromne wolumeny danych, poza standardami pozwalającymi na ich efektywne przetwarzanie, liczone w ponad peta- i eksabajtach
- *Velocity* (ang. prędkość) – duża tendencja przyrostowa danych w czasie ze skrajnymi momentami szczytowymi oraz wysoką częstością ich przetwarzania
- *Variety* (ang. różnorodność) – obejmowanie wielu typów danych, w tym nieustrukturyzowanych, binarnych, wideo, dźwiękowych, itd. oraz integrowanie wielu ich dostępnych źródeł, np. z portali społecznościowych, giełdowych, statystycznych, czy też tzw. „deep web” itp.
- *Veracity* (ang. prawdziwość) – wiarygodność oraz dokładność przetwarzanych danych w kontekście do osiągnięcia prawidłowych wyników analiz

---

<sup>3</sup> Goczyła K., *Big Data i 5V – Nowe wyzwania w świecie danych* VII Krajowa Konferencja Naukowa INFOBAZY 2014. Inspiracja - Integracja – Implementacja, Gdańsk: Politechnika Gdańska, Centrum Informatyczne TASK, 2014, s.21-23

<sup>4</sup> *Big Data: The 5 Vs Everyone Must Know* [online] Bernard Marr [dostęp: 26.08.2019]. Dostępny w Internecie: <https://www.linkedin.com/pulse/20140306073407-64875646-big-data-the-5-vs-everyone-must-know>

- *Value* (ang. wartość) – wypracowanie wartości z pozyskanych danych oraz przeprowadzonych analiz, mającej znaczący wpływ na wyciąganie wniosków ze zjawisk oraz proces podejmowania decyzji

Podjęcie klasyczne w rozwoju Big Data obrazuje jego ewolucję jako sukcesora dla hurtowni danych (ang. Data Warehousing), będących ówczesnym standardem przetwarzania analitycznego dużych zbiorów. Wraz z przyrostem objętościowym hurtowni nie były w stanie sprostać oczekiwaniom analitycznym wobec dynamicznie rosnących wolumenów. Firmy pionierskie doby Internetu (m. in. Google, Yahoo!, a następnie Facebook, Twitter czy Amazon) musiały wyjść naprzeciw wymaganiom rosnącej globalnej sieci, która generowała wcześniej niespotykane rozmiary danych, dla których dotychczas istniejące narzędzia przetwarzania okazywały się bezskuteczne<sup>5</sup>. Stanowiło to motor napędowy dla powstania nowych technologii przetwarzania Big Data, które zapoczątkowały rozwój open-source tej dziedziny inżynierii na globalną skalę.

Podstawową limitacją przetwarzania Big Data okazała się nieskalowalność architektury sprzętowej oraz oprogramowania – rozwiązania IT początków XXI-wieku postrzegały skalowalność wertykalnie, wyposażając się w coraz to wydajniejsze serwery oraz komponenty sprzętowe. Natomiast zamknięta struktura oprogramowania utrudniała lub też uniemożliwiała skalowalność horyzontalną. Wymogom problemów klasy Big Data nie mogły sprostać nawet najwydajniejsze maszyny, m. in. napotykając trudności z operowaniem na dużej porcji informacji przechowywanej w pamięci RAM oraz odczytu / zapisu z dysku twardego, a także granicą przerobową procesorów wielordzeniowych, gdzie model wielowątkowy oprogramowania powodował coraz mniejszy zysk wydajnościowy z paralelizacji w modelu współdzielonej pamięci, a nadmiar wątków skutkował coraz mniej efektywnym planowaniem czasu procesora przez dyspozytor systemu operacyjnego. Niemożliwość procesowania Big Data w istniejącym paradygmacie wymagała transformacji w kierunku architektury prosto skalowalnej – osiągnięcie jej powiodło się firmie Google, która opracowała model

---

<sup>5</sup> *A Short History of Big Data* [online] Dr Mark van Rijmenam [dostęp: 26.08.2019]. Dostępny w Internecie: <https://datafloq.com/read/big-data-history/239>

MapReduce – bazujący na przetwarzaniu rozproszonym (ang. distributed computing) na dużych klastrach wielu maszyn jednocześnie<sup>6</sup>.

Przewaga w zastosowaniu MapReduce opierała się na modelu przetwarzania danych, który łatwo było rozproszyć na klastrze wielu przeciętnie wydajnych maszyn. Przygotowanie tzw. joba (zadania) MapReduce obejmuje zdefiniowanie fazy algorytmu Map, który przetwarza dane wejściowe w kolekcję klucz – wartość, a następnie fazy Reduce, której zadaniem jest scalenie kolekcji wynikowej etapu Map, do formy danych pożądanых w analizie. Tak spreparowane dwuetapowe przetwarzanie pozwoliło na partycjonowanie danych w ramach algorytmu, który z powodzeniem rozprasza obliczenia wielu funkcji Map oraz Reduce poprzez wiele maszyn podłączonych w klastrze, oferując przewidywalną skalowalność horyzontalną, możliwą do zastosowania w skali produkcyjnej. Popularyzacja podejścia MapReduce stanowiła podwaliny pod opracowanie dedykowanego narzędzia analitycznego będącego następnikiem niewystarczających możliwości hurtowni danych – Apache Hadoop – ekosystemu, który zdefiniował współczesne rozumienie technologii Big Data<sup>7</sup>.

Ekosystem Hadoop stanowi kompletne rozwiązanie Big Data oparte o model MapReduce, od przetwarzania, po partycjonowanie i przechowywanie, do zapewniania zdolności analitycznych i rozproszonej paralelizacji obliczeń ogromnych zbiorów danych. Zapewnił orkiestrację komponentów i ich niezależny rozwój, w efekcie w przeciągu kilkunastu lat powstało ponad kilkadziesiąt narzędzi Big Data reprezentujących różnorodne podejścia, sprawdzające się w szczególnych scenariuszach wykorzystania, nadające dalszy kierunek postępowi w obszarze dużych zbiorów danych. Operowanie zgodnie z paradygmatem Hadoop / MapReduce opierało się na przetwarzaniu wsadowym, który sprawdził się przy hurtowniach danych. Jednakże przy Big Data stanowił coraz większe ograniczenia – rozproszenie danych po systemie Hadoop zajmowało znaczne ilości czasu, podobnie jak ich przetworzenie wiązało się z długim oczekiwaniem na efekty analiz i długotrwałą zajętość maszyn. Model MapReduce nie był projektowany z myślą analizy czasu rzeczywistego, a wraz z rosnącą objętością ogromnego wolumenu danych, przetwarzanie zwracało przestarzałe

---

<sup>6</sup> *MapReduce: Simplified Data Processing on Large Clusters* [online] Google AI [dostęp: 26.08.2019]. Dostępny w Internecie: <https://ai.google/research/pubs/pub62>

<sup>7</sup> *The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.* [online] The Apache Software Foundation. [dostęp: 27.08.2019]. Dostępny w Internecie: <https://hadoop.apache.org>

wyniki, mające coraz mniejszą wartość biznesową<sup>8</sup>. Dodatkowym utrudnieniem okazała się toporna praca z przetworzonymi danymi wynikowymi, bez możliwości ich interaktywnego modyfikowania, które jest elementem bardzo powszechnym przy operacjach data miningowych, przeprowadzanie nawet minimalnie zmienionej analizy wiązało się więc z ponownym przetworzeniem danych wejściowych. Aby sprostać wyzwaniom Big Data nowej generacji potrzebne były narzędzia przetwarzania rzeczywistego, tak powstał oparty na podejściu strumieniowym, wykorzystujący pamięć RAM do przechowywania zamiast przestrzeni dyskowej – Apache Spark, oferując paralelizację przetwarzania w klastrach maszyn do 100 razy szybciej, niż w klasycznym MapReduce<sup>9</sup>.

Biznesowa potrzeba przetwarzania czasu rzeczywistego rozpropagowała podejście strumieniowe i ustanowiła trend wyboru narzędzia Spark deprecjonując klasyczne podejście oparte na Hadoop / MapReduce. Zastosowanie nowatorskiego rozwiązania zwiększyło wydajność i efektywność, a także zapewniało pożądaną elastyczność w analizie danych przez Data Scientist'ów. Zyskano szybsze uzyskiwanie wyników oraz wcześniejsze reagowanie na trendy wynikające z danych, istotne dla procesu decyzyjnego w przedsiębiorstwach. Kluczowym elementem tej zmiany jest jedna z cech modelu 5V – *Velocity* (ang. prędkość) – to właśnie ciągłe przyspieszanie: rośnięcie objętości wolumenu, częstości analiz, tempa generowania nowych danych poprowadziła ewolucję Big Data w kierunku danych wysokiej prędkości *Fast Data*.

Istota Fast Data to strumieniowe systemy danych – oprogramowanie, które zostało zredefiniowane na potrzeby przetwarzania ogromnych wolumenów danych w czasie rzeczywistym, dające natychmiastowy dostęp do informacji i wartości wynikającej z ich przetwarzania, do wykorzystania bezpośrednio w części operacyjnej aplikacji<sup>10</sup>. Biznesowo prowadząc do przewagi konkurencyjnej i innowacji nad rozwiązaniami standardowymi, stosując nowoczesne techniki analityczne (w tym sztuczną inteligencję) na globalną skalę działania systemu (Rys.2).

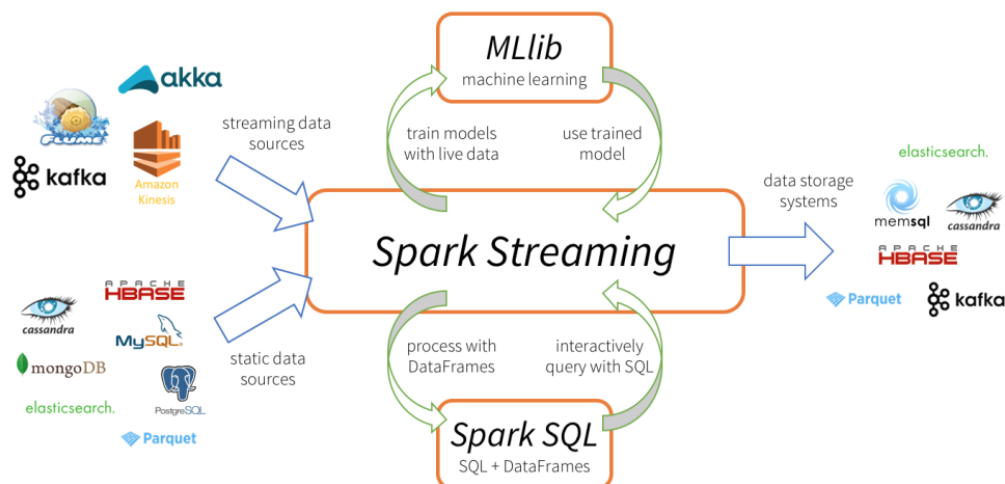
---

<sup>8</sup> *Hadoop vs. Spark: The New Age of Big Data* [online] Ken Hess [dostęp: 27.08.2019]. Dostępny w Internecie: <https://www.datamation.com/data-center/hadoop-vs.-spark-the-new-age-of-big-data.html>

<sup>9</sup> *Apache Spark™ is a unified analytics engine for large-scale data processing*. [online] The Apache Software Foundation. [dostęp: 27.08.2019]. Dostępny w Internecie: <https://spark.apache.org/>

<sup>10</sup> *What Is Fast Data And Why Is It Important?* [online] Lightbend, Inc. [dostęp: 28.08.2019]. Dostępny w Internecie: <https://www.lightbend.com/blog/executive-briefing-what-is-fast-data-and-why-is-it-important>

Rys.2. Komponenty strumieniowego systemu danych Fast Data opartego o Spark Streaming



Źródło: *What Is Spark Streaming?* [online] Databricks Inc. [dostęp: 01.09.2019]. Dostępny w Internecie: <https://databricks.com/glossary/what-is-spark-streaming>

Postęp tej dziedziny technologii wyprzedził pierwotne potrzeby analitycznego przetwarzania danych i zapoczątkował rozwój operacyjnego Big Data – nowej generacji aplikacji, które funkcjonują z powodzeniem w warunkach milionów aktywnych użytkowników, równoczesnych połączeń, wielu asynchronicznych stanów i nieustannie rosnących wolumenów danych.

## 1.2 SQL, NoSQL a NewSQL – wykorzystanie shardingu oraz replikacji

Rewolucja w kierunku Big Data wykazała ułomności obecnych standardów relacyjnych baz danych, służących do przechowywania informacji. Technologie takie jak m. in. MySQL, Oracle czy MS SQL nie zostały opracowane z myślą o skali Internetu i masowego transakcyjnego operowania na danych. Próbuąc sprostać wyzwaniom skalowalności horyzontalnej opracowano partycjonowanie, replikację oraz sharding. Stopniowo wprowadzając je, jako rozwiązania standardowe m. in. w PostgreSQL.

Wraz z rosnącym wolumenem danych bazy relacyjnej spadała jej wydajność, prowadząc do spowalniania odpowiedzi na zapytania odczytu, a także paraliżując operacje zapisu. W skrajnych przypadkach sprawia to, że system jest niezdolny do użytku. Podjęto, więc próby optymalizowania tabel, a gdy te działania okazywały się niewystarczające, migrowano dane tak, aby odciążyć instancję z przeciążenia nadmiarowością informacji. Podejście *partycjonowania* skupiało się na dzieleniu zawartości jednej encji na wiele tabel w ramach jednej instancji bazy danych, umożliwiało to podział jednej dużej kolekcji danych na pomniejsze, zmniejszając narzut na indeksowanie, wykonywanie zapytań oraz zwiększając responsywność. Gdy partycjonowanie w ramach pojedynczej instancji okazywało się niewystarczające, wkroczone w skalowanie horyzontalne: dla odczytu stosowano metodę *replikacji*, która duplikowała te same dane encji na wiele instancji RDBMS, na których aplikacja może wykonywać równoczesne niezależne zapytania; natomiast uzyskanie wydajnego zapisu osiągnano poprzez *sharding*, czyli w uproszczeniu podeście partycjonowania na wielu niezależnych instancjach baz danych.<sup>11</sup> Nowatorskie rozwiązania tamtych czasów zmierzały w kierunku podważania podstawowej cechy baz danych – gwarancji niezawodności transakcyjnej ACID<sup>12</sup>:

- *Atomicity* (ang. atomowość, niepodzielność) – każda transakcja jest pojedynczą jednostką zmiany stanu bazy danych, albo powiedzie się w całości, albo całkowicie się nie powiedzie wycofując wprowadzone dotychczas zmiany
- *Consistency* (ang. spójność) – zmiana stanu bazy danych może prowadzić tylko z jednego prawidłowego stanu do drugiego, eliminując wszelkie niepoprawności integralności danych na etapie transakcji
- *Isolation* (ang. izolacja) – równoczesne wykonywanie transakcji zapewnia taką samą poprawność, jak w przypadku wykonania tych samych transakcji sekwencyjnie, zmiany stanu bazy danych są izolowane od równoczesnych transakcji
- *Durability* (ang. trwałość) – gwarantuje, że raz zatwierdzona transakcja pozostanie w nienaruszonym stanie bazy danych nawet w przypadku awarii systemu

---

<sup>11</sup> *Understanding Database Sharding* [online] DigitalOcean, LLC [dostęp: 03.09.2019]. Dostępny w Internecie: <https://www.digitalocean.com/community/tutorials/understanding-database-sharding>

<sup>12</sup> *Breaking Down ACID Databases* [online] Zack Busch [dostęp: 04.09.2019]. Dostępny w Internecie: <https://learn.g2.com/acid-database>

Uzyskując nowe możliwości architektury bazodanowej w paradygmacie rozproszonym, kosztem standardów bezpieczeństwa danych w RDBMS.

Systemy przetwarzania Big Data wymagały narzędzi bazodanowych, które przewidywalnie skalowały się horyzontalnie, a w nadmiarze danych nie potrzebowały aż tak restrykcyjnych wymogów co do transakcyjności i bezpieczeństwa przechowywanych informacji. Pozwoliło to na odrodzenie technologii nierelacyjnych baz danych NoSQL (zaprzeczenie do standardowego SQL albo również ang. Not Only SQL – nie tylko SQL) w postaci nowoczesnych narzędzi, mogących z powodzeniem sprawdzać się w roli małej bazy na wzór RDBMS jak i z narzutem potężnego ruchu Google’a albo Facebooka zapewniając wymagane skalowanie w ramach rozwiązania. Przedstawiały je bazy danych takie, jak CouchDB, Cassandra, MongoDB czy Neo4J i wiele innych. Na ich potrzeby abstrahując do wcześniej przyjętego ACID opracowano CAP Theorem<sup>13</sup>:

- *Consistency* (ang. spójność) – wszystkie węzły systemu rozproszonej bazy danych dysponują zsynchronizowanymi danymi w tym samym czasie. Przeprowadzenie zapytania odczytu zawsze zwraca najnowszy wynik operacji zapisu. Osiągnięcie spójności systemu wiąże się z zablokowaniem możliwości dostępności danych do czasu, aż przebiegnie kompletna replikacja nowego zapisu na wszystkie węzły sieci
- *Availability* (ang. dostępność) – rozproszona baza danych zawsze wykonuje operację odczytu/zapisu, niezależnie od stanu dowolnego pojedynczego węzła w sieci. Dostępność systemu jest możliwie najwyższa, jednak wiąże się ze zwracaniem potencjalnie nie najnowszych spójnych zmian systemu
- *Partition tolerance* (ang. tolerancja podziału) – system rozproszony funkcjonuje niezależnie od ilości opóźnionych komunikatów, z powodu awarii sieci pomiędzy węzłami. Zapewniając replikację danych pomiędzy kombinacjami wielu węzłów, a w efekcie utrzymując działanie całego systemu nawet w wypadkach wielu lokalnych awarii

Wg którego bazy danych generacji Big Data mogły zapewniać jedynie dwie z trzech cech CAP kosztem wykluczonej. *Partition tolerance* stanowi podstawę tolerancji

---

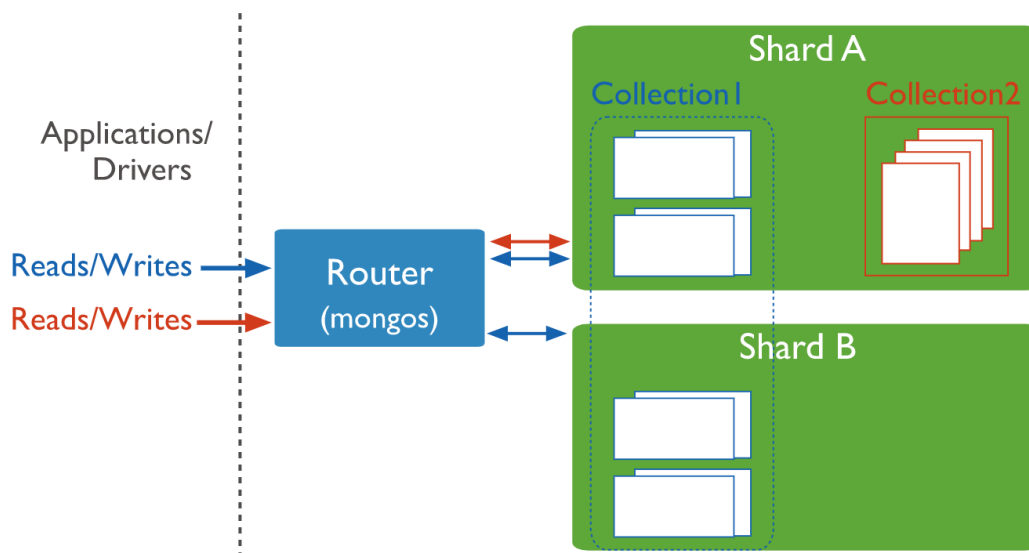
<sup>13</sup> *CAP Theorem and Distributed Database Management Systems* [online] Syed Sadat Nazrul [dostęp: 03.09.2019]. Dostępny w Internecie: <https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e>



na awarię systemu, w praktyce więc zapewnienie tej cechy jest koniecznością, dlatego też decyzja o doborze drugiej cechy dąży w kierunku faworyzowania *Availability* albo *Consistency*. W wielu rzeczywistych przypadkach biznesowych jest to satysfakcjonujący wynik, chociażby na potrzeby platform social media, gdzie bezwzględna spójność prezentowanych danych nie jest kluczowa, gdyż wpisy portali społecznościowych nie są informacją krytyczną tak jak np. w przypadku danych liczbowych rozmaitych operacji giełdowych na dużą skalę.

Bazy nierelacyjne zaprezentowały nowe podejście do przechowywania danych, ich podstawowe formy to bazy: kolumnowe, dokumentowe, par klucz-wartość, a także grafowe. Każda z nich przystosowana do nieco innego wyspecjalizowanego zastosowania, każda natomiast zapewniającą potrzebną skalowalność. Rozwiązania rozproszonego systemu bazy danych oparte są o podejście *shardingu* i *replikacji* na dużą skalę, wbudowane w narzędzie i gotowe do wdrożenia produkcyjnego. Zapewnia to gwarancję oprogramowaniu, że niezależnie od scenariusza rozwoju produktu, jego warstwa bazodanowa jest gotowa na przetwarzanie wolumenów danych Big Data operacyjnie jak i analitycznie, gdy tylko zajdzie taka potrzeba. Jednym z przykładów jest dokumentowa, nierelacyjna baza danych MongoDB, która obsługuje własny język zapytań zbliżony do standardu SQL, zamiast tabel zawiera kolekcje zdenormalizowanych danych w formie BSON, a w miejsce relacji pozwala na ustanowienie referencji między kolekcjami. Dla skalowalności horyzontalnej odczytu umożliwia replikację poprzez *replica-set*, a dla skalowania zapisu mechanizm *sharded cluster* (Rys. 3.).

Rys.3. Operacje zapisu / odczytu w *Sharded cluster* poprzez router *mongos*



Źródło: *Sharding — MongoDB Manual* [online] MongoDB, Inc [dostęp: 04.09.2019]. Dostępny w Internecie: <https://docs.mongodb.com/manual/sharding/>

Od ostatnich kilku lat rozpoczęto innowację w kierunku integracji gwarancji bezpieczeństwa ACID baz relacyjnych SQL razem ze skalowalnością horyzontalną rozproszonych systemów baz danych NoSQL – osiągając idealne rozwiązanie w postaci nazwanej NewSQL. Nowe systemy mogłyby być jednocześnie relacyjne, znormalizowane i łatwo skalowalne w ramach rozwiązania. Przykładami są np. VoltDB czy CockroachDB opracowany w języku Golang. Zapewnienie większego bezpieczeństwa operacyjnego Big Data pozwoli odważyć się na budowanie oprogramowania dużej skali w scenariuszach e-commerce wymagających wysokiej spójności.<sup>14</sup>

### 1.3 Programowanie konkurencyjne, nieblokujące i asynchroniczne, wykorzystanie paralelizacji

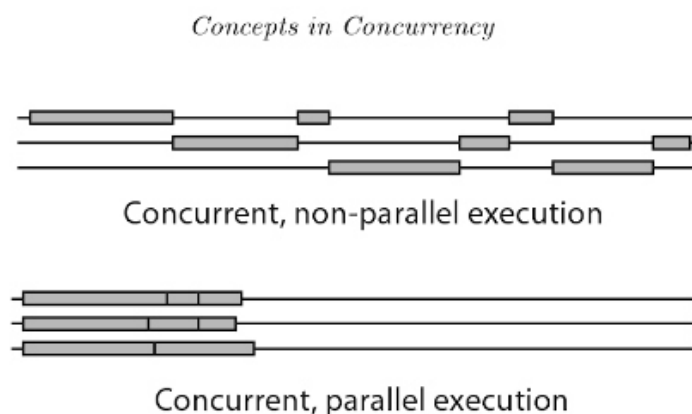
Sekwencyjność jest prymitywna i prosta, do tego stopnia, że trudno sobie wyobrazić współczesny komputer bez konkurencyjności (przetwarzania współbieżnego) – bez niej nie bylibyśmy w stanie zrobić na maszynie więcej, niż jednej rzeczy, nie dość więc, że wykonywanie kilku naturalnych czynności, jak jednocześnie słuchanie muzyki, przeglądanie Internetu, zgrywanie danych w tle byłoby niemożliwe, to

<sup>14</sup> *What Is New About NewSQL?* [online] Gokhan Simsek [dostęp: 04.09.2019]. Dostępny w Internecie: <https://softwareengineeringdaily.com/2019/02/24/what-is-new-about-newsq/>

prawdopodobnie taka maszyna nigdy by nie powstała, bo już na warstwie systemu operacyjnego konkurencyjność jest niezbędna dla jego prawidłowego działania.

Podobna przyszłość czekała oprogramowanie, czy to wydzielenie części intensywnych obliczeń na osobny wątek lub proces, czy nieblokowanie się interfejsu, kiedy odbywały się dwie niezależne operacje np. klikalność kontrolek w czasie ładowania zawartości. Jeszcze większe znaczenie konkurencyjność miała dla Big Data i obliczeń rozproszonych, które przy skalowalności horyzontalnej klastra wielu maszyn, wymagały daleko idącej paralelizacji (obliczeń równoległych) przetwarzanych zadań, dla uzyskania wysokiej wydajności. Zależność pomiędzy konkurencyjnością, a paralelizacją przedstawiono na Rys.4.

Rys.4. Konkurencyjność osiągnięta samodzielnie oraz wraz z paralelizacją



Źródło: *What is the difference between concurrency and parallelism?* [online] Apurva Thorat [dostęp: 07.09.2019]. Dostępny w Internecie: <https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism>

Konkurencyjność oznacza, że oprogramowanie czyni postępy w więcej, niż jednym zadaniu jednocześnie. Paralelizacja natomiast dzieli swoje zadania na mniejsze podzadania, które są wykonywane na procesorze jednocześnie w tym samym czasie.<sup>15</sup> Przy konkurencyjności bez paralelizacji aplikacja nie może wykonać więcej niż jednej rzeczy w tym samym czasie, dlatego zaczyna wiele zadań, nie kończy ich, i rozpoczyna kolejne, wracając jedynie do poprzednich, gdy bieżące zadanie zostanie zablokowane.

<sup>15</sup> *Don't be confused between Concurrency and Parallelism* [online] Skrew Everything [dostęp: 07.09.2019]. Dostępny w Internecie: <https://medium.com/from-the-scratch/dont-be-confused-between-concurrency-and-parallelism-eac8e703943a>

Mechanizm ten nazywany jest *context switching*, umożliwia przełączanie się pomiędzy niezakończonymi zadaniami. Istotnym również jest, że paralelizm oznacza zapewnioną konkurencyjność, natomiast konkurencyjność nie gwarantuje paralelizacji.

Oba podejścia przeplatają się między sobą i mają odpowiednie dla siebie szczególne zastosowania. Związane są z nimi pojęcia *synchroniczności*, gdy program wykonuje operacje zależnie od siebie oraz *asynchroniczności*, kiedy operacje są oddelegowywane i niezwiązane ze sobą (niesynchronizowane). Wraz z nimi rozważa się *blokowanie*, gdy główny wątek wykonując operacje musi zostać wstrzymany do czasu wykonania innego zadania oraz przetwarzanie *nieblokujące*, gdy wykonywane operacje nie powstrzymują wątku głównego od wykonywania następnych czynności.<sup>16</sup> Zazwyczaj wspólnie postrzega się asynchroniczność wraz z nieblokowaniem w przeciwieństwie do synchroniczności z blokowaniem, nie wyklucza to jednak pozostałych kombinacji tych pojęć.

Istnieją scenariusze, gdzie preferowana jest paralelizacja – wykorzystanie możliwości wielordzeniowych procesora do równoczesnego wykonania operacji na wątkach – są to problemy tzw. *CPU-bound* (związane z procesorem), np. wykonywanie algorytmu obliczeniowego, generowanie hashu *SHA-1 checksum* dla plików itp. We współczesnym oprogramowaniu o wiele częstszym jednak jest inny problem, mianowicie współpraca z dyskiem, bazą danych czy połączeniem sieciowym – ten szereg problemów określono, jako *IO-bound* (związane z systemami wejścia / wyjścia). Oczekiwanie na zakończenie odczytu pliku z dysku, zwrócenie rekordu bazy danych albo odpowiedź na zapytanie do strony internetowej wiąże się z czasem oczekiwania, w efekcie, więc z blokadą, zastosowanie w tym wypadku paralelizacji nie usprawni mechanizmu, gdyż niezależnie od ilości wątków, każdy z nich zostanie zablokowany i nie przyspieszy czasu odpowiedzi ze źródeł, na które oczekuje. Natomiast jest to idealny scenariusz na zastosowanie konkurencyjności, bez paralelizacji – w miejsce operacji wstrzymującej się na blokadzie następuje przełączenie na następne zadanie bez tracenia czasu i mocy obliczeniowej na zadanie, które oczekuje na odpowiedź; jak tylko odpowiedź zostanie otrzymana, blokada zniknie, a oprogramowanie powraca do

---

<sup>16</sup> *Async and non-blocking concepts in Java. Is it possible to be asynchronous and blocking together?* [online] Alexandre [dostęp: 07.09.2019]. Dostępny w Internecie: <https://stackoverflow.com/questions/25316798/async-and-non-blocking-concepts-in-java-is-it-possible-to-be-asynchronous-and-b>

przetwarzania dalszych poleceń operacji.<sup>17</sup> Dzięki temu uzyskuje się wyższą wydajność, z miejsc, gdzie oprogramowanie ma przestoje na blokady i w innym wypadku nie mogłoby skorzystać z zablokowanych zasobów. Model ten oryginalnie powstał na potrzeby JavaScript, rozwinięty, usprawniony oraz spopularyzowany w *asynchronicznym, nieblokującym* środowisku NodeJS oparty o implementację *Event Loop*, jednowątkowym rozwiązaniu konkurencyjnym.

Rozproszone systemy Big Data potrzebują konkurencyjności, aby wykonywać wiele zadań na poszczególnych maszynach w klastrach, jednocześnie wymagają również paralelizacji, dla zadań związanych z CPU, takich, jak operacje przetwarzania danych. Muszą także posiadać cechy asynchroniczności nieblokującej, aby uzyskać możliwie najwyższą wydajność z przeprowadzanych operacji bez przestojów. Dotychczasowy standard konkurencyjności oparty był o model współdzielonej pamięci, w którym startowano wiele wątków ściśle związanych ze stanem aktualnej aplikacji, wymagało to ręcznego blokowania dostępu wątków do stanu poprzez wykorzystanie synchronizacji zmiennych i metod, stosowania blokad, semaforów, upewniania się o wykorzystaniu odpowiednich wersji struktur danych przystosowanych do wielowątkowości, a także samodzielne dbanie o niezmiennność (ang. *immutability*) zmiennych wykorzystywanych przez wiele wątków. Dodatkowo kod był podatny na nowe rodzaje błędów m. in. *race condition* czy *deadlocks*. Ostatecznie po zastosowaniu tych zabiegów w dużej skali kod był tak bardzo wypełniony blokadami, że zysk płynący z paralelizacji był marginalny, a utrzymanie tak złożonej struktury bardzo utrudnione.

Razem z rozwojem zapotrzebowania na systemy rozproszone (nie tylko Big Data) powrócono do innego podejścia określanego, jako *message passing*, z sukcesem zaimplementowanego, jako główna funkcjonalność języka Golang, wykorzystująca mechanizm *coroutines* dla konkurencyjności, nazywany w tym języku goroutines. Zgodnie z motywacją zastosowania tej technologii w języku: „*Do not communicate by sharing memory; instead, share memory by communicating.*” (ang. „Nie komunikuj się, dzieląc pamięć; zamiast tego współdziel pamięć, komunikując się.”) innowacja wyszła naprzeciw potrzebom, rezygnując z podatnego na błędy modelu konkurencyjności

---

<sup>17</sup> *I/O-bound vs CPU-bound in Node.js* [online] Panu Pitkamaki [dostęp: 07.09.2019]. Dostępny w Internecie: <https://bytearcher.com/articles/io-vs-cpu-bound/>

współdzielonej pamięci na korzyść innego, modelu komunikacji poprzez przekazywanie wiadomości (Listing nr 1.).

Listing nr 1. Konkurencyjność modelu *message passing* w Golang

```
package main
// ...
func main() {
    // Tablica URL do wywołania konkurencyjnego
    urls := []string{
        "https://www.easyjet.com/",
        "https://www.skyscanner.de/",
        "https://www.ryanair.com",
    }

    // Inicjalizacja kanału komunikacji
    c := make(chan urlStatus)
    for _, url := range urls {
        // Wywołanie funkcji w mechanizmie goroutine
        go checkUrl(url, c)
    }

    // Przetwarzanie konkurencyjne, nieblokująco i asynchronicznie
    // Operowanie na pamięci otrzymanej z kanału
    result := make([]urlStatus, len(urls))
    for i, _ := range result {
        result[i] = <-c
        if result[i].status {
            fmt.Println(result[i].url, "is up.")
        } else {
            fmt.Println(result[i].url, "is down !!")
        }
    }
}

// Wykonanie zapytania i przekazanie stanu pamięci poprzez kanał
func checkUrl(url string, c chan urlStatus) {
    _, err := http.Get(url)
    if err != nil {
        // The website is down
        c <- urlStatus{url, false}
    } else {
        // The website is up
        c <- urlStatus{url, true}
    }
}
```

// ...

Źródło: *Asynchronous programming with Go* [online] Gaurav Singha Roy [dostęp: 08.09.2019].  
Dostępny w Internecie: <https://medium.com/@gauravsingharoy/asynchronous-programming-with-go-546b96cd50c1>

Istnieje wiele różnorodnych modeli konkurencyjności, w mniejszym lub większym stopniu wykorzystujących paralelizację – kluczowym jest, aby dla systemów nowej generacji były one *asynchroniczne* oraz *nieblokujące*, tak żeby przewidywalnie się skalowały oraz wykorzystywały maksimum dostępnych zasobów. Postęp w dziedzinie przetwarzania rozproszonego faworyzuje podejście konkurencyjności, w praktyce jednak nowoczesny system potrzebuje wielu z jej aspektów do kompletnego działania.

#### **1.4 Systemy reaktywne z architekturą mikroserwisową w chmurze**

Oprogramowanie dotychczas opierało się na architekturze monolitu, gdzie niezależnie od skali działania, czy to mała strona internetowa, czy potężna korporacyjna aplikacja, program był zamkniętą strukturą w ramach jednej technologii oprogramowania, łączący się z jedną bazą danych i działający na jednym serwerze. Wraz z rozwojem Internetu podejście monolityczne okazało się zbyt limitujące i niewystarczające dla uzyskania wymaganej skalowalności.

Aby przezwyciężyć limitacje monolitu zaproponowano nową architekturę, będącą rozwinięciem SOA (ang. *Service Oriented Architecture* – Architektura zorientowana na usługi), która miała pozwalać na skalowalność istniejącego przestarzałego kodu i natywność do przetwarzania w chmurze, tak, aby sprostać rosnącym wymaganiom doby Internetu. Tak powstała architektura mikroserwisowa – struktura oprogramowania składająca się ze zbioru usług, będących: wysoce utrzymywalne i testowalne, luźno powiązane, niezależnie wdrażane, zorganizowane wokół biznesu i zarządzane przez małe zespoły. Umożliwia to szybkie, częste i niezawodne dostarczanie dużych, złożonych aplikacji, a także pozwala na zastosowanie wielu niezależnych technologii.<sup>18</sup> Oprogramowanie zostało podzielone na kolekcję mikroservisów, które komunikują się

---

<sup>18</sup> *What are microservices?* [online] Chris Richardson, Microservices.io [dostęp: 08.09.2019].  
Dostępny w Internecie: <https://microservices.io/>

między sobą zamiast być wspólnie zaimplementowane w jednej bazie kodu, nadal jednak prezentując użytkownikowi integralne rozwiązanie tak jak miało to miejsce w monolicie. Dzięki temu poszczególne elementy domenowe aplikacji, takie jak np. katalog produktów i obsługa płatności mogą być rozwijane od siebie niezależnie (nawet w zupełnie różnych technologiach) i zyskują możliwość do wdrożenia w chmurze. Umożliwia to skalowalność oprogramowania per mikroserwis, zastosowanie *konteneryzacji*, *load balancingu*, monitorowania i reagowania na szczyty wykorzystania (*elastyczność*). Gdy w aplikacji generowany jest nadzwyczajny ruch (np. nadszedł dzień promocji Black Friday lub wychodzi nowy sezon popularnego serialu), mikroserwis odpowiedzialny za obsłużenie tego ruchu (produkty, płatności czy streamowanie wideo) może zostać odpowiednio zeskalowany, jeżeli tylko jego pojedyncza instancja nie jest dostatecznie wydajna. W takim wypadku infrastruktura chmurowa może utworzyć nowe instancje danego mikroserwisu, które wspólnie (konkurencyjnie) będą wykonywały operacje równo między sobą dzięki zastosowaniu *load balancingu*, wytrzymując nadzwyczajny ruch i gwarantując działanie całemu oprogramowaniu. Architektura mikroserwisowa pozwoliła na dalszy rozwój dużych aplikacji w kierunku systemu rozproszonego przetwarzania osiągniętemu dzięki wirtualizacji w chmurze.

Tworzenie systemów przetwarzania rozproszonego (ang. *distributed computing*) jest zadaniem nietrywialnym, często obciążonym wieloma błędami, które są kosztowne dla biznesu. Architektura mikroserwisowa zbliża oprogramowanie do przetwarzania rozproszonego, nie wyczerpuje jednak wszystkich jego aspektów zawartych w *Reactive Manifesto*, definiujący nowoczesne systemy rozproszone jako posiadające cechy<sup>19</sup>:

- *Responsive* (ang. responsywny) – system reaguje w odpowiednim czasie, zapewniając użyteczność i przewidywalność działania. Gwarantuje wczesne wykrywanie błędów i ich efektywne obsługiwane. Koncentruje się na szybkim i spójnym czasie reakcji, stanowiąc o niezawodności
- *Resilient* (ang. odporny, wytrzymały) – system pozostaje responsywny w wypadku każdej awarii, osiągane jest to dzięki replikacji, izolacji i delegowaniu jego elementów. Awaria odnotowana w jednym z komponentów dotyczy tylko niego i nie przerywa działania systemu, pozwalając niemu na przetworzenie błędów i zregenerowanie się, a

---

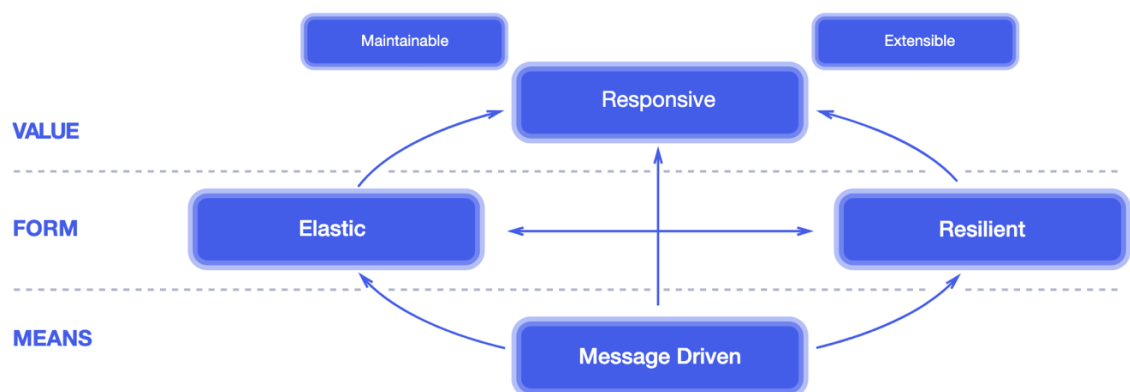
<sup>19</sup> *The Reactive Manifesto* [online] [reactivemanifesto.org](https://www.reactivemanifesto.org) [dostęp: 08.09.2019]. Dostępny w Internecie: <https://www.reactivemanifesto.org/>



następnie powrót do działania w systemie (podejście ang. *let it crash* – przyzwól na awarię; przywróć do działania)

- *Elastic* (ang. elastyczny) – system nie tylko skaluje się, ale również pozostaje responsywny w warunkach zmiennego obciążenia. Potrafi autonomicznie reagować na sytuacje zwiększonego i zmniejszonego ruchu, inteligentnie alokując zasoby. Architektura systemu nie pozwala na powstanie wąskich gardeł ani punktów krytycznych, dynamicznie dzieli lub replikuje komponenty, aby przetwarzały wspólnie dane wejściowe
- *Message Driven* (ang. sterowany wiadomościami) – system opiera się o *asynchroniczne przekazywanie wiadomości* (ang. asynchronous message-passing) pomiędzy komponentami zapewniając luźne powiązania pomiędzy nimi oraz izolację. Wiąże się to również z delegowaniem błędów, jako wiadomości w systemie. Komunikacja pomiędzy elementami jest nieblokująca w oczekiwaniu na zasoby, a przetwarzanie wiadomości umożliwia zastosowanie *back-pressure*, gdy komponent jest przeciążony

Rys.5. Cechy systemu rozproszonego *Reactive System*



Źródło: *The Reactive Manifesto* [online] [reactivemanifesto.org](http://reactivemanifesto.org) [dostęp: 08.09.2019]. Dostępny w Internecie: <https://www.reactivemanifesto.org/>

Tak zdefiniowany system nazywamy systemem reaktywnym (ang. *Reactive System*) – „*The largest systems in the world rely upon architectures based on these properties and serve the needs of billions of people daily.*” (ang. „Największe systemy na świecie opierają się na architekturze opartej na tych właściwościach i służą codziennie miliardom ludzi.” wg Reactive Manifesto). Głównym motorem napędowym za

nowoczesnymi systemami jest responsywność, jeżeli użytkownik nie otrzyma wartości w odpowiednim czasie, porzuci rozwiązanie biznesowe; nie różni to się niczym od nie otrzymania wartości wcale, a prowadzi do spadku zysków i udziału w rynku. Aby uzyskać responsywność potrzebna jest operacyjność w trakcie awarii (*resilient*) oraz operacyjność przy obciążeniu (elastyczność), aby to osiągnąć system musi być *message-driven* (Rys.5)<sup>20</sup>.

Niezależnie od systemów reaktywnych na popularności zyskało programowanie reaktywne (ang. *Reactive Programming*) – jest to forma programowania asynchronicznego, polegająca na deklaratywnym przetwarzaniu asynchronicznych strumieni danych i propagacji zmiany nieblokująco z *back-pressure*. Terminem *back-pressure* określa się reagowanie elementów przetwarzania na sytuację, w której, jeden z jej elementów nie jest w stanie przetworzyć nadmiaru operacji. Jeżeli dany komponent jest przeciążony, zapełnianie go kolejnymi operacjami do wykonania może spowodować awarię przetwarzania strumienia – *back-pressure* zapewnia informację zwrotną, że należy wstrzymać się z przyjmowaniem nowych danych, aby kolejne komponenty były w stanie nadążyć z przetwarzaniem. W kierunku standaryzacji programowania reaktywnego strumieni danych utworzono inicjatywę *Reactive Streams* – „an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure” (ang. „inicjatywa dla zapewnienia standardu asynchronicznego przetwarzania strumieni danych z nieblokującym *back pressure*”).<sup>21</sup>

W kontekście architektury mikroserwisowej programowanie reaktywne jest stosowane w pojedynczym mikroserwisie, dla uzyskania asynchronicznej i nieblokującej konkurencyjności wewnątrz serwisu. Natomiast model systemu reaktywnego stosuje się pomiędzy wieloma mikroserwisami w celu opracowania niezawodnego systemu rozproszonego. Nie są to paradygmaty od siebie zależne, jednakże mogą zyskać na wspólnym zastosowaniu. Istotna jest między nimi odpowiednia komunikacja o charakterze reaktywnym, nieblokującym i dwukierunkowym.

---

<sup>20</sup> *Reactive Programming versus Reactive Systems* [online] Lightbend, Inc. [dostęp: 09.09.2019]. Dostępny w Internecie: <https://www.lightbend.com/white-papers-and-reports/reactive-programming-versus-reactive-systems>

<sup>21</sup> *Reactive Streams* [online] reactive-streams.org [dostęp: 09.09.2019]. Dostępny w Internecie: <https://www.reactive-streams.org/>

Rewolucja systemów reaktywnych wyznacza kierunek branży w dążeniu do obsłużenia coraz więcej użytkowników, coraz więcej danych i coraz więcej operacji, łącząc w sobie operacyjne Big Data i zapewniając potrzebne narzędzia do przetwarzania Fast Data za pomocą reaktywnych strumieni danych ze skalowalnością horyzontalną w systemie rozproszonym. Platforma Lightbend, propagatorzy podejścia reaktywnego i strumieniowego, mogą poszczycić się implementacjami dla największych globalnych platform, m. in.: Wix – „*From 5 To 50 Million Users And Beyond: Why Wix Went Reactive*” (ang. “Od 5 do 50 milionów użytkowników i więcej: dlaczego Wix stał się reaktywny”), Groupon – „*How Groupon Scales Personalized Offers To 48 Million Customers On Time*” (ang. “Jak Groupon skaluje spersonalizowane oferty do 48 milionów klientów na czas”) czy PayPal – “*PayPal Blows Past 1 Billion Transactions Per Day Using Just 8 VMs With Akka, Scala, Kafka and Akka Streams*” (ang. “PayPal osiąga ponad 1 miliard transakcji dziennie za pomocą zaledwie 8 maszyn wirtualnych z Akka, Scala, Kafka i Akka Streams”).<sup>22</sup> Systemy reaktywne to przyszłość oprogramowania dużej skali, dla największych inicjatyw naszych czasów.

### 1.5 Actor model – konkurencyjne przetwarzanie rozproszone

Biblioteki i platformy opracowane dla systemów reaktywnych (takie jak projekt *Akka* czy platforma *Erlang*) stosunkowo nie są rozwiązaniem nowym – dzisiejsze systemy rozproszone są efektem dziesięcioleci doświadczeń w tworzeniu wydajnych i niezawodnych wdrożeń na skalę globalną. Ich początki sięgają lat 80-tych, kiedy rozpowszechniono model przetwarzania konkurencyjnego *actor model*.

*Actor model* adresuje problemy obecnie ustandaryzowanych technologii, takich jak Java czy C#, które bazują na przestarzałych paradygmatach lat 80-tych i 90-tych w programowaniu konkurencyjnym. W czasach, kiedy były to przełomowe rozwiązania wprowadzane na rynek jako innowacje, rozproszone systemy były rzadkością, a na potrzeby wytwarzanych aplikacji, wielowątkowy model współdzielonej pamięci był

---

<sup>22</sup> *Customer Case Studies* [online] Lightbend, Inc. [dostęp: 09.09.2019]. Dostępny w Internecie: <https://www.lightbend.com/case-studies>

wystarczający w inżynierii oprogramowania. Krytyka tradycyjnych języków programowania obiektowego obejmowała m. in.:

- *Wyzwanie enkapsulacji* – obiekty mogą mieć gwarancję enkapsulacji (ochronę przed niepoprawnymi stanami) jedynie w jednowątkowym modelu wykonywania, wielowątkowe przetwarzanie niemal zawsze prowadzi do zniekształcenia wewnętrznego stanu. Blokady nałożone na współdzielone obszary pomiędzy wątkami są próbą zachowania enkapsulacji dla obiektów, w praktyce prowadzą do skrajnie niewydajnych implementacji i logiki podatnej na błędy (*deadlocks*, *race condition*). Blokady funkcjonują lokalnie, a możliwości ich rozproszenia są ograniczone i nieskalowalne.
- *Iluzja współdzielonej pamięci* – współczesne procesory nie przechowują zmiennych w pamięci, lecz w tzw. *cache lines*, które są lokalne dla rdzenia CPU, w efekcie, aby wymieniać pamięć pomiędzy wątkami istnieje mechanizm przekazywania wiadomości na poziomie procesora. Aby wiele wątków mogło współpracować z współdzielonym stanem potrzebne jest przekazywanie pamięci pomiędzy *cache lines*, co jest operacją niewydajną. Zamiast ukrywania mechanizmu *message passing* za zmiennymi oznaczonymi jako współdzielone, cały model komunikuje się wiadomościami na wyższym poziomie abstrakcji zachowując zmienne lokalne niezależne od siebie na poziomie pojedynczego aktora.
- *Iluzja stosu wywołań (ang. call stack)* – obsługiwanie błędów pomiędzy wątkami prowadzi często do utraty stosu wywołań, np. gdy główny wątek deleguje operacje na wiele wątków pobocznych, błąd na takim pojedynczym wątku nie jest bezpośrednio przekazywany do głównego wątku, ostatecznie kończąc się jego utratą. Systemy konkurencyjne delegujące zadania muszą odpowiednio reagować na błędy i potrafić powrócić z nich do działania. Błędy funkcjonowania stają się więc częścią modelu architektonicznego systemu.

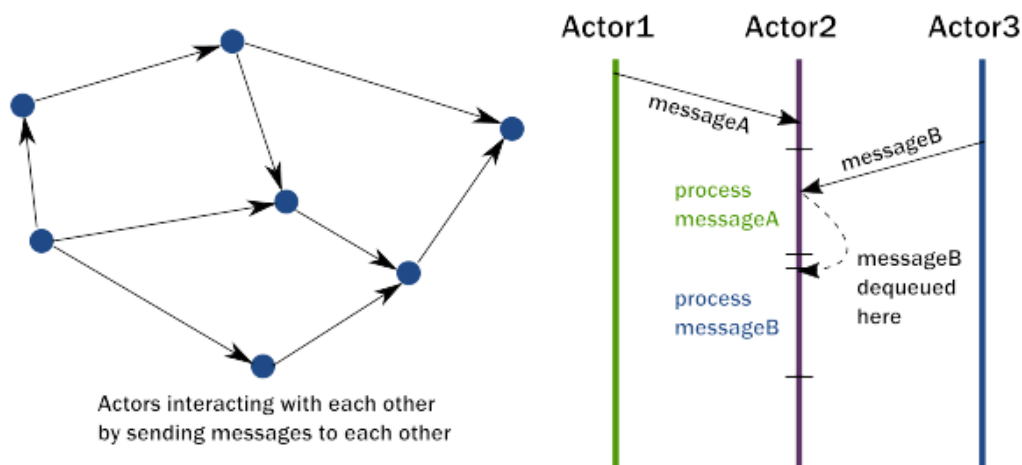
Aby umożliwić rozwój nowoczesnych systemów potrzebny był nowy paradygmat, pozwalający na konkurencyjne przetwarzanie w wysoce wydajnej sieci rozproszonych maszyn.<sup>23</sup>

---

<sup>23</sup> *Why modern systems need a new programming model* [online] Akka [dostęp: 13.09.2019].  
Dostępny w Internecie: <https://doc.akka.io/docs/akka/current/guide/actors-motivation.html>

W *Actor model* wykonywanie oparte jest na aktorach, będących podstawową jednostką przetwarzania konkurencyjnego. Posiadają tzw. *mailbox*, który przyjmuje wiadomości do wykonania z mechanizmu *message passing*. Ma to charakter kolejki, która przechowuje kolejne zadania otrzymane od wysyłających aktorów, następnie planuje ich wykonanie w lokalnym środowisku aktora. Aktorzy wysyłający wiadomości nie oczekują na otrzymanie odpowiedzi zwrotnej, kontynuują więc swoją pracę. Gdy jedno z oddelegowanych zadań zostanie wykonane przez aktora otrzymującego, w odpowiedzi odeśle on nową wiadomość do aktora, który wysłał oryginalne zadanie. Co istotne dzięki temu komunikacja odbywa się nieblokująco i asynchronicznie, zapewniając modelowi pełną konkurencyjność z maksymalną wydajnością na wszystkich rdzeniach procesora (Rys.6.)<sup>24</sup>.

Rys.6. Funkcjonowanie aktorów oraz *message passing* w *Actor model*



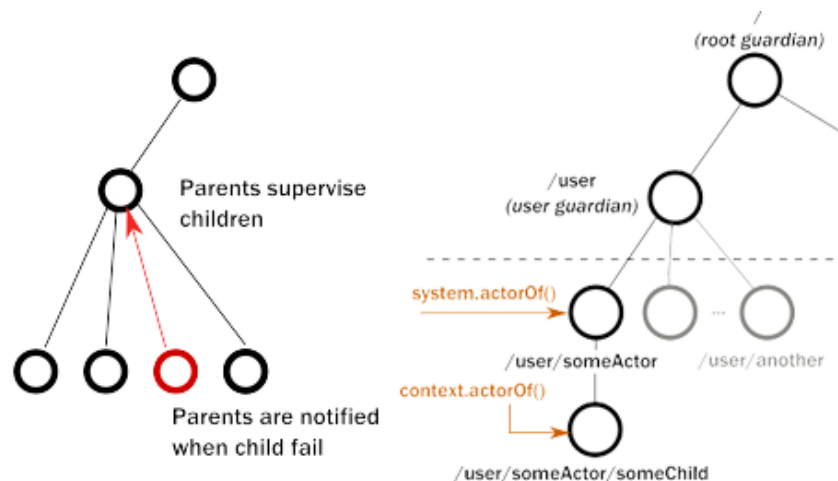
Źródło: *How the Actor Model Meets the Needs of Modern, Distributed Systems* [online] Akka [dostęp: 13.09.2019]. Dostępny w Internecie: <https://doc.akka.io/docs/akka/current/guide/actors-intro.html?language=scala>

Systemy rozproszone opierają się na podejściu przyzwalającym na błędy poszczególnych elementów, pozwalają się im odrodzić i nie wpływają na działanie całego systemu. *Actor model* zawiera w swojej architekturze mechanizmy kontroli błędów. Błędy domenowe związane z logiką aplikacji (np. walidacja danych wprowadzonych przez użytkownika) są odsyłane przez aktora otrzymującego z powrotem do aktora, który wysłał wiadomość z zadaniem. Natomiast w sytuacji awarii

<sup>24</sup> *How the Actor Model Meets the Needs of Modern, Distributed Systems* [online] Akka [dostęp: 13.09.2019]. Dostępny w Internecie: <https://doc.akka.io/docs/akka/current/guide/actors-intro.html?language=scala>

wewnętrznej aktora bezpieczeństwo systemu rozproszonego zapewnia mechanizm nadzoru (ang. *supervision*). Serwisy aktorów są zhierarchizowane w strukturę drzewa (Rys.7.), gdzie każdy nadrzędny aktor stanowi nadzór nad swoim dzieckiem. W wypadku awarii nadzorowanego aktora, jego nadzorca decyduje o jego restarcie, bez przerywania pracy pozostałych aktorów, pozwalając systemowi zreanimować się i dalej funkcjonować tak, jakby sytuacja awaryjna nie miała miejsca.

Rys.7. Mechanizm nadzoru oraz struktura hierarchiczna drzewa aktorów



Źródło: *How the Actor Model Meets the Needs of Modern, Distributed Systems* [online] Akka [dostęp: 13.09.2019]. Dostępny w Internecie: <https://doc.akka.io/docs/akka/current/guide/actors-intro.html?language=scala>

*Actor model* eliminuje problemy wynikające z modelu współdzielonej pamięci – zapewnia wymaganą enkapsulację stanów aktorów, przekazuje stan pamięci dzięki wiadomościom oraz bezpiecznie zarządza błędami dla całości systemu. Zastosowanie konkurencyjności *actor model* pozwala na uzyskanie pełnej wydajności nowoczesnych wielordzeniowych procesorów w sposób nieblokujący i asynchroniczny. Niezależnie, czy system funkcjonuje na jednej lokalnej maszynie, czy na klastrze tysięcy maszyn, *actor model* zapewnia niezawodne działanie systemu rozproszonego w dużej skali, z zachowaną skalowalnością horyzontalną.

Model konkurencyjności w języku Golang wywodzi się z CSP (ang. *Communicating Sequential Processes*), który pomimo funkcjonowania na bazie *message passing*, jest bardziej prymitywny i prosty w zastosowaniu. Wiążą się z tym pewne ograniczenia, które powodują, że Golang nie jest stosownym wyborem do

budowy złożonych systemów rozproszonych w sposób, jaki pozwala na to Akka lub Erlang. W przeciwieństwie do aktorów, komunikowanie się poprzez kanały w *goroutines* jest asynchroniczne, lecz nie w pełni nieblokujące – kanały oczekują na otrzymanie/zapisanie wiadomości do przetworzenia. Język zapewnia możliwość stosowania blokad *Mutex* / *WaitGroup*, które nieumiejętnie zastosowane zamiast kanałów *channel*, powodują przestoje w wykonywaniu konkurencyjnym. Ostatecznie kod asynchroniczny może prowadzić do *deadlocks* oraz *race conditions*, do pewnego stopnia wcześniej wykrywanych przez kompilator. Jednakże największą przeszkodą modelu jest brak możliwości wykorzystania poza pojedynczą maszyną – oznacza to, że niemożliwym jest skalowanie horyzontalne aplikacji w klaster wielu maszyn wykonujących przetwarzanie rozproszone, w związku z tym więc Golang nie obsługuje odpowiedniej komunikacji pomiędzy instancjami, ani bezpiecznego zarządzania błędami oraz awariami. Język ten natomiast zyskał miano „mikroserwisowego”, ze względu na jego szerokie zastosowania w architekturze mikroserwisowej, gdzie jego limitacje architektury są ograniczane na poziomie pojedynczego mikroserwisu, a ich zorkiestrowana kolekcja pozwala na wysoką skalowalność i dostępność aplikacji.<sup>25</sup>

Pierwszą implementacją *actor model* będącą globalnym sukcesem stała się platforma Erlang/OTP (ang. *Open Telecom Platform*) – język programowania wykorzystywany do budowy wielce skalowalnych systemów czasu rzeczywistego z wymaganiami wysokiej dostępności; wbudowaną konkurencyjnością, rozproszeniem oraz tolerancją błędów.<sup>26</sup> Technologia wywodzi się od dostawcy telekomunikacyjnego *Ericsson*, z przeznaczeniem dla wykorzystania w globalnych sieciach telefonicznych GPRS/3G oraz LTE. Erlang/OTP zapewnia nieprzerwane działanie systemów na poziomie *nine „9”s* (99.9999999% nieprzerwanego czasu działania) i jest wykorzystywany wszędzie, gdzie potrzebna jest masowa komunikacja wielu milionów operacji jednocześnie. Technologia ta jest powszechna w sektorze telekomunikacyjnym m. in. przez Nortel oraz T-Mobile, a także w bankowości, e-commerce, czy komunikatorach (aplikacja WhatsApp funkcjonuje dzięki systemowi rozproszonemu zbudowanemu na Erlang) i prawdopodobnie bez niej nie istniałaby współczesna

---

<sup>25</sup> *How are Akka actors different from Go channels? How are two related to each other?* [online] Rick Beton [dostęp: 25.09.2019]. Dostępny w Internecie: <https://www.quora.com/How-are-Akka-actors-different-from-Go-channels-How-are-two-related-to-each-other>

<sup>26</sup> *Erlang programming language* [online] Erlang [dostęp: 25.09.2019]. Dostępny w Internecie: <https://www.erlang.org/>

telefonii w formie jaką znamy dzisiaj. Twórca języka Joe Armstrong w jednym ze swoich wywiadów stwierdził, że: „*If Java is 'write once, run anywhere', then Erlang is 'write once, run forever'*” (ang. „Jeśli Java jest ‘napisz raz, wykonaj wszędzie’, to Erlang jest ‘napisz raz, działaj wiecznie’”). Społeczność IT coraz częściej interesuje się tą technologią ze względu na potrzebę powstawania usług konkurencyjnych, powstał nawet nowy język programowania *Elixir*, który wykorzystuje maszynę wirtualną Erlang i zapewnia zalety platformy Erlang/OTP w nowoczesnym wydaniu, w dużej mierze pod wpływem społeczności języka *Ruby*.

Dla systemów rozproszonych *actor model* naturalnie spełnia wymagania nowoczesnego systemu reaktywnego. Również w scenariuszu fast data, dla strumieniowych systemów danych powstały implementacje takie jak *Akka Streams*, które wewnętrznie wykorzystują aktorów do obsługi asynchronicznych strumieni danych.<sup>27</sup> Modele te więc wpasowują się w przetwarzanie danych wysokiej prędkości Fast Data w sposób rozproszony i analizy informacji w czasie rzeczywistym. Innym powszechnie wykorzystywanym narzędziem strumieniowania jest *Apache Spark*, ten natomiast wywodzi się z zastosowania analitycznego.

## 1.6 Sztuczna inteligencja – machine learning oraz deep learning

Systemy strumieniowe danych Fast Data operują na ogromnych wolumenach danych przekazywanych w czasie rzeczywistym, szeroko rozumiane algorytmy sztucznej inteligencji potrzebują ich do wypracowywania inteligentnych zachowań. Połączenie operacyjnej aplikacji big data z modelami *machine learning* pozwoli na zastosowanie wyników algorytmów sztucznej inteligencji ze środowiska analitycznego, bezpośrednio do warunków operacyjnych, wzbogacając aplikacje o inteligentne zachowania już podczas korzystania z oprogramowania.

AI (ang. *Artificial intelligence*) czyli sztuczna inteligencja to dziedzina informatyki zajmująca się symulacją *inteligentnych zachowań* w komputerach; nadawaniem zdolności do naśladowania inteligentnych ludzkich zachowań; umożliwieniem

---

<sup>27</sup> *Akka Streams Introduction* [online] Akka [dostęp: 25.09.2019]. Dostępny w Internecie: <https://doc.akka.io/docs/akka/current/stream/stream-introduction.html>



przeprowadzenia zadań, które zazwyczaj możliwe są do wykonania wyłącznie przez ludzi (np. percepcja wizualna, rozpoznawanie mowy, podejmowanie decyzji, tłumaczenia języków itp.).<sup>28</sup> Stanowi zbiorczą kategorię wszelkich inicjatyw, tworzących maszyny zachowujące się *inteligentnie*, takie jak *machine learning* i *deep learning*. AI może być zwyczajnym kodem instrukcji warunkowych *if/then* zaprogramowanym przez człowieka (bez zdolności nauki) albo złożonym modelem statystycznym determinującym wynik na bazie prawdopodobieństwa, oba jednak imitujące zachowanie inteligentne, reagujące odpowiednio do swojego środowiska na bazie postrzegania (kognitywistyka) i przeszłych doświadczeń. W skład takiego podejścia wchodzi tzw. *rules engines* (silniki decyzyjne), systemy eksperckie, grafy wiedzy i logika / symbolika – nazywane zbiorczo, jako *GOF AI* (ang. *Good, Old-Fashioned AI*).

Termin sztucznej inteligencji nabrał tempa wraz z postępem w tej dziedzinie poczynionym przez firmy globalnego Internetu, takie jak Google, Facebook, Microsoft czy Apple – ich produkty takie jak: rozpoznawanie zdjęć, tłumaczenia języków, rozpoznawanie twarzy, opisywanie obrazków tekstowo, inteligentni asystenci, autonomiczne auta i inne – osiągnęły przełomowe wyniki, pobudzając ludzką wyobraźnię i ściągając zainteresowanie mediów. Implementacja mechanizmów AI w swoich produktach wiązała się z postrzeganiem firm jako nowoczesnych i wizjonerskich, została więc szeroko wykorzystywana marketingowo, prowadząc do wyrobienia się błędnych wyobrażeń o sztucznej inteligencji w odbiorze klientów. Myśląc zdroworozsądkowo większość z nas postrzega termin “sztuczna inteligencja” jako formę istnienia myślącego takiego jak my (człowiek), jednocześnie nie będącego człowiekiem – sztucznie zaadaptowaną jednostką, która jest myślącą i posiada samoświadomość – taka forma AI znana jest pod terminem *Strong AI / General AI* i jest ponad 50-cioletnim marzeniem badaczy, wierzących, że osiągając obecne symulowane inteligentne zachowania, zbliżamy się do powołania do istnienia prawdziwie inteligentnego bytu. Obecne prace w zakresie sztucznej inteligencji prowadzone są w bardzo wąskim zakresie, osiągając możliwe do uzyskania wyniki do momentu, w którym dotrzemy do granicy innowacji, na którą pozwala dotychczasowa technologia. Sceptycy AI postrzegają to jako niemożliwym do osiągnięcia jeszcze przez wiele lat,

---

<sup>28</sup> *Artificial Intelligence (AI) vs. Machine Learning vs. Deep Learning* [online] Skymind [dostęp: 27.09.2019]. Dostępny w Internecie: <https://skymind.ai/wiki/ai-vs-machine-learning-vs-deep-learning>

jednak postępy w tej dziedzinie cały czas są poczyniane, a ich historia sięga połowy XX-wieku, kiedy to pierwszy system sztucznej inteligencji pokonał w partii warcabów dotychczasowego mistrza ligi – było to pierwszy duży sukces maszyny zachowującej się inteligentnie, jednocześnie niebędącej krok po kroku zaprogramowanej do podejmowania deterministycznych decyzji – to maszyna więc decydowała jaki ruch wykonać i jej zwycięstwo jest zasługą jej systemu sztucznej inteligencji.

*Machine learning*, czyli uczenie maszynowe to dziedzina dająca maszynom możliwości nauki, jednocześnie nie będąc do tego ściśle zaprogramowanym, tak jak ma to miejsce np. w algorytmach składających się z instrukcji warunkowych *if/then*. Oprogramowanie posiada, więc zdolność do modyfikowania swojego procesu decyzyjnego, tak, aby podejmować decyzje najbardziej adekwatne do aktualnego środowiska (danych wejściowych), osiągając możliwość inteligentnej adaptacji w swoich zachowaniach i ewoluowania w wybranym przez algorytm kierunku. Machine learning postrzegany jest, jako poddział dziedziny sztucznej inteligencji, co oznacza, że jej metodyka zawiera się w zakresie AI, natomiast nie każda forma sztucznej inteligencji jest jednocześnie uczeniem maszynowym.

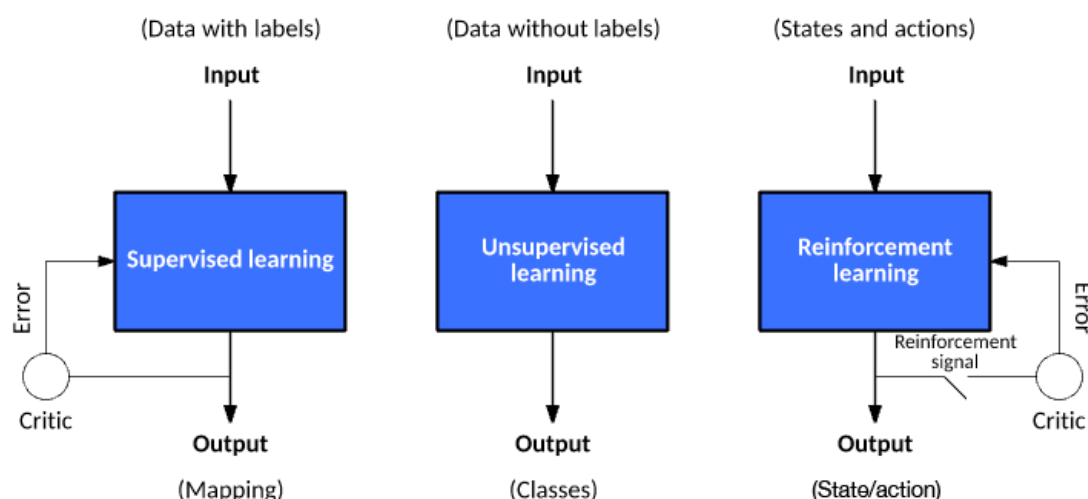
W uczeniu maszynowym algorytmy poddaje się procesowi uczenia w postaci wprowadzania dużego zbioru odpowiednio przystosowanych danych wejściowych. Są one następnie przetwarzane zgodnie z wybraną metodą, proces ten nazywany jest treningiem. WYROBIENIE SIĘ INTELIGENTNYCH ZACHOWAŃ następuje poprzez wychwycenie wzorców pomiędzy danymi przez algorytm uczenia maszynowego, np. różnice w pikselach na obrazkach, pozwalające na ocenę czym jest obiekt na nim przedstawiony. Głównymi rodzajami algorytmów uczenia maszynowego są (Rys.8.):

- Uczenie nadzorowane (ang. *Supervised Learning*) – algorytmy w procesie treningu wykorzystujące dane testowe już oznaczone na poprawny werdykt ich predykcji, w trakcie uczenia model modyfikuje więc swój wynik w stosunku do znanej poprawnej odpowiedzi na wprowadzone dane
- Uczenie nienadzorowane (ang. *Unsupervised Learning*) – trening przebiega z danymi nieposiadającymi oznaczeń, algorytm więc przetwarza dane w poszukiwaniu wzorców pomiędzy nimi, próbując nauczyć się odtwarzać dane wejściowe

- Uczenie przez wzmacnianie (ang. *Reinforcement Learning*) – specyficzny rodzaj algorytmów zorientowanych na osiągnięcie danych celów, uczenie polega na podejmowaniu wielu kroków metodą prób i błędów, maksymalizując wynik związany z danym celem, np. osiągnięciem wygranej w grze. Stosowane w systemach agentowych, grach, wykonywaniu zadań, robotyce, a także w podejmowaniu decyzji itp. Są to m. in. algorytmy: Q-Learning, Deep Adversarial Networks, Temporal Difference

Każdy z rodzajów ma swoje szczególne scenariusze zastosowania, nic nie stoi jednak na przeszkodzie, aby w bardziej wyrafinowanych systemach uczenia maszynowego łączyć wiele algorytmów z różnych ich rodzajów.<sup>29</sup>

Rys.8. Główne modele uczenia maszynowego



Źródło: *Models for machine learning* [online] IBM Developer [dostęp: 27.09.2019]. Dostępny w Internecie: <https://developer.ibm.com/articles/cc-models-machine-learning/>

Kluczowym jest dobór odpowiedniego rodzaju algorytmu do rozwiązywanego problemu, potencjalnie są one ogólne i mogą być zastosowane do wielu z nich, ale końcowy efekt może się między nimi różnić w dokładności predykowania, którą chcemy osiągnąć jak najwyższą, aby nasze modele sztucznej inteligencji były możliwie niezawodne. Osiąga się to dzieląc wejściowe dane treningowe na zestaw: danych walidacyjnych i danych testowych, zazwyczaj w zbliżonej proporcji, kilkakrotnie

<sup>29</sup> *Machine Learning Algorithm - Backbone of emerging technologies* [online] Vishakha Jha [dostęp: 03.10.2019]. Dostępny w Internecie: <https://www.techleer.com/articles/203-machine-learning-algorithm-backbone-of-emerging-technologies/>

mniejszej od zestawu treningowego. Etap walidacji wykorzystuje swój zestaw do oceny najlepszego wyniku dokładności pomiędzy modelami, które weryfikujemy. Gdy podejmiemy już decyzję odnośnie wyboru jednego z modeli, sprawdzamy na etapie testowym ostateczną dokładność na dotychczas niepoznanych danych zestawu testowego.<sup>30</sup> Separowanie danych wiąże się z potrzebą sprawdzenia jak algorytm zachowa się w scenariuszu bliskim rzeczywistemu, kiedy reaguje na nieznane dane wejściowe. W machine learningu najpowszechniejszym są 4 rodzaje problemów, które przypisuje się dwóm rodzajom algorytmów – dla *Supervised Learning* jest to: klasyfikacja oraz regresja, a dla *Unsupervised Learning*: analiza skupień i redukcja liczby wymiarów (Rys.9.):

- Klasyfikacja (ang. *Classification*) – określenie do jakiej kategorii przypisane są wprowadzane obserwacje, na podstawie znanych, sklasyfikowanych wcześniej obserwacji. Np. przeprowadzenie predykcji do jakiego rodzaju grupy krwi należy pacjent na podstawie wprowadzonych danych (jednej z czterech poznanych: A, B, AB albo 0). Algorytmy: klasyfikatorów liniowych takie jak regresja logistyczna czy naiwny klasyfikator bayesowski; maszyna wektorów nośnych (SVM), drzewa decyzyjne, sieci neuronowe, boosting, k najbliższego sąsiada i inne
- Regresja (ang. *Regression*) – proces estymacji relacji pomiędzy zmiennymi (danymi), prognozujący wynikową wartość numeryczną (np. wyniki notowań akcji na giełdzie na podstawie danych historycznych). Algorytmy takie jak: regresja liniowa, regresja logistyczna, ElasticNet, drzewa decyzyjne, regresor SGD, SVR itd.
- Analiza skupień (ang. *Clustering*) – zadanie grupowania obserwacji w klastry, w taki sposób, że obserwacje w jednym klastrze są do siebie bardziej podobne, niż do obserwacji w pozostałych klastach. Umożliwia to odkrycie zależności pomiędzy grupami danych i skategoryzowanie ich do jednego z klastrów obserwacji. Wykorzystywane m. in. w wykrywaniu anomalii, gdzie wprowadzane dane nie pasują do żadnej z wytrenowanych grup. Algorytmy: k najbliższych sąsiadów, k średnich, MeanShift, sieci neuronowe

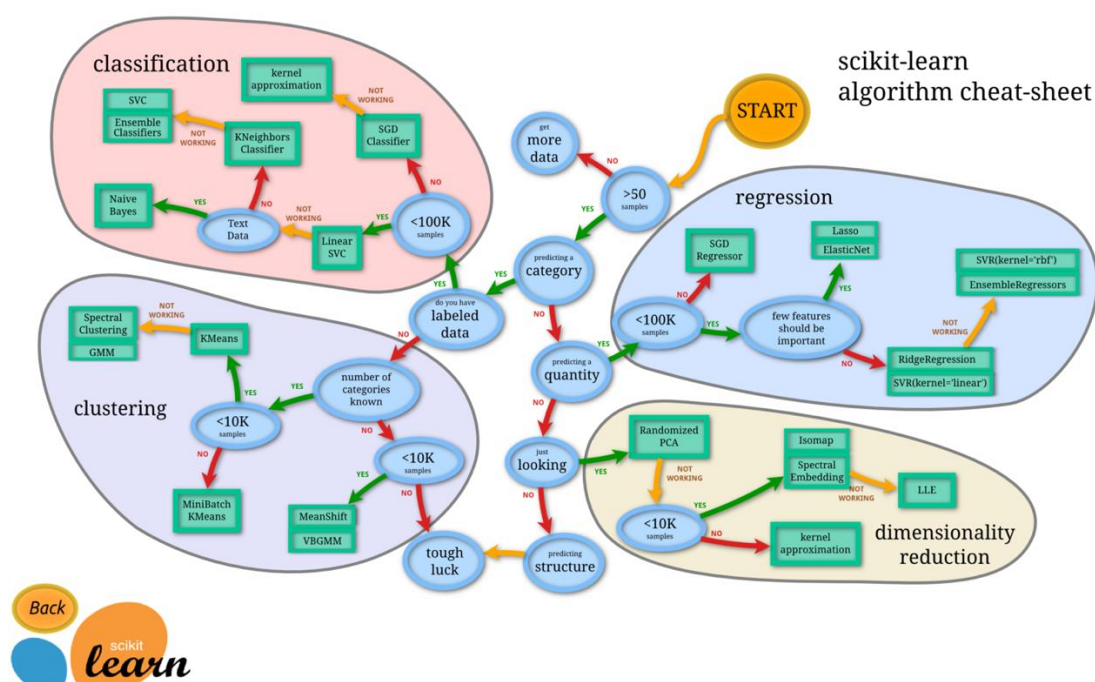
---

<sup>30</sup> What is the difference between test set and validation set? [online] Alexander Galkin [dostęp: 03.10.2019]. Dostępny w Internecie: <https://stats.stackexchange.com/questions/19048/what-is-the-difference-between-test-set-and-validation-set>

- Redukcja liczby wymiarów (ang. *Dimensionality reduction*) – proces redukowania ilości zmiennych będących przedmiotem badania do tylko tych kluczowych, mających wpływ na ostateczny wynik. Algorytmy: PCA, LDA, MDS, Isomap i LLE

Odpowiednie zastosowanie algorytmów do rozwiązywanego problemu pozwala osiągnąć pożądane zachowania *symulowanej inteligencji*, przedstawiając nam wyniki zbliżone do tych, osiąganych w tych zadaniach przez ludzi, a nawet przewyższające je.

Rys.9. Drzewo decyzyjne doboru metody ML dla danego problemu wg Scikit-learn



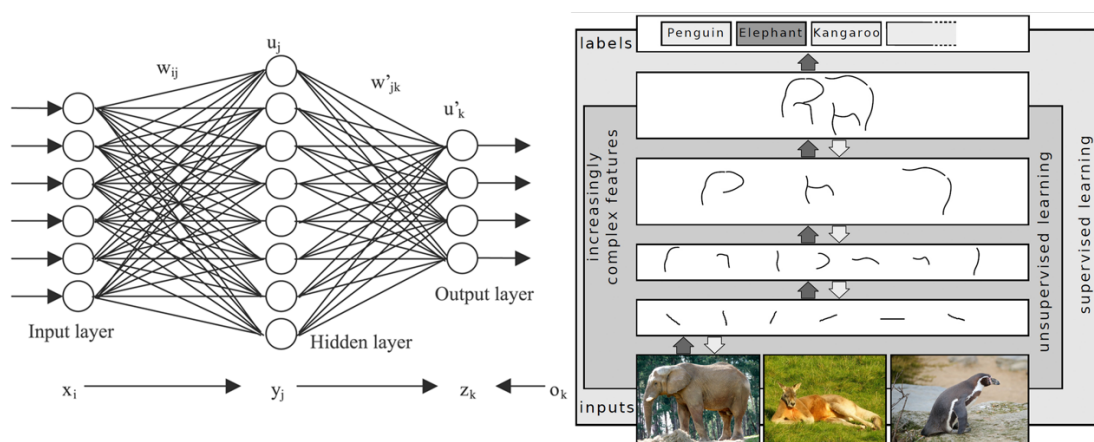
Źródło: *Scikit-learn Tutorial – Choosing the right estimator* [online] scikit-learn [dostęp: 27.09.2019]. Dostępny w Internecie: [https://scikit-learn.org/stable/tutorial/machine\\_learning\\_map/index.html](https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html)

W ostatnich latach szczególnie istotnym jest algorytm sztucznych sieci neuronowych (ang. *Artificial Neural Networks*) – koncept matematyczny luźno wzorowany na modelu przesyłania impulsów w mózgu poprzez *neurony*. Jego początki sięgają statystycznej metody *regresji logistycznej*, której szczególna forma to jednowarstwowa sieć neuronowa, stanowiło to podwaliny do dalszego rozwoju tej dziedziny algorytmu. W sztucznej sieci neuronowej pojedynczą jednostkę stanowi neuron, który ma przypisaną do siebie wagę, a skupiska neuronów formowane są w warstwy: pierwszą wejściową, wiele warstw pośrednich oraz ostatnią wyjściową, a

między neuronami zawiera się połączenia. Struktura utworzona z warstw ma postać ważonego grafu skierowanego. W procesie uczenia (wprowadzania danych) do sieci neuronowej, każdy z neuronów przyjmuje wartości numeryczne i przeprowadza na nich obliczenia (operacje macierzowe) modyfikując wstecznie wartości wag na neuronach w odpowiedzi na przetwarzanie wprowadzonych danych. Tym sposobem na wynikowych neuronach warstwy produkcyjnej otrzymuje się wartości, które są wynikiem predykcji.

*Deep Learning*, czyli głębokie uczenie jest specyficzną formą uczenia maszynowego, zorientowanego wokół sztucznych sieci neuronowych składających się z wielu warstw pośrednich, stąd też tzw. *głębia* tego rodzaju algorytmów sztucznej inteligencji. Głębokie sieci neuronowe (ang. *Deep Neural Networks*) przyczyniły się do ponownego rozkwitu sztucznej inteligencji w XXI wieku, osiągając rekordowe wyniki dokładności w wielu trudnych problemach, m. in: rozpoznawania obrazów, dźwięku, systemach rekomendacji, przetwarzaniu języka naturalnego i innych. Zaletą zastosowania wielu warstw pośrednich jest możliwość sieci do wyuczenia się bardziej złożonych wzorców. Przekazując dane przez więcej warstw, przeprowadzane jest więcej operacji matematycznych, każda z nich wykrywa więc część wzorca, w kolejności od najprostszych przy początkowych warstwach, po bardziej złożone im głębiej w warstwy sieci, np. przy przetwarzaniu obrazu pierwsza warstwa zawiera informację o kilku pikselach, na drugiej warstwie może już składać się na kształt linii, a w trzeciej na zarys prostego kształtu sylwetki (Rys.10.).

Rys.10. Głęboka sieć neuronowa, a rozpoznawanie wzorców poprzez warstwy



Źródło: Na podstawie *Deep learning* [online] Sven Behnke [dostęp: 04.10.2019]. Dostępny w Internecie: [https://en.wikipedia.org/wiki/Deep\\_learning](https://en.wikipedia.org/wiki/Deep_learning) oraz *Artificial neural networks are changing the world. What are they?* [online] Graham Templeton [dostęp: 04.10.2019]. Dostępny w

Internecie: <https://www.extremetech.com/extreme/215170-artificial-neural-networks-are-changing-the-world-what-are-they>

Postęp osiągnięty dzięki głębokim sieciom neuronowym wiąże się ze zwiększonym zapotrzebowaniem na moc obliczeniową, ponieważ każda warstwa wykonuje więcej operacji. Popyt na sprzęt dedykowany do zastosowania w algorytmach sztucznej inteligencji zwiększa się, powstają nowego rodzaju procesory np. *TPU* (ang. *tensor processing unit*) i inne przeznaczone dla efektywniejszego przeprowadzania uczenia maszynowego.

Głębokie sieci neuronowe wraz z wysiłkami badaczy ewoluowały w nowe formy algorytmów jeszcze bardziej zwiększające osiągi w wykonywaniu dotychczasowych inteligentnych zadań oraz pozwalając na starania w próbach rozwikłania coraz śmielszych problemów. *Convolutional Neural Networks (CNN)* zawierają różne rodzaje warstw pośrednich: *convolutional layers* i *pooling layers*, przetwarzając dane wejściowe np. obrazka algorytm przesuwając się ramką po pikselach i redukuje ich wartości dla kolejnej warstwy (*convolution*), a następnie zmniejsza wymiarowość danych poprzez wybranie pojedynczej największej wartości z wielu ramek, łącząc z nich nową macierz wyników, pozbawioną mniej istotnych informacji z wyższej warstwy (*pooling*). Pozwala to na wyuczenie się sieci jeszcze więcej ilości wzorców, niezwykle zwiększając jej dokładność predykcji. Wykorzystywane prawie wyłącznie dla rozpoznawania obrazu sieci neuronowe *CNN* są jednym z głównych powodów rozkwitu deep learningu w obecnej dekadzie, ich zastosowanie pomaga rozwiązać kluczowy problem jakim jest percepcja wizualna maszyn, wykorzystywana m. in. w autonomicznych autach, robotyce, dronach, diagnozach medycznych i innych. *Recurrent Neural Networks (RNN)* prócz standardowych połączeń sieci neuronowych od warstw wejściowych, pośrednich do wyjściowych dodatkowo zawierają połączenia wsteczne pomiędzy warstwami, pozwalając wrócić przetwarzanym danym do warstwy poprzedniej w celu ponownego wpływu na obliczenia. Adresuje to istotny problem, który nie był rozważany w tradycyjnej sieci neuronowej, mianowicie wpływ poprzednich obserwacji na wynik predykcji. W myślni sieci neuronowej po procesie treningu uzyskuje się statyczny model predykcji, niewiążący informacji wstecznej z poprzednio przetwarzanych obserwacji, gdy w rekurencyjnej sieci neuronowej wyniki przy poprzednich obserwacjach są do pewnego stopnia zapamiętywane i wpływają na

wynik aktualnej predykcji, generując za każdym razem nowy dynamiczny model reagujący na przeszłe informacje. Dzięki temu uzyskuje się wyższą dokładność w zadaniach, gdzie istotna jest wiedza z przeszłych obserwacji, tak jak ma to miejsce np. w prognozach giełdowych cen akcji, ale również w problemie przetwarzania języka naturalnego (*NLP*, ang. *Natural Language Processing*), gdzie ważny jest kontekst wyrażen. Szczególna forma rekurencyjnych sieci neuronowych *Long Short-Term Memory (LSTM)* wyróżnia się posiadaniem pamięci krótkotrwałej i długotrwałej, mając wpływ na zachowanie informacji przeszłej, kluczowej dla wyniku przyszłej predykcji. Wielu badaczy postrzega rekurencyjne sieci neuronowe jako krok naprzód w dziedzinie sztucznej inteligencji, ponieważ w mózgu zwierząt sieci neuronowe formują się właśnie w rozległy układ rekurencyjny wzajemnie przekazujący impulsy pomiędzy neuronami.<sup>31</sup>

Na początku lat 80-tych wraz z rozwojem dziedziny logicznych języków programowania takich jak *Prolog* odkryto potencjał w systemach opierających się na jego mechanizmach, m. in.: bazy wiedzy, deklarowanie twierdzeń i ewaluacja zapytań logicznych. W przeciwieństwie do czwartej generacji języków (np. *SQL*, *R*, *MATLAB*) cechujące się deklaratywną, bliską językowi naturalnemu składnią oraz dedykowanym środowiskiem, nowa piąta generacja miała opierać się na rozwiązywaniu problemów logicznych, zamiast stosowania algorytmów. Wiele instytucji rządowych zainwestowało olbrzymie pieniądze w badania nad stworzeniem *Fifth Generation Computer*, komputera piątej generacji opartego w całości na programowaniu logicznym i masowego zastosowania paralelizacji. System ten miał stanowić podwaliny pod dalsze próby stworzenia prawdziwej sztucznej inteligencji, jednakże postępy prac nie oddawały oczekiwanych efektów, a po kilku latach wyczerpało się finansowanie projektu. Inicjatywa piątej generacji miała znikomy wpływ na informatykę i z czasem stała się tematem historycznym, a sam projekt *FGCS* opiniowany jest albo, jako kompletna porażka, albo koncept wyprzedzający swoje czasy.<sup>32</sup> *Neuroevolution*, czyli wykorzystywanie przez sieci neuronowe algorytmów genetycznych i programowania ewolucyjnego pozwala na bardziej dowolną optymalizację swojej dokładności. Sieć

---

<sup>31</sup> *Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning* [online] Vibhor Nigam [dostęp: 05.10.2019]. Dostępny w Internecie: <https://towardsdatascience.com/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90>

<sup>32</sup> *Fifth generation computer* [online] Wikipedia [dostęp: 07.10.2019]. Dostępny w Internecie: [https://en.wikipedia.org/wiki/Fifth\\_generation\\_computer](https://en.wikipedia.org/wiki/Fifth_generation_computer)



neuronowa funkcjonująca na zasadzie neuroewolucji może zmieniać nie tylko swoje wagi, ale również całą topologię sieci oraz jej parametry. Badacze w firmie Uber wysnuli opinię, że jej zastosowanie jest lepsze od dotychczasowej metody *gradient descent*, pod względem omijania przez niej niepożądanych minimów lokalnych. W przeciwieństwie do stosowania algorytmów z uczenia nadzorowanego, sieć neuronowa oparta o neuroewolucję wymaga jedynie miary wyniku sieci w wykonywanym zadaniu, będąc pod tym kątem zbliżona do zasady funkcjonowania uczenia przez wzmacnianie (*reinforcement learning*).<sup>33</sup>

Uczenie maszynowe obarczone jest dużą ilością problemów, które przez nadchodzące lata muszą zostać rozwiązane, jeżeli chcemy dotrzeć do ogólnego AI. Podstawowymi trudnościami są *overfitting* oraz *underfitting*, czyli przeuczenie oraz niedouczenie algorytmu. W trakcie treningu bardzo łatwo jest przeuczyć sieć ciągle zwiększając jej dokładność, jest to sytuacja, w której algorytm przestaje predykować generalizując na podstawie danych treningowych, a dosłownie wyucza się tych danych, reagując prawidłowo tylko na te dane. Np. algorytm rozpoznawania obrazu, którego celem jest stwierdzenie, jakie zwierzę jest na obrazku, po przedstawieniu do uczenia obrazków samych czarnych psów, będzie sądził, że psy są tylko czarne, w efekcie nie rozpozna na nowych danych innych psów, niż te czarne. W daleko idącym przypadku przeuczenie dąży do jeszcze większego uogólnienia – każda czarna plama na zdjęciu byłaby określana, jako pies. Natomiast niedouczenie pojawia się w przypadku, kiedy trenując algorytm nie chcemy, aby wykrywał zbyt wielu wzorców, zatrzymujemy się, więc z treningiem stosunkowo wcześniej, na tyle, że algorytm może nie wychwycić głównych, najistotniejszych dla nas wzorców na danych w czasie predykcji.<sup>34</sup> Aby osiągnąć zamierzony efekt uczenia maszynowego potrzebna jest ogromna ilość danych, co stosunkowo różni się od uczenia obserwowanego u ludzi – dziecko, któremu raz pokaże się np. kota z powodzeniem zidentyfikuje go ponownie, na innym obrazku, kiedy ludzie uczą się wzorców niemal natychmiastowo, algorytmy uczenia maszynowego potrzebują tysięcy iteracji przetwarzania danych, zanim ich możliwości predykcji zbliżą się do ludzkiej dokładności w zadaniu sztucznej inteligencji. Obecne

---

<sup>33</sup> *Artificial intelligence can 'evolve' to solve problems* [online] Matthew Hutson [dostęp: 07.10.2019]. Dostępny w Internecie: <https://www.sciencemag.org/news/2018/01/artificial-intelligence-can-evolve-solve-problems>

<sup>34</sup> *What Are Overfitting and Underfitting in Machine Learning?* [online] Anas Al-Masri [dostęp: 07.10.2019]. Dostępny w Internecie: <https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690>

rekordowe sieci neuronowe składają się z wielu milionów parametrów i nie sposób człowiekowi ich wszystkich zrozumieć, docieramy, więc do momentu, kiedy osiągamy najlepsze wyniki na modelach, których kompletnie nie rozumiemy, a sama maszyna sztucznej inteligencji staje się tzw. czarną skrzynką (*blackbox*), przyjmującą dane i zwracającą pożądany wynik – aby pójść naprzód w dziedzinie sztucznej inteligencji musimy rozumieć i wiedzieć jak funkcjonują jej mechanizmy. Możliwość zaprogramowania AI wiąże się z posiadaniem rozwiązania (danych) na stawiany problem, nie pozwoli nam to, więc rozwiązać problemów o dotychczas nieznanym rozwiązaniu. Przyszłość sztucznej inteligencji prawdopodobnie w dużej mierze zależy od rozwoju uczenia nienadzorowanego (*unsupervised learning*) danych skategoryzowanych wysokiej jakości jest bardzo niewiele, a ich pozyskiwanie kosztowne, możliwość osiągania zachowań inteligentnych na danych szerzej dostępnych pozwoli na szersze rozpropagowanie algorytmów uczenia maszynowego.

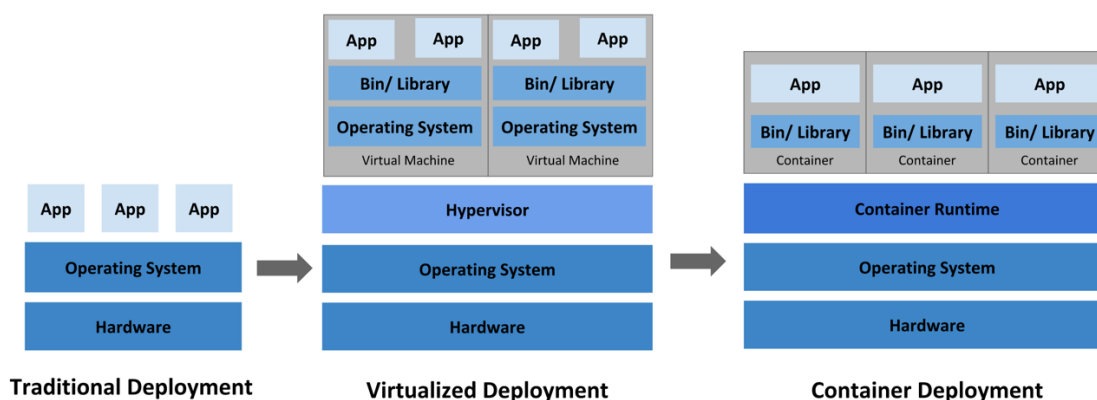
W historii sztucznej inteligencji wielokrotnie sądzono, że zbliżamy się do stworzenia myślącej maszyny, nie mając jednak pojęcia jak odległa jest to dla ludzkości perspektywa. Wyróżnia się dwa okresy tzw. *AI winter* (ang. zimy AI), pierwsza nastąpiła w latach 70-tych, kiedy to po dekadzie intensywnych badań i kosztownego finansowania nie przyniosły one oczekiwanych rezultatów. Badacze zawiedli w przewidzeniu jak trudnym zadaniem jest budowa sztucznej inteligencji, padały wtedy stwierdzenia takie, jak Herberta Simona: „*machines will be capable, within twenty years, of doing any work a man can do*” (ang. „maszyny będą w stanie w ciągu dwudziestu lat wykonać każdą pracę, jaką może wykonywać człowiek”) oraz Marviniego Minskiego: „*within a generation [...] the problem of creating 'artificial intelligence' will substantially be solved*” (ang. „w ciągu pokolenia [...] problem stworzenia sztucznej inteligencji zostanie rozwiązany”), jak okazało się docierając do czasów nam obecnych nadal jest to niemożliwe. Druga *AI winter* nastąpiła pod koniec lat 80-tych, badania pobudzone dzięki sukcesom systemów ekspertowych, ponownie spotkały się z porażką po załamaniu rynku *LISP Machine*, komercyjnego rozwiązania systemu ekspertowego. Przeszłe doświadczenia przedsięwzięć sztucznej inteligencji pouczają o tym jak niezwykle złożonym jest tworzenie AI oraz jak wielkie (naiwne) oczekiwania są wobec niej przedstawiane – prawdopodobnym jest oczekiwać ponownego nastąpienia okresu *AI winter*, kiedy rynek przeładowany marketingiem sztucznej inteligencji odkrywa, że to, co może zaoferować obecna technologia, nie jest nawet

ułamkiem oczekiwań ludzkości wobec maszyny inteligentnej, skutkując zaprzestaniem badań oraz końcem finansowania. Trudność w zaakceptowaniu tworu sztucznej inteligencji została opisana jako *AI effect*: „*AI is whatever hasn't been done yet.*” (ang. „AI jest wszystkim, czego jeszcze nie zrobiono”) – Douglas Hofstadter, obrazuje to problem psychologiczny w postrzeganiu tworu, jako zachowującego się inteligentnie, ze względu na to, że zasada ich działania jest przez ludzi dobrze poznana, odnosi się to do humanistycznego przekonania, że inteligencja jest niepoznawalna oraz jest czymś tak głęboko ludzkim, że wszystko inne zachowujące się ‘inteligentnie’ nie jest w istocie tworem inteligentnym. Kontrowersyjnie, bądź nie, za przedmiot inteligentny uznawany jest kalkulator – jest on narzędziem tak pospolitym, że myślenie o nim w kategoriach *sztucznej inteligencji* zdaje się być irracjonalne, jednakże to maszyna, która zamienia graficzne symbole na wyniki obliczeń, dychotomia takiego postrzegania problemu myślących maszyn stanowi jeszcze większy dysonans w pospolitym rozumieniu AI pośród ludzi. Pomimo wielu starań przez ponad pół wieku wciąż jesteśmy na dalekiej drodze w osiągnięciu prawdziwej *strong / general AI*, obecny stan rozwoju w dziedzinie czyni postępy w nadawaniu inteligentnych zachowań poprzez znajdowanie wzorców w danych – przedstawia to przed jak bardzo trudnym, złożonym zadaniem stoi ludzkość.

### **1.7 Orkiestracja oraz komunikacja systemu w chmurze, przetwarzanie strumieniowe**

Infrastruktura chmurowa stała się następnym etapem postępu w tworzeniu aplikacji internetowych o globalnej skali – przy coraz większym zapotrzebowaniu na moc obliczeniową, przestrzeń dyskową i pamięć RAM skalowanych aplikacji, podejście wirtualizowania na pojedynczej fizycznej instancji infrastruktury nie spełniało wymagań – wirtualizacja w chmurze pozwoliła na dowolne wykorzystanie infrastruktury i jeszcze prostsze osiągnięcie potrzebnej skalowalności pomiędzy aplikacjami zorientowanymi na kontenerach (Rys.11.). Do liderów dostawców platform chmurowych należą m. in.: Amazon AWS, Microsoft Azure oraz Google Cloud Platform.

Rys.11. Ewolucja wdrożeń w kierunku wirtualizacji w chmurze



Źródło: *What is Kubernetes* [online] The Kubernetes Authors [dostęp: 11.10.2019]. Dostępny w Internecie: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

Kontenery są zbliżone do maszyn wirtualnych (VMs) we wdrożeniu wirtualizowanym na pojedynczej maszynie, natomiast posiadają mniejsze restrykcje wobec izolacji, np. współdzielą jeden system operacyjny oraz środowisko kontenerowe. Podobnie do VM, posiadają własny system plików, CPU, pamięć itd. Dzięki temu, że kontenery są niezależne od infrastruktury, mogą być łatwo migrowane pomiędzy platformami chmurowymi i różnymi dystrybucjami systemów operacyjnych.<sup>35</sup> Taka abstrakcja infrastruktury umożliwia budowanie systemu jako niezależnych komponentów, wpisując się w architekturę aplikacji mikroserwisowych, będących formą reaktywnego systemu aplikacji rozproszonej.

Wraz z popularyzacją konteneryzacji w chmurze, powstały dedykowane narzędzia powszechnie wykorzystywane podczas opracowywania oraz wdrażania oprogramowania. Podczas procesu deweloperskiego lokalnie używany jest *Docker*, pozwala na tworzenie kontenerów z poziomu kodu i pracy programisty, definiując środowisko działania aplikacji tak, żeby możliwe było jego proste odtworzenie na dowolnym innym środowisku programistycznym. Narzędzie *Docker* jest dostępne na wszystkich głównych systemach operacyjnych i stanowi model wirtualizacji kontenerów w środowisku tworzenia oprogramowania – dzięki niemu możemy łatwo pracować z kodem, stosować *pipeline'y CI/CD* oraz przystosowywać aplikację do wdrożenia w środowisku platform chmurowych (np. na AWS ECS). W scenariuszach bardziej złożonych, kiedy dla działania aplikacji należy wdrażać wiele kontenerów

<sup>35</sup> *What is Kubernetes* [online] The Kubernetes Authors [dostęp: 11.10.2019]. Dostępny w Internecie: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

potrzebny jest orkiestrator, który zapewni ich niezawodne działanie. *Kubernetes* stanowi narzędzie, które automatyzuje wdrożenia, skalowanie i zarządzanie aplikacjami skonteneryzowanymi. Zapewnia mechanizmy *load balancing* oraz *service discovery* kluczowe w architekturze mikroserwisowej, zarządzanie przestrzenią i konfiguracją, odpowiada za stan kontenerów poprzez mechanizm *self-healing*, w zależności od potrzeb *Kubernetes* zrestartuje kontener, jeśli ten napotka błąd, z którego nie powróci do działania.<sup>36</sup> Platformy chmurowe wraz z rosnącym zapotrzebowaniem na kontenery, zaczęły tworzyć własne rozwiązania do wykorzystania pomiędzy nimi, m. in. objawia się to w powstaniu nowych implementacji baz danych, takich jak Amazon DynamoDB czy Azure Cosmos DB, stanowi to alternatywę do wdrażania jako kontener innych rodzajów baz danych, takich jak PostgreSQL czy MongoDB oraz zapewnia prostą konfigurację i wykorzystanie zasobów chmury w sposób skalowalny – dąży to w kierunku wdrożeń *Serverless*, czyli aplikacji nie funkcjonujących na kontenerach, lecz na samodzielnych funkcjach (partiach kodu) wykonywanych niezależnie w chmurze.

Podstawowym fundamentem systemu reaktywnego, a także złożonej architektury mikroserwisowej jest *asynchronous message passing* – nieblokująca komunikacja pomiędzy serwisami, oszczędzająca zasoby i zwiększająca wydajność systemu, ale również podtrzymująca jego niezawodność. W tym celu wykorzystywani są *message brokers*, narzędzia takie, jak RabbitMQ czy Apache Kafka. Powstały one w scenariuszach zwiększenia wydajności systemów, wykonujących intensywne zadania, np. proces przetwarzania obrazu jest *CPU-intensive* może więc spowodować duże spowolnienie aplikacji, jeżeli w jej głównej logice zostanie umieszczone takie zadanie i nagle drastycznie zwiększy się ruch w produkcji. Takie zadania były wyodrębniane na osobne usługi, a komunikacja pomiędzy aplikacją główną, a usługą odbywała się za pomocą wiadomości, kolejkowanych i odbieranych przez serwis do przetworzenia. Mechanizm ten wykorzystano na dużą skalę w systemach rozproszonych, każdy z komponentów komunikuje się między sobą bazując na asynchronicznych wiadomościach, zapewniając zgodnie z *Reactive Manifesto*: elastyczność, odporność, a ostatecznie wymaganą responsywność.<sup>37</sup>

---

<sup>36</sup> *The Advantages of Using Kubernetes and Docker Together* [online] Stackify [dostęp: 11.10.2019]. Dostępny w Internecie: <https://stackify.com/kubernetes-docker-deployments/>

<sup>37</sup> *Building Microservices: Inter-Process Communication in a Microservices Architecture* [online] NGINX Inc. [dostęp: 12.10.2019]. Dostępny w Internecie: <https://www.nginx.com/blog/building-microservices-inter-process-communication/>

Prócz architektonicznych rozwiązań asynchroniczności, limitem są również protokoły komunikacyjne pomiędzy serwisami, standardowe *REST - HTTP JSON*, stawia problemy jednokierunkowej komunikacji w paradygmacie *request/response*, synchroniczności oczekiwania na odpowiedź, przekazywania danych w postaci kompletnego zasobu wymagającego każdorazowego osobnego zapytania, dodatkowo serializacja danych jest łatwo czytelna, ale zajmuje zbyt dużo zasobów. Aby rozwiązać te problemy poszukiwano innych form komunikacji, m. in. opracowany przez Google framework gRPC wykorzystujący Protobuf – pozwala na definiowanie wywołań metod pomiędzy serwisami, binarną oraz strumieniową komunikację dwukierunkową synchroniczną, jak i asynchroniczną, minimalny narzut serializowanych danych, a będący w całości oparty o specyfikację protokołu *http/2*.<sup>38</sup> W scenariuszu strumieniowych systemów danych Fast Data szczególnie istotnym jest strumieniowanie danych poprzez protokół komunikacyjny, w tym celu powstał *RSocket*, binarny protokół strumieniowania danych poprzez TCP implementujący semantykę *Reactive Streams*, zapewniający m. in. *backpropagation* na poziomie komunikacji między serwisami. Rozwój podejścia reaktywnego wpłynął również na gRPC, objawiając się powstaniem *Reactive gRPC*, mający jednak pewne limitacje wynikające z protokołu *http/2*.

Przetwarzanie strumieniowe jest jednym z możliwości pracy na ogromnych wolumenach danych Big Data, bez natrafiania na problemy braku zasobów czy spowalniania oprogramowania. Stosuje się w tym celu m. in: *Akka Streams* oraz *Apache Spark*. Pierwsze z nich wywodzi się z *Akka*, służącego do budowania operacyjnych aplikacji rozproszonych, ma więc charakter systemu rozproszonego wzbogaconego o strumieniowanie. *Spark* natomiast od początku zastosowania jest narzędziem analitycznym, i w tym zakresie posiada najwięcej udogodnień, limituje go więc sam mechanizm partycjonowania, oznacza to, że nie posiada cech reaktywnego strumieniowania danych. Ze względu na różnice w scenariuszach zastosowań, często więc architektura będzie składać się z obu tych narzędzi, przy czym *Akka Streams* stanowi część operacyjną, a *Apache Spark* typowo analityczną.<sup>39</sup> Stosowanie programowania reaktywnego dla strumieniowanych danych wiąże się również z inną,

---

<sup>38</sup> *gRPC Concepts* [online] gRPC [dostęp: 12.10.2019]. Dostępny w Internecie: <https://grpc.io/docs/guides/concepts/>

<sup>39</sup> Kamil Owczarek, *Spark and Akka for Big Data Systems, in practice* Scalar Conference 2018 by SoftwareMill.Warszawa, 2018

bardzo istotną cechą interfejsów metod, mianowicie zastosowaniu paradygmatu *Functional Programming* obok tradycyjnego *Object Oriented Programming*. Programowanie funkcyjne jest szczególnie ważne dla pracy na danych, gdyż pozwala na proste, deklaratywne operowanie na ich strumieniach z możliwością zastosowania ukrytej paralelizacji.

Niniejszy rozdział zaprezentował wstęp do złożonej tematyki jaką są systemy rozproszone, operacyjne Big Data, Machine Learning, chmura, paradygmaty dużej skali itp. Omówiono ich umotywowanie, drogę rozwoju, historię jaka za nimi przemawia. Wprowadzono również przypadki zastosowania, w niektórych z nich nawet historie wdrożeń zakończonych sukcesem. Nie zabrakło również rozważań nad zagadnieniami, które wymagają osobnej oceny pod kątem zastosowania, bezpieczeństwa, kosztów wykorzystania, czy etycznych aspektów użycia, tak jak ma to miejsce w sztucznej inteligencji. Jest to niezwykle obszerna dziedzina i wyczerpanie jej w rozsądnym wymiarze stanowiło trudne zadanie, niemniej jednak położono nacisk na syntezę wystarczającej ilości wiedzy, aby odbiorca swobodnie mógł poruszać się w omawianych zagadnieniach. Zainteresowanym badaczom polecane jest pogłębianie wiedzy w poruszanej tematyce, za każdą z przedstawionych dziedzin istnieje znacznie większa ilość informacji.

## 2. Komponenty reaktywnego systemu rozproszonego Big Data

Rozdział przedstawia szereg narzędzi wykorzystywanych w tzw. aplikacjach SMACK stack, tworzących rozwiązania skalowalne, czasu rzeczywistego i napędzane danymi. Omówiony został główny język programowania Scala, framework mikroserwisowy Lagom oparty o Akka z Actor model. W części bazodanowej opisano nierelacyjną bazę danych Cassandra. Dla części analitycznej Fast Data zostały przedstawione narzędzia Apache Spark oraz Spark ML dla uczenia maszynowego. A rolę komunikacji poprzez event bus przedstawiono w postaci Apache Kafka. Ostatnia sekcja skupia się na wykorzystywanych wzorcach projektowych, wspierających budowę dużego, skalowalnego oprogramowania. Całość dobranych komponentów składa się na architekturę opracowanego systemu reaktywnego, będącego przedmiotem następnego rozdziału.

### 2.1 Scala – scalable programming language

Budowa systemu rozproszonego wiąże się z pewnego rodzaju złożonością, która jest problematyczna dla prawie każdego aspektu produktu, od architektury, po komunikację, implementację rozwiązań w projekcie, aż po zarządzanie nim i określanie przewidywalnych ram projektu (takich, jak terminowość, zapotrzebowanie na programistów czy szacowanie kosztów). Od strony technologicznej wymaga to zastosowania narzędzi przeznaczonych dla scenariuszy złożonych, które prosto adaptują się na etapie prototypu tak samo, jak i w już rozwiniętym produkcie, a także jego utrzymaniu – zapewniając potrzebną elastyczność inżynierom oprogramowania w przewidywalny sposób (bez niestandardowych obejść). Technologie te skupiają się na utrzymaniu złożoności możliwie jak najprostszej, gdyż tylko prostota pozwala uniknąć pułapek systemu skomplikowanego – taką technologią jest język programowania Scala.

Scala, czyli skalowalny język programowania (ang. *scalable programming language*) to technologia przeznaczona dla tworzenia złożonych aplikacji. Określenie terminem ‘skalowalnego’ oznacza posiadanie cech wspierających pracę nad oprogramowaniem podczas jego ewolucji, powstrzymując przed stosowaniem złych rozwiązań, które w przyszłości skomplikują pracę oraz zapewnienie potrzebnych



elementów do zachowania czystego i wysoko jakościowego kodu w dużych i złożonych systemach.<sup>40</sup> Jednocześnie Scala posiada wbudowane rozwiązania dla konkurencyjności i paralelizacji, a również asynchroniczności z nieblokowaniem, dążąc do łatwego tworzenia systemów rozproszonych np. poprzez aktorów. Technologia ta została zaprojektowana z myślą o oprogramowaniu nowej generacji wykorzystującym możliwości wielordzeniowych procesorów oraz infrastrukturę chmurową. Myślą przewodnią Scali jest „*Do More With Less (Code)*” (ang. „Zrób więcej z mniej (kodu)”) oznaczając, że możemy zaimplementować bardziej złożony program lub logikę z mniejszą ilością linijek kodu.

Scala jest językiem kompilowanym wykonującym się na JVM (ang. *Java Virtual Machine*) i jest alternatywą dla powszechnego języka Java, zachowując jednak przy tym pełną interoperacyjność z kodem napisanym w Javie. Oba języki mogą być dowolnie wykorzystywane w projekcie i korzystać z opracowanych przez nie implementacji pomiędzy sobą. Dodatkowo Scala zyskuje dostęp do szerokiej kolekcji wysokojakościowych bibliotek opracowanych dla Javy, które można wykorzystać w ramach języka Scala. Język ten prócz znanego z Javy paradygmatu programowania obiektowego (OOP) łączy w sobie paradygmat programowania funkcyjnego (FP), dając dowolność implementowania kodu w bardziej przystępnej formie, niż ramy narzucone przez Javę oraz uzyskanie zalet typowych dla obu paradygmatów (lub eliminacji wad jednego z nich). Wysoki poziom wsparcia programowania funkcyjnego objawia się w formie zaawansowanych konceptów, takich jak *monads*, *higher-order functions*, *currying*, *closures*, *functions as a value*, *pattern matching* i inne, oferując kompletny zestaw narzędzi do tworzenia złożonych systemów w oparciu o paradygmat funkcyjny. Główną zaletą języka Scala jest zwięzła, elegancka i bezpiecznie typowana składnia, pozwalająca skupić się na rozwiązywaniu problemu, zamiast pisania zbędnego kodu, osiągając o wiele większą prędkość i wydajność, niż jest to możliwe w Javie. Prócz zastosowania bardziej zminimalizowanej składni, typowanie statyczne języka posiada maksymalnie zredukowany narzut dzięki zastosowaniu *type inference*, czyli mechanizmu dedukcji typu, bez potrzeby jego jawnego wpisywania do kodu, znacznie ułatwia to pisanie oraz kompozycję kodu, bez ręcznego definiowania typów w miejscach, kiedy kompilator może inteligentnie wydedukować typ za programistę. Poza

---

<sup>40</sup> *What is scalable programming language?* [online] Simone Pezzano [dostęp: 24.10.2019]. Dostępny w Internecie: <https://www.quora.com/What-is-scalable-programming-language>

oferowanymi szerokimi możliwościami, Scala pozwala również na rozszerzanie samego języka o nowe elementy języka domenowego dopasowanego do scenariusza biznesowego.<sup>41</sup>

Scala to lepsza Java, zaprojektowana przez prof. Martina Odersky'ego – twórcę Sun *javac compiler* oraz mechanizmu *Java 5 Generics* – jako pragmatyczna ewolucja mechanizmów Javy, będącą bardziej zwięzłą i produktywną jej wersją. Jest najpopularniejszą alternatywą dla Javy na JVM. Scala przejęła sektor systemów Fast Data, oferując nową falę silników przetwarzania wysokiej prędkości i czasu rzeczywistego. Narzędzia takie, jak: Apache Spark, Akka / Akka Streams, Apache Kafka, Apache Flink oferują możliwości przetwarzania Big Data w czasie rzeczywistym bazując na strumieniowaniu, będąc opracowane w tymże języku. Niszą Scali są, więc (reaktywne) systemy rozproszone opracowane w Akka oraz przetwarzanie strumieniowe Big Data dzięki Apache Spark. Zgodnie z potrzebami dużych i złożonych systemów reaktywnych, wykorzystywane przez firmy takie, jak: Apple, Coursera, IBM, Intel, Mesosphere, Databricks i inne.<sup>42</sup> Przykładowe elementy języka zostały przedstawione na Listingu nr 2.

#### Listing nr 2. Wybrane funkcjonalności oraz implementacje w języku Scala

```
// Definicja klasy oraz Companion Object
class Author(val firstName: String,
             val lastName: String) extends Comparable[Author] {

    override def compareTo(that: Author) = {
        val lastNameComp = this.lastName compareTo that.lastName
        if (lastNameComp != 0) lastNameComp
        else this.firstName compareTo that.firstName
    }
}

object Author {
    def loadAuthorsFromFile(file: java.io.File): List[Author] = ???
}

// Type inference
```

---

<sup>41</sup> *The Scala Programming Language* [online] École Polytechnique Fédérale Lausanne (EPFL) [dostęp: 26.10.2019]. Dostępny w Internecie: <https://docs.scala-lang.org/tour/tour-of-scala.html>

<sup>42</sup> *Scala – Part of Lightbend Platform* [online] Lightbend [dostęp: 26.10.2019]. Dostępny w Internecie: <https://www.lightbend.com/scala-part-of-lightbend-platform>

```

class Person(val name: String, val age: Int) {
    override def toString = s"$name ($age)"
}

def createRandomPeople() = {
    val names = List("Alice", "Bob", "Carol", "Dave", "Eve", "Frank")

    for (name <- names) yield {
        val age = (Random.nextGaussian() * 8 + 20).toInt
        new Person(name, age)
    }
}

val people = createRandomPeople()
// people: List[Person] = List(Alice (16), Bob (16), Carol (19), Dave (18), Eve (26),
Frank (11))

// Konkurencyjność
val x = Future { someExpensiveComputation() }
val y = Future { someOtherExpensiveComputation() }
val z = for (a <- x; b <- y) yield a * b

for (c <- z) println("Result: " + c)
println("Meanwhile, the main thread goes on!")

// Traits - interfejsy poszerzone o funkcjonalność
abstract class Spacecraft {
    def engage(): Unit
}

trait CommandoBridge extends Spacecraft {
    def engage(): Unit = {
        for (_ <- 1 to 3)
            speedUp()
    }

    def speedUp(): Unit
}

trait PulseEngine extends Spacecraft {
    val maxPulse: Int
    var currentPulse: Int = 0

    def speedUp(): Unit = {
        if (currentPulse < maxPulse)
            currentPulse += 1
    }
}

```

```

    }
}

class StarCruiser extends Spacecraft
    with CommandoBridge
    with PulseEngine {
    val maxPulse = 200
}

// Pattern matching
// Define a set of case classes for representing binary trees.
sealed abstract class Tree
case class Node(elem: Int, left: Tree, right: Tree) extends Tree
case object Leaf extends Tree

// Return the in-order traversal sequence of a given tree.
def inOrder(t: Tree): List[Int] = t match {
    case Node(e, l, r) => inOrder(l) ::: List(e) ::: inOrder(r)
    case Leaf          => List()
}

// Higher-Order functions
val people: Array[Person]
// Partition `people` into two arrays `minors` and `adults`.
// Use the anonymous function `(_.age < 18)` as a predicate for partitioning.
val (minors, adults) = people.partition(_.age < 18)

```

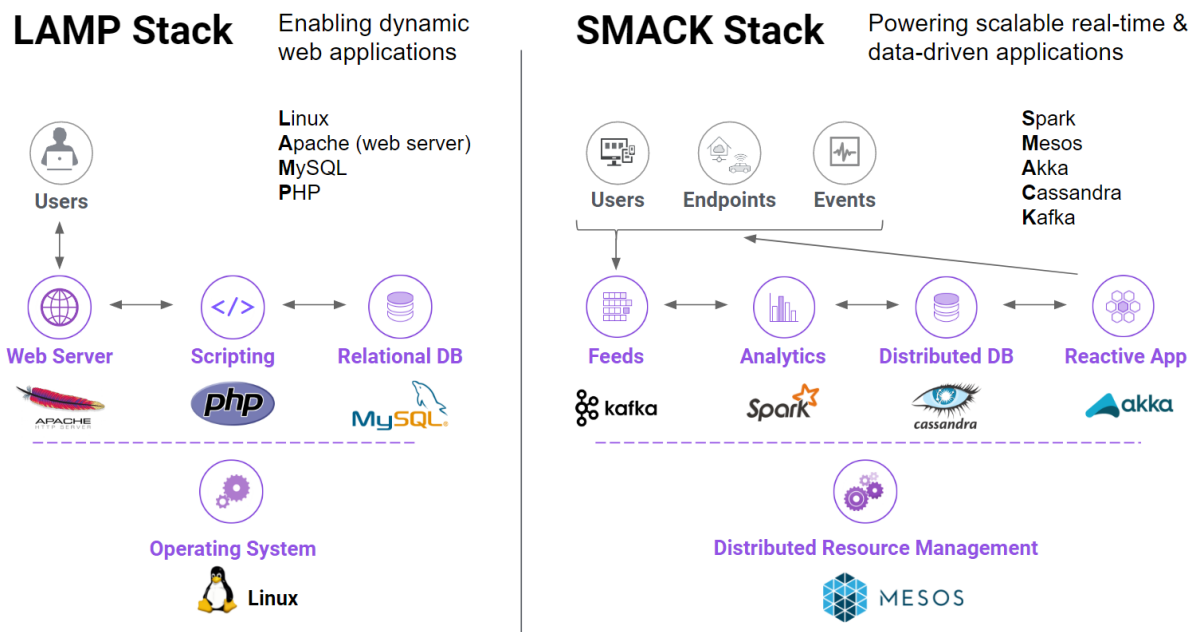
Źródło: *The Scala Programming Language* [online] École Polytechnique Fédérale Lausanne (EPFL) [dostęp: 26.10.2019]. Dostępny w Internecie: <https://docs.scala-lang.org/tour/tour-of-scala.html>

## 2.2 Lagom Framework (Play Framework oparty o Akka)

Emergencja Internetu w formie Web 2.0 wprowadziła postęp w interakcjach pomiędzy ludźmi. Dzięki oprogramowaniu uzyskano udostępnianie treści, komunikację, zakupy, dzielenie się wiedzą i inne. Pozwoliło to biznesom na dotarcie do klienta w niespotykanej dotąd łatwości, m. in. poprzez personalizowane oferty czy wymienianie się pomysłami – technologia *LAMP stack* umożliwiała wtedy dynamiczne generowanie stron internetowych z treścią dodawaną przez użytkowników. Współczesne największe przedsiębiorstwa stają przed wyzwaniem zawsze podłączonej ekonomii, gdzie to użytkownicy generuje dane czasu rzeczywistego, które poddawane są analizie predykcyjnej, w tym machine learningowi. Standardowym są personalizacja w czasie

rzeczywistym, detekcja anomalii, analityka danych czy internet rzeczy, które są przetwarzane, jako Fast Data. Sprostanie nowym wyzwaniom wymaga zastosowania nowych architektur technologicznych, takich jak *SMACK stack*, definiujący podstawowe elementy dla nowoczesnej aplikacji biznesowej typu *data-driven*, czyli napędzanej danymi (Rys.12).<sup>43</sup>

Rys.12. Dawny LAMP stack, nowy SMACK stack dla aplikacji biznesowych



Źródło: *The SMACK Stack is the New LAMP Stack* [online] Edward Hsu, D2IQ [dostęp: 29.10.2019]. Dostępny w Internecie: <https://d2iq.com/blog/smack-stack-new-lamp-stack>

Lightbend, propagatorzy oraz twórcy rozwiązań systemów reaktywnych (Scala, Akka, Play itd.) będąc w centrum przemian aplikacji biznesowych w kierunku Fast Data zaproponowali ewolucje swoich dwóch produktów: Akka oraz Play Framework, w dedykowane narzędzie dla tworzenia aplikacji w architekturze SMACK na reaktywnych mikroservisach z Cassandra, Kafką oraz Akka Streams – jest to Lagom Framework.<sup>44</sup>

Akka, czyli narzędzie wykorzystujące *message-driven Actor model* na JVM dla osiągnięcia konkurencyjności oraz elastyczności systemu rozproszonego stanowi podstawę dla budowy rozwiązań w tym paradygmacie osiągając niezawodny system

<sup>43</sup> *The SMACK Stack is the New LAMP Stack* [online] Edward Hsu, D2IQ [dostęp: 29.10.2019]. Dostępny w Internecie: <https://d2iq.com/blog/smack-stack-new-lamp-stack>

<sup>44</sup> *Lagom Framework – Part of Lightbend Platform* [online] Lightbend [dostęp: 29.10.2019]. Dostępny w Internecie: <https://www.lightbend.com/lagom-framework-part-of-lightbend-platform>

reaktywny. Natomiast Play Framework, bazując na Akka jest nowoczesnym dedykowanym frameworkiem webowym wysokiej prędkości (ang. *high velocity*), przeznaczonym dla produktywnego i wielce skalowalnego tworzenia aplikacji internetowych w metodyce *rapid development*, docenianym zarówno w środowisku start-up'owym, jaki i wysokowydajnościowym, w miejscach, gdzie standardowy Spring framework / JavaEE docierają do granicy swoich możliwości. Lagom to *microservices framework*, do tworzenia mikroservisów na potrzeby systemów rozproszonych. Budowanie systemu reaktywnego jest trudnym zadaniem, Lagom zapewnia abstrakcję upraszczając to, co złożone, a Akka oraz Play zapewniają środowisko wykonawcze, tak, aby inżynier oprogramowania mógł skupić się na prostym implementowaniu scenariuszy biznesowych w architekturze *event-driven* (pol. napędzane zdarzeniami) uzyskując wszelkie zalety modelu *message-driven* bez ręcznego opracowywania architektury.<sup>45</sup>

Wg autorów rozwiązania, istniejące frameworki mikroservisowe skupiają się na tworzeniu nietrwałych, jednoinstancyjnych mikroservisów, które nie są skalowalne ani odporne na błędy – Lagom buduje mikroservisy, jako system reaktywny, dzięki temu tworzone mikroservisy zyskują na elastyczności, odporności, a w efekcie na responsywności aplikacji. Technologie zawarte we frameworku są sprawdzone w zastosowaniach produkcyjnych od ponad dekady. Zastosowane rozwiązania są opiniowane przez najlepsze praktyki oraz poradniki producenta na bazie doświadczeń wdrożeniowych. Podobnie jak Play, skupia się na prędkości wytwarzania funkcjonalności, pozwala tworzyć nowe systemy w szybkim tempie. Zapewnia asynchroniczność, przechowywanie danych skali Big Data oraz integruje się z istniejącymi rozwiązaniami kontenerowymi takimi, jak Docker i Kubernetes w środowisku platform chmurowych. Dostępny jest w języku Java oraz Scala.

### **2.3 Cassandra – nierelacyjna baza danych big data (NoSQL)**

Instytucje tzw. skali Internetu (ang. *web-scale*), takie jak Facebook wraz z potężnym przyrostem danych wymaganych do przetwarzania zetknęły się z problemem

---

<sup>45</sup> *Lagom - Microservices Framework* [online] Lightbend [dostęp: 30.10.2019]. Dostępny w Internecie: <https://www.lagomframework.com>

braku narzędzi, które byłyby w stanie nadążyć za potrzebą rozwoju ich aplikacji. *Search team* w Facebooku usiłował rozwiązać problem wyszukiwarki wiadomości, w którym skala objętości danych, ich przyrostu i ściśle wymagania cd. responsywności wyraźnie ukazały, że stworzenie nowego rozwiązania do przechowywania danych jest absolutnie niezbędne. Celem opracowanego narzędzia miało być zastosowanie nie tylko w problemie wyszukiwarki wiadomości, ale również w wielu innych podobnych problemach skali danych. Tak powstała kolumnowa, rozproszona, nierelacyjna baza danych (NoSQL) dla przechowywania dużych wolumenów danych pomiędzy serwerami, nazwana później Cassandra.<sup>46</sup>

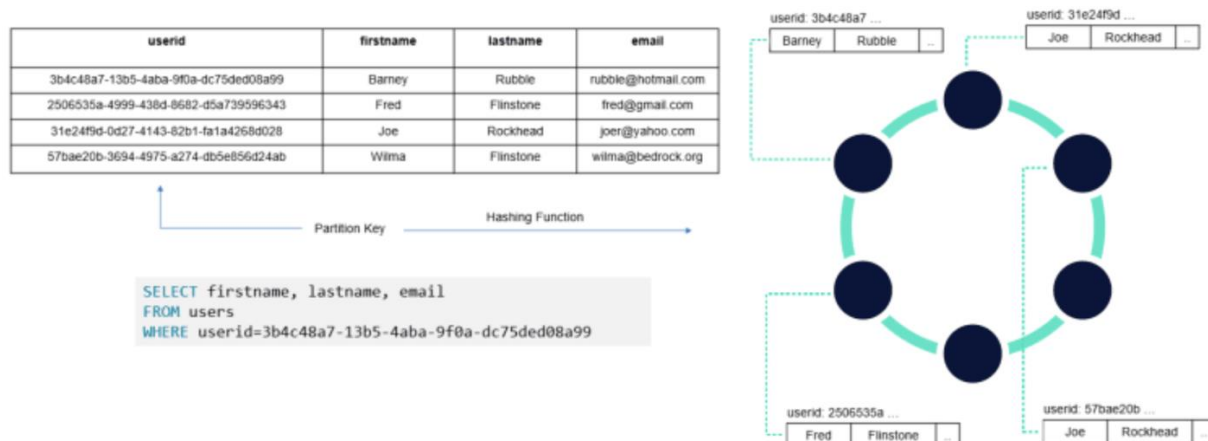
Cassandra to rozproszony system przechowywania ustrukturyzowanych danych, zaprojektowany do prostego horyzontalnego (oraz liniowego) skalowania na wiele serwerów, bez pojedynczego punktu awarii. Oryginalnie opracowana dla wysokiej dostępności (wg CAP Theorem), Cassandra przeznaczona jest do funkcjonowania w pierścieniu setek węzłów przechowujących dane, komunikujących się w postaci sieci P2P, z wbudowanym mechanizmem obsługi awarii. Zamiast stosowania modelu wierszowego znanego z relacyjnych baz danych RDBMS, wykorzystywany jest tzw. *wide column store*, który swoją charakterystyką zbliżony jest do formy nierelacyjnych baz danych *key-value*, lecz w postaci dwuwymiarowej. Wiersz definiowany jest poprzez kolekcję dowolnych kolumn, zawartych w zbiorach określanych jako *column family* (tablice) – kolumna w wierszu natomiast przedstawia konkretną wartość, indeksowaną poprzez wiersz, nazwę kolumny oraz datę. Ścisła różnica pomiędzy standardową bazą kolumnową, a *wide column* wynika, z bliskiego przechowywania danych kolumn zależnych między sobą w postaci *column families*, gdy natomiast baza kolumnowa przechowuje obok siebie wszelkie dane z wierszy, podobnie jak wierszowa wszystkie dane wiersza z kolumn tabeli. Jest to szczególnie przydatna cecha w zastosowaniach typu *business intelligence*, gdzie pozyskiwane są konkretne informacje z kolumn, a pozostałe dane z wiersza są zbędne i wpływają jedynie spadkowo na wydajność zapytań. Dla ustandaryzowanego i wygodnego pozyskiwania danych, Cassandra opracowała własną wersję języka zapytań CQL, która zbliżona jest do powszechnie znanego SQL. Dane wstawiane do Cassandra są następnie przechowywane w

---

<sup>46</sup> *Cassandra – A structured storage system on a P2P Network* [online] Facebook Engineering [dostęp: 08.11.2019]. Dostępny w Internecie: <https://www.facebook.com/notes/facebook-engineering/cassandra-a-structured-storage-system-on-a-p2p-network/24413138919/>

pierścieniu wg rozkładu funkcji hashującej, która określa odpowiednie miejsce składowania danych w węzłach (Rys.13).<sup>47</sup>

Rys.13. Przechowywanie danych w pierścieniu węzłów Cassandra



Źródło: *Five Steps to an Awesome Data Model in Apache Cassandra* [online] Scotch.io [dostęp: 09.11.2019]. Dostępny w Internecie: <https://scotch.io/tutorials/five-steps-to-an-awesome-data-model-in-apache-cassandra>

Rozdystrybuowanie danych w pierścieniu jest zarządzane poprzez *Consistent Hashing* oraz funkcję hashującą zachowującą kolejność (*Order Preserving Hash*), odpowiada to za przewidywalny rozkład danych w pierścieniu, możliwie proporcjonalnie pomiędzy węzłami. Nadzór nad nimi w klastrze spełnia tzw. *Gossip style membership algorithm*, który komunikuje się pomiędzy węzłami i weryfikuje, czy nie wystąpiły awarie. Wysoka dostępność Cassandra polega na mechanizmie replikacji pomiędzy serwerami, który jest definiowalny i domyślnie wynosi 3 kopie danych na różnych węzłach, zwiększając przepustowość odczytu. Architektura oparta o *Order Preserving Hash* oraz horyzontalne skalowanie z nowymi węzłami pierścienia, zapewnia partycjonowanie danych z przewidywalnymi lokacjami, otrzymując wysoką wydajność zapisu. W scenariuszu strumieniowych systemów danych (Fast Data) stosowanie narzędzia bazodanowego wysokiej dostępności rokuje problemy otrzymania najnowszych danych z bazy, znacznie bardziej preferowane byłoby użycie bazy wysokiej spójności. Z powodu tego Cassandra oferuje dostosowywalny poziom spójności z definiowalnym poziomem dla osobno: odczytów i zapisów, od opcji „zapisy nigdy nie kończą się błędem” do „zablokuj dostęp do replik, przed migracją

<sup>47</sup> *Introduction to Apache Cassandra's Architecture* [online] Akhil Mehra [dostęp: 09.11.2019]. Dostępny w Internecie: <https://dzone.com/articles/introduction-apache-cassandras>



najnowszych danych” – jest to szczególnie istotne dla scenariuszy danych wysokiego bezpieczeństwa.<sup>48</sup>

Cassandra przeznaczona jest dla aplikacji, które nie mogą pozwolić sobie na utratę danych, nawet w wypadku awarii całych centrów serwerowych. Wysoka wydajność, liniowa skalowalność, brak pojedynczego punktu awarii, wysokie bezpieczeństwo i konfigurowalna spójność pozwoliła na ugruntowaniu jej pozycji w miejscu bazy danych dla największych możliwych operacyjnych zastosowań Big Data. Cechy te również doskonale sprawdziły się w wykorzystaniu Cassandra do przechowywania zdarzeń we wzorcach architektury *Event Sourcing* z *Command Query Responsibility Segregation*, jako trwały środek zapisu zdarzeń i odzyskiwania stanu aplikacji w czasie awarii. Wiele światowych firm ery Internetu wykorzystuje Cassandrę do świadczenia usług milionom swoich użytkowników, m. in.: Apple (75 tys. węzłów, 10 PB danych), Netflix (2500 węzłów, 420 TB danych, ponad 1 trylion zapytań dziennie), eBay (100 węzłów, 250 TB) i inne.

## 2.4 Spark – strumieniowe przetwarzanie danych big data

Powstanie scenariuszy Big Data pochodziło od przeładowanych narzutem tradycyjnych hurtowni danych (ang. *Data Warehousing*), dlatego pierwotnie obrany kierunek rozwoju obejmował narzędzia analityczne, przetwarzające w trybie wsadowym na zasadzie ETL. Wraz z rozwojem aplikacji skali Internetu, przetwarzanie ogromnych wolumenów danych wymagane było po stronie operacyjnej, powstały, więc wersje baz danych NoSQL przystosowane do działania operacyjnego – oba fronty rozwiązań Big Data zaczęły z czasem jednoczyć się w implementacjach we wspólnych scenariuszach wykorzystania (Rys.14).<sup>49</sup>

Rys.14. Scenariusze zastosowania Big Data operacyjnego razem z analitycznym

---

<sup>48</sup> Apache Cassandra – *Manage massive amounts of data, fast, without losing sleep* [online] The Apache Software Foundation [dostęp: 12.11.2019]. Dostępny w Internecie: <http://cassandra.apache.org/>

<sup>49</sup> *Big Data: Examples and Guidelines for the Enterprise Decision Maker* [online] MongoDB, Inc. [dostęp: 13.11.2019]. Dostępny w Internecie: <https://www.mongodb.com/collateral/big-data-examples-and-guidelines-enterprise-decision-maker>

	MongoDB	Hadoop or Spark
<b>eBay</b>	User data and metadata management for product catalog	User analysis for personalized search & recommendations
<b>China Eastern Airlines</b>	Data supporting flight search application	Calculate fares based on permutations of rules stored in MongoDB
<b>Orbitz</b>	Management of hotel data and pricing	Hotel segmentation to support building search facets
<b>Pearson</b>	Student identity and access control, content management of course materials	Student analytics to create adaptive learning programs
<b>Foursquare</b>	User data, check-ins, reviews, venue content management	User analysis, segmentation and personalization
<b>Tier 1 Investment Bank</b>	Tick data, quant analysis, distribution of reference data	Risk modeling, security and fraud detection
<b>Industrial Machinery Manufacturer</b>	Storage and real-time analytics of sensor data collected from connected vehicles	Preventive maintenance programs for fleet optimization. Monitoring of manufactured components in the field
<b>SFR</b>	Customer service applications accessed via online portals and call centers	Analysis of customer usage, devices & pricing to optimize plans

Źródło: *Big Data: Examples and Guidelines for the Enterprise Decision Maker* [online] MongoDB, Inc. [dostęp: 13.11.2019]. Dostępny w Internecie: <https://www.mongodb.com/collateral/big-data-examples-and-guidelines-enterprise-decision-maker>

Hadoop stanowi pierwotne narzędzie analitycznego Big Data, przechowując ogromne wolumeny danych na dyskach rozproszonych maszyn, do przetwarzania wsadowego w celu analizy historycznej, tak jak miało to miejsce przy hurtowniach danych. Spark natomiast zrezygnował z modelu wsadowego na korzyść strumieniowania (z wykorzystaniem pomniejszych wsadów, tzw. *microbatches*), a zamiast stosowania powolnej przestrzeni dyskowej wykorzystuje przechowywanie w pamięci operacyjnej RAM. Model ten jest bardziej zasobochłonny oraz kosztowny, lecz oferuje możliwości przetwarzania danych w czasie bliskim rzeczywistości, interaktywnie, iteracyjnie z szerszymi możliwościami zastosowania dla analityki danych, czy machine learningu. Oba rozwiązania nie wykluczają się nawzajem, dlatego często stosowane są wspólnie, gdzie Hadoop spełnia rolę analityki historycznej na największych zbiorach danych, o charakterze retrospektywnym, a Spark analizuje dane krytyczne, w których możliwości analityczne oraz tempo uzyskania wyników mają kluczowe znaczenie dla działania operacyjnego. W porównaniu do Hadoop, wg Apache Foundation: Apache Spark jest do 100 razy szybszy dla danych w RAM oraz 10 razy szybszy przy danych dyskowych. Interfejsy API umożliwiają wykorzystanie topowych języków data science w ramach Spark: Java, Scala, Python oraz R i SQL. Moduły narzędzia poszerzają możliwości analityczne o analizę danych grafowych (GraphX),

machine learning (Spark ML) czy o możliwości strumieniowania czasu rzeczywistego (Spark Streaming). Wdrożenie Spark możliwe jest na niezależnym klastrze, w chmurze, platformie Hadoop YARN czy Kubernetes. Z wieloma implementacjami integracji źródeł danych, m. in. HDFS, Apache HBase, Apache Cassandra, Apache Hive.<sup>50</sup>

*Spark Streaming* to biblioteka zawarta w Apache Spark skupiająca się w pełni na potrzebach przetwarzania strumieni danych czasu rzeczywistego Fast Data. Opracowana dla skalowalności, odporności na awarie, dedykowana jest dla interaktywnych aplikacji strumieniujących dane. Integruje dane z wielu źródeł, m. in. Kafka, Flume, HDFS, Kinesis przeprowadza operacje przetwarzania i analizy, zwracając dane wynikowe do dysków, baz danych oraz dashboardów analitycznych, prezentując możliwie najszybszy wgląd w istniejący strumień danych z działań operacyjnych. W ramach rozwiązania możliwym jest wykorzystanie elementów analityki opracowanych w macierzystym rozwiązaniu Spark, włączając w to integrowanie strumieni danych historycznych na nowych strumieniach danych, przetwarzanych przez Spark Streaming. Zbliżone API bazujące na operacjach strumieniowych dla analiz wsadowych ułatwia pracę z analizą strumieniową, są to operatory znane m. in. z *Reactive Streams*, aczkolwiek Spark nie implementuje tego standardu, a jedynie wykorzystuje jego koncepcje do wygodnej pracy z danymi.<sup>51</sup>

Architektura strumieniująca wymaga rozważenia wielu aspektów przy tranzycji z Big Data do Fast Data, przede wszystkim są to:

- *Latency* (pol. opóźnienie) – Jak niskie opóźnienie jest wymagane? Jakiego rodzaju zadań są przetwarzane?
- *Volume* (pol. objętość) – Jak duża objętość danych jest wymagana? Czy występuje złożone przetwarzanie zdarzeń?
- *Integration* (pol. integracja) – Które i jak? Zapewnienie interoperacyjności
- *Data processing* (pol. przetwarzanie danych) – Jakiego rodzaju danych? W formie wsadów czy pojedynczych zdarzeń?

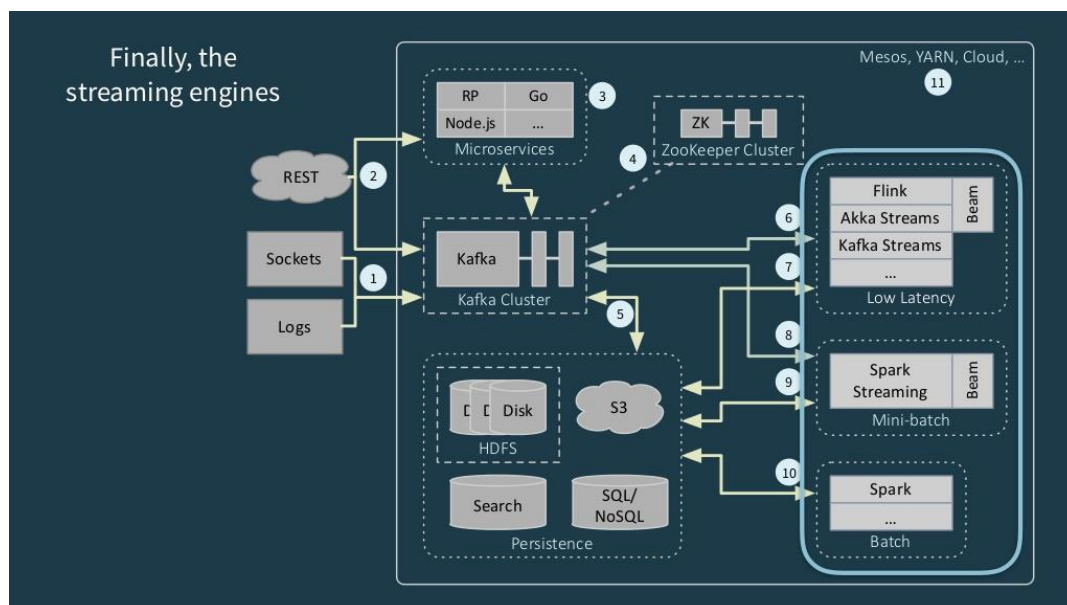
---

<sup>50</sup> *Apache Spark - Unified Analytics Engine for Big Data* [online] The Apache Software Foundation [dostęp: 14.11.2019]. Dostępny w Internecie: <https://spark.apache.org/>

<sup>51</sup> *Spark Streaming Programming Guide* [online] The Apache Software Foundation [dostęp: 14.11.2019]. Dostępny w Internecie: <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

W zależności od tych cech system Fast Data korzysta z różnorodnych narzędzi strumieniujących (Rys.15).<sup>52</sup>

Rys.15. Rodzaje silników strumieniujących wg *batch*, *mini-batch* i *low latency*



Źródło: *Moving from Big Data to Fast Data? Here's How To Pick The Right Streaming Engine* [online] Lightbend [dostęp: 22.11.2019]. Dostępny w Internecie: <https://www.lightbend.com/blog/moving-from-big-data-to-fast-data-heres-how-to-pick-the-right-streaming-engine>

W przypadku bardzo niskich opóźnień, mniej niż 1 mikrosekunda, wymagane jest opracowanie prototypowego sprzętu razem z implementacją w językach bliskich maszynom, takim jak C / C++. Przy mniej niż 100 mikrosekund sprawdzą się szybkie systemy przekazywania wiadomości, taki jak JVM Akka Actors. Dla 10 milisekund można zastosować już szybkie silniki strumieniujące: Akka Streams, Kafka Streams, Flink. Przy 1 sekundzie opóźnienia przetwarzanie strumieniowe na zasadzie micro-batch okaże się wystarczające, w postaci Spark Streaming. A kiedy opóźnienie przekracza sekundę wystarczy standardowe przetwarzanie wsadowe w Spark. Pod kątem objętości, gdy występuje mniej, niż 10 tys. zdarzeń / sekundę, wystarczającą jest komunikacja poprzez REST. Dla 100 tys. zdarzeń / sekundę komunikacja poprzez REST musi być nieblokująca, np. poprzez Akka Actors. Dla ponad 1 miliona zdarzeń na sekundę wymagany do przetwarzania jest Flink albo Apache Spark, najlepiej w

<sup>52</sup> *Moving from Big Data to Fast Data? Here's How To Pick The Right Streaming Engine* [online] Lightbend [dostęp: 22.11.2019]. Dostępny w Internecie: <https://www.lightbend.com/blog/moving-from-big-data-to-fast-data-heres-how-to-pick-the-right-streaming-engine>

postaci zagregowanej. Dobór odpowiedniej architektury nie jest łatwy i często wiąże się z rozważeniem wielu trudnych do przewidzenia ewentualności.

## 2.5 Spark ML – analiza danych machine learning

Różnorodne biblioteki analityczne Apache Spark przekładają się na szersze możliwości wykorzystania danych. W dobie trendu na zastosowania *machine learning*, szczególnie istotnym dla scenariuszy wywodzących się z Big Data jest możliwość trenowania oraz ewaluowania modeli uczenia maszynowego, które wymagają dużych zbiorów danych do osiągnięcia wysokich wyników predykcji – umożliwia to *Spark MLlib*.

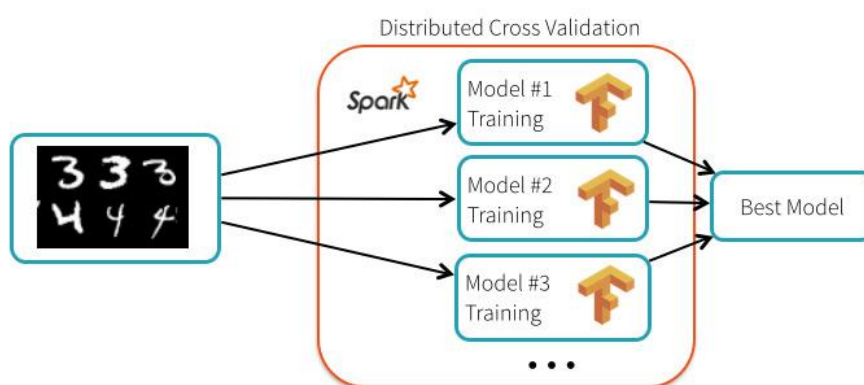
MLlib to biblioteka skalowalnego uczenia maszynowego dla platformy Spark. Integrująca się z rozwiązaniami z Python (m. in. numpy) oraz bibliotekami R, które są najszerszym ekosystemem data science oraz wiodącym motorem integracji rozwiązań z obszaru *machine learning*. MLlib zawiera szereg narzędzi: algorytmów uczenia maszynowego, elementy budowy przepływu danych, tradycyjne narzędzia statystyczne oraz inne. Adresuje problemy klasyfikacji, regresji, klasteryzacji, wykorzystując algorytmy takie, jak: regresja logistyczna, naiwny klasyfikator bayesowski, drzewa decyzyjne, k najbliższego sąsiada, k średnich itd. Umożliwia budowanie tzw. pipeline'ów machine learningowych, które są sekwencją zaimplementowanych operacji przetwarzających dane, tak, aby ich przepływ wytrenował odpowiedni model problemu albo przeprowadził ewaluację mechanizmu predykcji. Spark MLlib nadaje się zarówno do wsadowego, a także dla strumieniowego trenowania modeli w czasie rzeczywistym, które następnie ewaluuje się, jeżeli osiągnięta jest założona efektywność. Wadą narzędzia natomiast jest fakt, że stanowi bibliotekę uczenia maszynowego w postaci klasycznej, zbioru funkcji wywodzących się ze statystyki i algorytmów modeli matematycznych – nie posiada natomiast możliwości przeprowadzania uczenia *deep learning*, brak w niej nowoczesnych algorytmów sieci neuronowych.<sup>53</sup>

---

<sup>53</sup> *MLlib is Apache Spark's scalable machine learning library.* [online] The Apache Software Foundation [dostęp: 18.11.2019]. Dostępny w Internecie: <https://spark.apache.org/mllib/>

Przełomowe osiągnięcia w dziedzinie głębokiego uczenia zwróciło zainteresowanie branży od klasycznego machine learningu do głębokich sieci neuronowych. Data Scientist'ci chcieli wykorzystać możliwości nowoczesnych modeli sieci neuronowych ANN, CNN, RNN czy LSTM dla osiągnięcia lepszych wyników w złożonych problemach klasyfikacji, regresji, klasteryzacji itp. dla dużej objętości danych. Trening modeli *deep learning* nierzadko wiąże się z dużym narzutem obliczeniowym, który wymaga odpowiednio dużych zasobów. Apache Spark zapewnia platformę potrzebną do rozpraszania obliczeń, usprawniając proces uczenia wymagających wydajnościowo modeli. Powstało wiele implementacji open source, m. in.: *Elephas*, *TensorFlowOnSpark* od Yahoo, *dist-keras* od CERN, *BigDL* od Intel czy *spark-deep-learning* opracowany przez Databricks – skupiające się na zintegrowaniu Apache Spark z bibliotekami TensorFlow oraz Keras, które są wiodące w dziedzinie *deep learning*.<sup>54</sup> Rozpraszanie procesu trenowania modeli w TensorFlow, który z założenia jest wyłącznie jednoinstancjowy, ukazały wzrost wydajności i spadek ilości błędów predykcji w porównaniu do klasycznego, nieklastrowanego treningu na pojedynczej maszynie (Rys.16).<sup>55</sup>

Rys.16. Rozproszony trening modeli TensorFlow oparty o Apache Spark



<sup>54</sup> *Deep Learning With Apache Spark — Part 1* [online] Favio Vázquez [dostęp: 22.11.2019]. Dostępny w Internecie: <https://towardsdatascience.com/deep-learning-with-apache-spark-part-1-6d397c16abd>

<sup>55</sup> *Deep Learning with Apache Spark and TensorFlow* [online] Databricks [dostęp: 22.11.2019]. Dostępny w Internecie: <https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html>

Źródło: *Deep Learning with Apache Spark and TensorFlow* [online] Databricks [dostęp: 22.11.2019]. Dostępny w Internecie: <https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html>

Podobnie jak środowisko chmurowe ewaluujące kontenerowe implementacje oprogramowania posiada swojego orkiestratora w postaci Kubernetes, tak samo, aby wpasować się w istniejący ekosystem kontenerowy (również mikroserwisowy), powstało narzędzie *Kubeflow*, które odpowiada za wdrażanie mechanizmów Machine Learning w systemach cloud-native. Rozwiązanie pozwala na tworzenie pipeline'ów ML, przystosowanych do treningu oraz ewaluacji modeli, stworzonych w narzędziu TensorFlow, ale prowadzone są również prace nad integracją PyTorch, MXNet, Nuclio i innych. Kubeflow został upubliczniony open source dzięki Google, bazując na mechanizmach ich wewnętrznej obsługi pipeline'ów uczenia maszynowego na TensorFlow Extended, z których ewoluował do narzędzia niezależnego od architektury czy dostawcy chmurowego, dla kompletnej obsługi produkcyjnych wdrożeń ML. W ramach rozwiązania, obsługiwane jest: Jupyter Notebooks, orkiestracja przepływu danych, budowa, trening oraz wdrożenia modeli, dostosowywanie parametrów sieci neuronowych oraz ewaluacja modeli – skalowanie, wraz z monitorowaniem.<sup>56</sup>

## 2.6 Kafka – asynchroniczny broker wiadomości

Wzrost skali działania oprogramowania wymuszał przemysł na nowo architektury tak, aby możliwym było jej rozproszone działanie. Monolityczne aplikacje wykorzystywały osobno rozwijane usługi, które wykonywały kluczowe zdania takie, jak np. przechowywanie plików czy transakcyjną obsługę płatności. Sercem funkcjonowania niezależnych komponentów systemu jest ich możliwość niezawodnej komunikacji. Podejście *message-driven* zapewnia luźne powiązania pomiędzy producentami, a konsumentami wiadomości w systemie, wymienianymi poprzez brokerów wiadomości (ang. *message brokers*). W architekturze wykorzystującej usługi ten komponent komunikacyjny określa się, jako *service bus* – wymienia on wiadomości w charakterze komend (*command bus*), zapytań (*query bus*), czy też zdarzeń (*event*

---

<sup>56</sup> *Industrializing AI & Machine Learning Applications with Kubeflow* [online] Tianxiang (Ivan) Liu [dostęp: 22.11.2019]. Dostępny w Internecie: <https://towardsdatascience.com/industrializing-ai-machine-learning-applications-with-kubeflow-5687bf56153f>

*bus*)<sup>57</sup> – zapewniając możliwości działania systemu w czasie rzeczywistym, pomimo rozproszenia jego komponentów. Stanowi to podstawową cechę systemu reaktywnego.

Apache Kafka to *distributed streaming platform* (pol. rozproszona platforma strumieniująca) obsługująca przekazywanie wiadomości w modelu *Publish / Subscribe* (od producentów do konsumentów wiadomości) zapewniając mechanizmy przetwarzania strumieni (Kafka Streams) oraz przechowywania logu wiadomości. Kafka wykorzystywana jest do tworzenia pipeline'ów danych czasu rzeczywistego oraz aplikacji strumieniujących, będąc horyzontalnie skalowalna w klastrze instancji brokerów, odporna na błędy i wysoce wydajna, sprawdzona produkcyjnie w oprogramowaniu ponad tysiąca instytucji.<sup>58</sup>

Tradycyjne systemy przekazywania wiadomości obsługiwały dwa paradygmaty: *queue* (pol. kolejka) oraz *publish-subscribe*. W kolejce wiadomości były przechowywane do odczytania przez pierwszego lepszego konsumenta, po czym wiadomość ginęła bez możliwości poinformowania innych konsumentów. A *publish-subscribe* pozwalała przekazać wiadomość wielu konsumentom, lecz bez możliwości skalowania procesowania, gdyż każda wiadomość jest przekazana każdemu z subskrybentów. Apache Kafka unifikuje oba modele poprzez wykorzystanie tzw. *consumer group* (pol. grupa konsumentów), podobnie jak w przypadku kolejkowania, wiadomości mogą zostać podzielone poprzez konsumentów zawartych w grupie skalując procesowanie, a także równocześnie poinformować wielu konsumentów o zdarzeniu w myśl *publish-subscribe*, zawartych w różnych grupach konsumentów. Kafka gwarantuje również kolejność przekazywanych wiadomości przy asynchronicznej komunikacji pomiędzy brokerem, a konsumentem, eliminując problem będący przy kolejce, gdy wielu konsumentów asynchronicznie pochłania wiadomości z paralelizacją, tracąc faktyczną kolejność wynikającą z przepływu wiadomości. Wykonywane jest to za pomocą mechanizmu *partition* strumienia wiadomości z *topics* (kolekcja wiadomości), dla każdego konsumenta w *consumer group* – każdy z konsumentów posiada własną partycję wiadomości z *topics*, dzięki czemu to broker

---

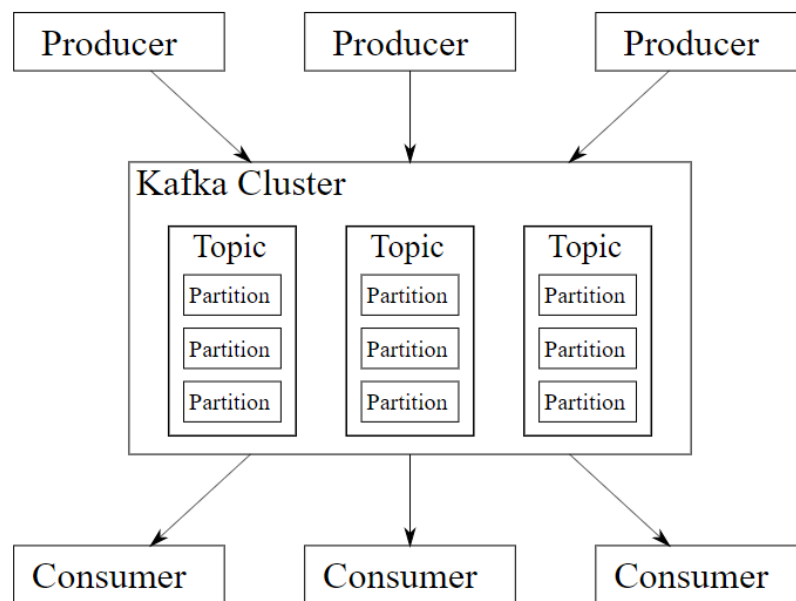
<sup>57</sup> *Different kinds of service bus: command bus, service bus and query bus*. [online] Barry van Veen [dostęp: 01.12.2019]. Dostępny w Internecie: <https://barryvanveen.nl/blog/59-different-kinds-of-service-bus-command-bus-service-bus-and-query-bus>

<sup>58</sup> *Apache Kafka – a distributed streaming platform* [online] Apache Software Foundation [dostęp: 02.12.2019]. Dostępny w Internecie: <https://kafka.apache.org/>



wiadomości odpowiada za paralelizację konsumpcji wiadomości z kolejki w ramach *consumer group* oraz load balancing dla każdego konsumenta z osobna (Rys.17). Skalowalność horyzontalna osiągnięta jest dzięki wielu instancjom brokerów Kafki zawartych w klastrze, zarządzanym poprzez ZooKeeper, orkiestrator klasy Big Data. Każda z instancji przyjmuje rolę lidera albo follower'a, dla danej partycji danych w *topic*. Follower jest zreplikowaną kolekcją wiadomości, która w wypadku awarii przejmie funkcję lidera. Natomiast lider odpowiada za faktyczne operacje odczytu/zapisu na partycji. Dane zapisywane są bezpośrednio na dysku w sposób wydajny, z niską złożonością obliczeniową. Kompletny zapis otrzymanych wiadomości od producentów zachowywany jest w *commit log*, w postaci identyfikatora rekordu oraz treści wiadomości, bez możliwości modyfikowania czy usuwania raz dodanej zawartości. Istotnym również jest, że Apache Kafka gwarantuje *at-least-once delivery*, dając pewność niezawodnego dostarczenia wiadomości w systemie rozproszonym.<sup>59</sup>

Rys.17. Ogólna architektura Publish / Subscribe w Apache Kafka



Źródło: *Overview of Apache Kafka* [online] Wikipedia [dostęp: 02.12.2019]. Dostępny w Internecie: [https://en.wikipedia.org/wiki/Apache\\_Kafka](https://en.wikipedia.org/wiki/Apache_Kafka)

Komponent działający niezależnie w systemie opartym o komunikację *message-passing* wykorzystuje mechanizm *Publish / Subscribe* Kafki poprzez implementację

<sup>59</sup> *Thorough Introduction to Apache Kafka™* [online] Stanislav Kozlovski [dostęp: 02.12.2019]. Dostępny w Internecie: <https://hackernoon.com/thorough-introduction-to-apache-kafka-6fbf2989bbc1>

klienckich API. Dodanie wiadomości odbywa się za pomocą *Producer API*, który w zależności od wykonywanej logiki biznesowej, strumieniuje wiadomości do zdefiniowanego *topic* w Kafce. Podobnie w przypadku konsumentów wiadomości, wykorzystując *Consumer API* pozwala to na odczyt wiadomości z *topics* Kafki, a następnie reaktywne podjęcie akcji po jej przyjęciu. Tak niezależnie związane komponenty uzyskują możliwość niezawodnej komunikacji czasu rzeczywistego w postaci Kafki, jako *message broker*, będącą krytycznym elementem działania systemu rozproszonego. Prócz podstawowego mechanizmu producentów i konsumentów wiadomości, zapewnione jest również *API Kafka Streams*, przystosowane do przetwarzania strumieniowych danych w czasie rzeczywistym od topiców wejściowych, poprzez procesowanie strumieniowe do topiców wyjściowych, wykorzystując mechanizmy strumieniowania i przechowywania Apache Kafka. Dodatkowo Kafka może integrować strumienie danych z innych źródeł za pomocą *Connector API*, takich, jak bazy danych, czy inne narzędzia strumieniowania. Kafka, jako *Storage* umożliwia skalowalne przechowywanie dużych wolumenów wiadomości, będąc przydatnym dla retencji danych w czasie, czy też stanowego przetwarzania strumieniowego, zastępując część scenariuszy wymagających dotychczas zewnętrznych baz danych.<sup>60</sup>

Dotychczasowo wiodącymi brokerami wiadomości były ActiveMQ, RabbitMQ będące kolejkami implementującymi protokół AMQP, pochodzący z takich mechanizmów przekazywania wiadomości, jak JMS (Java Message Service). Nadal stanowią one powszechny standard, czy to w architekturze wykorzystującej usługi, czy również, jako event bus w mikroservisach. Apache Kafka pochodząc ze scenariuszy rozmiaru Big Data, wprowadziła skalowalność oraz strumieniowanie, jako rozproszona platforma strumieniująca dla systemów będących *data-driven*, pozwalając osiągnąć im jeszcze większą skalę funkcjonowania czasu rzeczywistego, niezależnie od wielkości wolumenu danych. Spośród nowatorskich systemów *message-driven*, popularność zdobywa NATS, wysoce wydajny broker wiadomości *publish-subscribe*, który nie zapewnia przechowywania, ani przetwarzania strumieni czy odporności na awarie, natomiast w swojej prostocie cechuje się niską konsumpcją zasobów i dużą

---

<sup>60</sup> *Apache Kafka – documentation* [online] Apache Software Foundation [dostęp: 02.12.2019]. Dostępny w Internecie: <https://kafka.apache.org/documentation.html>

przepustowością nawet do 11 milionów wiadomości na sekundę, opracowany z myślą o środowisku cloud-native.<sup>61</sup>

## 2.7 Wzorce architektury mikroservisowej: CQRS, ES, DDD

Rosnąca złożoność oprogramowania skomplikowała funkcjonowanie oraz pracę zespołów nad produktem. Sukcesywnie rozrastające się implementacje zaczynały przełamywać tradycyjną architekturę projektów splatając ze sobą elementy, znacznie utrudniając dalszy rozwój dużych aplikacji – dążyło to do anty-wzorca, tzw. *big ball of mud*, w którym entropia projektu wzrastała w czasie do tak dużego poziomu, że nic nie było już proste, ani transparentne, a całość stanowiło niepoznaną dżunglę kodu *spaghetti-code*, niemożliwego do klarownego zrozumienia. Aby zapobiegać tym problemom powstały inicjatywy doświadczonych architektów oprogramowania, takie jak DDD (*Domain Driven Design*), CQRS (*Command Query Responsibility Segregation*) czy ES (*Event Sourcing*), w kierunku budowania czystej i skalowalnej architektury dla dużych i złożonych projektów.

*Domain Driven Design* to wzorec obejmujący nie tylko architekturę oprogramowania, ale przede wszystkim poprawną współpracę programistów z interesariuszami (np. klientami biznesowymi), zwanymi ekspertami domeny (*domain experts*). Główną ideą jest, że jako programiści dobrze znamy się na tworzeniu oprogramowania, gdy nasi klienci znają się najlepiej na swoim biznesie, dlatego też komunikacja z bezbłędnym obopólnym zrozumieniem jest niezbędna do osiągnięcia produktu zadowalającego obie strony. Komunikację tą osiąga się używając tzw. *ubiquitous language*, prostego i powszechnego języka, zrozumiałego dla inżynierów, jak i dla biznesu. Dyskusja nad funkcjonalnościami projektu odbywa się na zasadzie wymiany krótkich opisów działania, np. „*Kiedy osoba rejestruje się na kurs, wydawana jest zastrzeżona rejestracja. Jeśli dostępne jest miejsce i otrzymano płatność, rejestracja zastrzeżona jest akceptowana. Jeśli na kursie nie ma dostępnych miejsc, zarezerwowana rejestracja jest umieszczana na liście oczekujących, jako rejestracja rezerwowa. [...]*”.

---

<sup>61</sup> *Modern Open Source Messaging: NATS, RabbitMQ, Apache Kafka, hmbdc, Synapse, NSQ and Pulsar* [online] Philip Feng Ph.D [dostęp: 03.12.2019]. Dostępny w Internecie: <https://medium.com/@philipfeng/modern-open-source-messaging-apache-kafka-rabbitmq-nats-pulsar-and-nsq-ca3bf7422db5>

często zaczerpnięte jest to z BDD (*Behavior-Driven Development*), jako przykład domenowy, z przypadkiem użycia oraz rozmaitymi scenariuszami jego wykorzystania. Istotnym jest zagłębianie się w te konwersacje, rozwijając wątpliwości, razem z ciągłym poprawianiem opisu działania, naprowadzając na najbardziej prawidłowy dla jednej i drugiej strony model rozwiązania. Niezwykle owocną metodą pozyskiwania informacji o domenie biznesowej jest *Event Storming*, który działa podobnie do burzy mózgów ze skupieniem na zdarzeniach, które zachodzą w domenę – programiści i eksperci domenowi wspólnie zapisują zdarzenia, komendy, aktorów oraz agregaty, które występują w domenie na tablicy (najczęściej, jako karteczki samoprzylepne) – w efekcie szybko i prosto tworzy nam się obraz domeny, którą modelujemy w naszym rozwiązaniu, jako punkt odniesienia do architektury oraz dalszych dyskusji. DDD opiera się na koncepcji wzorca strategicznego domeny dla ogólnego dużego obrazu rozwiązania, oraz wielu pomniejszych wydzielonych kontekstów domenowych (*bounded contexts*), które skupiają się szczegółowo na procesach biznesowych. Domena może się składać z wielu wydzielonych kontekstów, które tworzą ze sobą mapy zależności, znane jest to, jako mapowanie kontekstu, i może zostać zamodelowane w różnorodne wzorce, np.: *shared kernel*, *customer / supplier*, *conformist*, *separate*, *anti-corruption layer* i inne. Kluczowym aspektem architektury w myśl *Domain Driven Design* jest prawidłowe zastosowanie elementów modelowych do rozwiązania domeny:

- *Entities* (pol. encje) – klasy, których instancje reprezentują globalnie unikalną tożsamość, niezależnie od zmian stanu, np. adres klienta może zmienić się wielokrotnie, ale tożsamość klienta nie ulega zmianie
- *Value Objects* – lekkie, niemodyfikowalne obiekty, zawierające atrybuty, ale nie posiadające tożsamości, np. gdy adres klienta zmienia się, instancjonuje się nowy *Address* value object, przypisywany do klienta
- *Services* (pol. usługi) – gdy alokacja danego działania nie jest możliwa dla jednej klasy, encji czy *value object*, posługujemy się usługą, bezstanową klasą, która odpowiada za funkcjonalność działającą na kilku klasach, gdy żadna z nich nie bierze za nią odpowiedzialności
- *Aggregates* (pol. agregaty) – dodawanie elementów do modelu sprawia, że graf obiektów staje się duży i złożony. Duże grafy obiektów stawiają wyzwania dla transakcyjności, rozproszenia oraz konkurencyjności i nie są pożądane. Agregaty są wydzielonymi granicami spójności, a klasy, które są

w nich zawarte są niezależne od reszty grafu obiektów. Każdy agregat posiada pojedynczą encję, która jest jego korzeniem (*aggregate root*).

- *Factories* (pol. fabryki) – fabryki zarządzają początkiem cyklu życia agregatów, zachowują przyjęte reguły biznesowe, w szczególności, co do tzw. *invariants*, które są warunkami integralności obiektów zawartych w agregatach
- *Repositories* (pol. repozytoria) – gdy fabryki zarządzają początkiem cyklu życia, repozytoria zajmują się środkiem oraz ich końcem. Delegują odpowiedzialności przechowywania do ORM-ów, które zwracają obiekty. Również funkcjonują z agregatami, co oznacza, że zwracane obiekty muszą spełniać reguły agregatu
- *Domain Events* (pol. zdarzenia domenowe) – zdarzenia pojawiające się, aby informować system w całości o fakcie, który wystąpił w jednej z jego części. Modelowane w formie, która pozwala na odtworzenie stanu encji z kolekcji zdarzeń domenowych. W systemach rozproszonych służą do komunikacji wewnętrznej pomiędzy elementami domeny

DDD skupia się na tworzeniu modelu domeny bogatej w funkcjonalność i elastycznej na zmiany. W kompletnej architekturze oprogramowania, domena musi być odseparowana od infrastruktury, np. w postaci architektury warstwowej – składającej się, z: Interfejsu Użytkownika, Aplikacji, Domeny oraz Infrastruktury, gdzie to wyższe warstwy architektury wykonują metody interfejsów niższych warstw, zachowując luźno związaną formę komunikacji pomiędzy warstwami, uniezależniając szczegóły implementacyjne wewnątrz nich od pozostałych. DDD, jako kolekcja wielu stosunkowo prostych wzorców, potrafi znacznie skomplikować pracę, jeśli zostanie błędnie zastosowana – wymaga to doświadczenia programistów, menedżerów oraz odpowiednich warunków biznesowych, aby uniknąć problemu *big ball of mud*. Jednakże wykonana dobrze, pozwala na wygodną pracę w przypadkach dużych oraz bardzo dużych projektów, dążących w kierunku aplikacji rozproszonych.<sup>62</sup> oraz <sup>63</sup>

*Command Query Responsibility Segregation*, czyli segregacja odpowiedzialności pomiędzy komendami, a zapytaniami rozdziela w systemie operacje zapisu (komendy)

---

<sup>62</sup> *Domain-Driven Design – Object-Orientation Done Right* [online] Aslam Khan, Obi Oberoi [dostęp: 04.12.2019]. Dostępny w Internecie: <https://dzone.com/refcardz/getting-started-domain-driven>

<sup>63</sup> *A Decade of DDD, CQRS and Event Sourcing* [online] tacta.io [dostęp: 04.12.2019]. Dostępny w Internecie: <https://tacta.io/a-decade-of-ddd-cqrs-and-event-sourcing/>

od odczytu (zapytania) tak, że są one wykonywane przez różne obiekty. Zastosowanie tego wzorca pozwala dalej na odseparowanie baz danych, na te odpowiedzialne za zapis (*write store*) oraz inne odpowiadające za odczyt (*read store*). Mogą one korzystać wtedy z różnych technologii bazodanowych, a także wydzielać się na inne w ramach *bounded contexts*, umożliwiając skalowalność stron odczytu niezależnie od strony zapisu. Zastosowanie CQRS zazwyczaj wiąże się z pojęciem zdarzeń domenowych (*events*), gdzie komendy wyrażają zamiar (np. *RegisterUser*), a zdarzenia reprezentują fakt, który miał miejsce w systemie (np. *UserRegistered*), z reguły komendy są synchroniczne, a zdarzenia asynchroniczne, podejście to dąży do architektury *event-driven*, aplikacji napędzanej zdarzeniami. W procesie funkcjonowania oprogramowania w myśl CQRS wywołania komend możliwe są z wielu API systemu, a informacje przekazywane są poprzez DTO (*Data Transfer Object*), które zawierają wymagane atrybuty. Następnie komendy wywoływane są na bazie atrybutów z DTO, a te przekazywane do *command handlers*, które je ewaluują. Efektem prawidłowego wykonania komendy jest wywołanie zdarzenia, które asynchronicznie rozpropagowuje się w systemie poprzez *event bus*, na które nasłuchują odpowiednie *event handlers*, reagując na zdarzenie w systemie. W tak opracowanej architekturze komendy obsługują podstawową logikę biznesową w ramach swojego *bounded context*, emitując zdarzenia dla reszty domeny, aby odpowiednio zareagowały na zmianę w systemie poza ich kontrolą, jest to forma inter-komunikacji między komponentami systemu. Wprowadza to charakterystykę systemu rozproszonego, gdzie zmiany przeprowadzone w jednym miejscu, dopiero po pewnym czasie zostaną przyjęte i wprowadzone w innych częściach systemu, nazywane jest to *Eventual Consistency*.<sup>64, 65, 66, 67</sup>

*Event Sourcing* zapewnia, że wszystkie zmiany stanu aplikacji są przechowywane, jako sekwencja zdarzeń. Zamiast przechowywania obecnego stanu aplikacji bezpośrednio w bazie danych, wczytując jego fragmenty oraz nadpisując

---

<sup>64</sup> *1 Year of Event Sourcing and CQRS* [online] Teiva Harsanyi [dostęp: 05.12.2019]. Dostępny w Internecie: <https://hackernoon.com/1-year-of-event-sourcing-and-cQRS-fb9033ccd1c6>

<sup>65</sup> *CQRS, Event Sourcing and DDD FAQ* [online] Edument AB [dostęp: 05.12.2019]. Dostępny w Internecie: <https://cqrs.nu/Faq>

<sup>66</sup> *Are CQRS commands part of the domain model?* [online] Vladimir Khorikov [dostęp: 05.12.2019]. Dostępny w Internecie: <https://enterprisecraftsmanship.com/posts/cQRS-commands-part-domain-model/>

<sup>67</sup> *Best Practices for Event-Driven Microservice Architecture* [online] Jason Skowronski [dostęp: 05.12.2019]. Dostępny w Internecie: <https://dzone.com/articles/best-practices-for-event-driven-microservice-archi>

poszczególnymi zmianami, zachowywana jest chronologicznie posortowana kolekcja zdarzeń. Na jej podstawie stan aplikacji jest rekonstruowany, zdarzenie po zdarzeniu. Przechowywane zdarzenia opisują nie tylko obecny stan agregatu domeny, ale również jak ten stan został osiągnięty. Możliwym jest odtworzenie dowolnego stanu aplikacji, poprzez rekonstrukcję stanu do określonego zdarzenia z pewnego punktu w czasie. Ewentualnie można wykorzystać ten mechanizm do poprawnego procesowania przeszłych nieprawidłowych zdarzeń, czy nadejścia opóźnionego zdarzenia. Zdarzenia w *Event Sourcing*, są niezmiennie po przechowaniu, nawet pomimo zmiany logiki biznesowej domeny. Log zdarzeń zachowuje, więc informację o przeszłych zdarzeniach, jakie zostały wykonane w systemie, bez dopuszczenia możliwości nadpisywania ich historii. Przechowywanie zdarzeń dla ES odbywa się w *event store*, który może być zwyczajną konstrukcją kolekcji przechowywanej w pamięci (*in-memory*), tradycyjną bazą danych (RDBMS) albo skalowalną bazą NoSQL, np. Apache Cassandra. Zdarzenia domenowe (*domain events*) w myśl DDD niekiedy dobrze sprawdzają się w scenariuszach zdarzeń ES, jednakże tworzone są one z myślą o informowaniu systemu o zajściu zdarzeń w niepowiązanych *bounded contexts*. Nie dla przechowywania stanu jednego agregatu, który może być odtworzony na podstawie logu zdarzeń. Dlatego też dla separacji odpowiedzialności nie powinny być one razem mieszane. *Event Sourcing* jest wzorcem szczególnie ważnym dla kluczowych elementów systemów rozproszonych, gdyż dzięki niemu możliwym staje się odtworzenie stanu systemu po awarii, bazując na logu zarejestrowanych zdarzeń, łatwo przywracając poprzedni spójny stan, bez utraty krytycznych danych produktu.<sup>68</sup> W przypadku Akka obsługiwane jest to poprzez *Akka Persistence*, który implementuje ES na poziomie pojedynczego aktora. Jeżeli aktor otrzyma komendę, która z nieokreślonego powodu spowoduje jego awarię, jego stan zostanie odtworzony na bazie zachowanych zdarzeń, wracając do pełnej operacyjności tak, jak miało to miejsce przed awarią.

Wzorce te przeplatają się między sobą i wiele z ich elementów jest niezależnie od siebie wykorzystywanych w oprogramowaniu. Kiedy i w jakim zakresie je stosować jest już elementem dyskusyjnym, natomiast z pewnością warto je rozważyć, jeśli projekt staje się tak duży, że jego rozwój zaczyna powodować problemy. Na przykład

---

<sup>68</sup> *Domain Events vs. Event Sourcing – Why domain events and event sourcing should not be mixed up* [online] Christian Stetter [dostęp: 06.12.2019]. Dostępny w Internecie: <https://www.innoq.com/en/blog/domain-events-versus-event-sourcing/>

CQRS i ES są uznawane za wzorce wsparcia, a nie główne wzorce projektowe, dlatego nie powinny być aplikowane do całości projektu. Często też jeden wzorzec implikuje drugi, tak jak np. *Event Sourcing* do zaimplementowania najczęściej wymaga zastosowania CQRS, natomiast nic nie stoi na przeszkodzie, żeby wprowadzić *Command Query Responsibility Segregation* bez ES. Wzorce te, podobnie jak *Domain Driven Design* nie są stosunkowo nowe, pochodzą od architektur monolitycznych, i w ramach rozwoju architektury mikroserwisowej są ponownie odkrywane, jako kluczowe w budowie systemów rozproszonych. Współcześnie DDD implementowane jest pod postacią CQRS oraz ES, dla systemów *event-driven* napędzanych zdarzeniami.

W rozdziale omówiono narzędzia składające się na architekturę systemów skalowalnych, czasu rzeczywistego oraz napędzanych danymi – wykorzystujących Fast Data oraz Machine Learning. Prócz szczegółów technicznych posiłkowano się również przypadkami użycia. Pokrótkę przedstawiono metody integracji pomiędzy komponentami. Zwrócono również uwagę na zastosowanie odpowiednich wzorców projektowych wspierających pracę nad produktem. Przedstawiony zestaw komponentów został wykorzystany w implementacji systemu reaktywnego, szerzej omówionego w następnym rozdziale.



### **3. Implementacja systemu reaktywnego strumieniującego dane Fast Data**

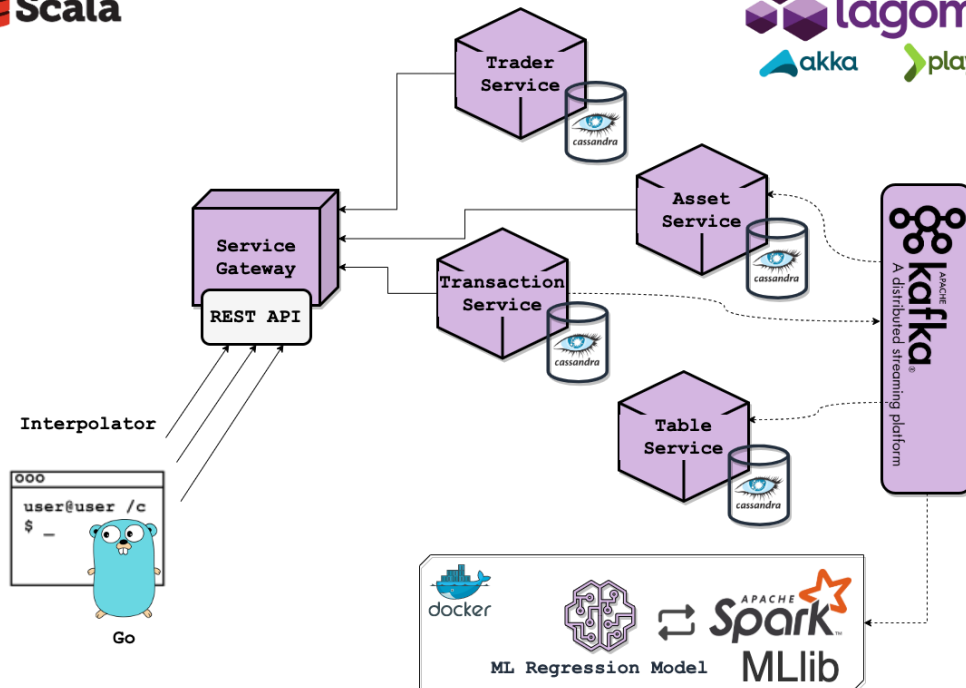
Niniejszy rozdział prezentuje opracowaną implementację systemu reaktywnego, strumieniującego dane Fast Data wraz z zastosowaną analizą predykcyjną Machine Learning. Omówione zostają poszczególne elementy systemu, od architektury rozwiązania, implementacji, przez interpolator po konstrukcję modelu predykcyjnego. Rozdział zawiera wiele szczegółowych listingów kodu, będących kluczowymi w całokształcie działania systemu oraz wyczerpuje punkty komunikacji odrębnych komponentów. Aspektem technicznym towarzyszy wysokopoziomowy opis działania prototypu wraz z drogą wykonania logiki systemu.

Badane rozwiązanie stanowi empiryczną formę weryfikacji osiągalności postawionych założeń w postaci analizy przypadku – giełdy papierów wartościowych, gdzie użytkownicy mogą dokonywać operacji nabycia i/lub sprzedaży zasobów. Implementacja jest prototypem (PoC, ang. *proof of concept*) kompletnego rozwiązania systemu reaktywnego, gdzie wybrane informacje istotne biznesowo – transakcje giełdowe – są danymi strumieniowanymi klasy Fast Data, na których przeprowadzana jest analiza czasu rzeczywistego z wykorzystaniem modelu regresyjnego Machine Learning. Ewaluacja implementacji ma charakter eksperymentu, a jej wyniki są gotowe do natychmiastowego wykorzystania operacyjnego.

#### **3.1 Architektura rozwiązania**

Kompletna architektura rozwiązania projektu *reactive-stock* składa się z elementów charakteryzujących systemy reaktywne oraz narzędzia ekosystemu Big Data (Rys.18).

Rys.18. Architektura rozwiązania projektu *reactive-stock*



System reaktywny – implementacja oprogramowania oparta o mikroserwisy, na podstawie Lagom Framework. Składa się z czterech serwisów, każdy z nich odpowiada za jego własny wydzielony kontekst:

- *Trader Service* (ang. usługa handlarzy) – odpowiada za rejestrację oraz autoryzację użytkowników, a także przechowywane informacji o posiadanych przez nich zasobach
- *Asset Service* (ang. usługa zasobów) – przechowuje kolekcje zasobów, będących w obrocie na giełdzie, oraz aktualizuje ich cenę
- *Transaction Service* (ang. usługa transakcji) – składanie, walidacja oraz przekazywanie transakcji giełdowej typu nabycie / sprzedaż
- *Table Service* (ang. usługa tabeli) – przetwarzanie transakcji giełdowych analogicznie do tabeli kupna sprzedaży, monitoruje wpływające akceptowane transakcje i wiąże nabywcę ze sprzedawcą, zamykając transakcje

Za ekspozycję zdefiniowanego w mikroservisach REST API odpowiada *Service Gateway*, jest to element Lagom Framework, w architekturze mikroservisowej pośredniczy on za komunikację pomiędzy klientami systemu (przeglądarka internetowa, aplikacja mobilna itp.), a poszczególnymi usługami, wystawiając ich odnośniki do świata zewnętrznego. Kryje się za tym dodatkowy mechanizm *Service Locator*, który

pozwala na odkrycie się poszczególnych mikroserwisów i komunikację pomiędzy nimi. Każdy z mikroserwisów wykorzystuje wzorce architektury CQRS oraz ES, a zdarzenia są przechowywane w logu zdarzeń bazy danych Cassandra, która jest instancjonowana niezależnie dla każdego z serwisów, dodatkowo spełnia ona rolę prezentowania strony odczytu w myśl *Query* z CQRS. Implementacja mikroserwisów została wykonana w języku Scala. Apache Kafka odpowiada za Event Bus, propagując wiadomości w systemie reaktywnym tak, aby zapewnić luźną, asynchroniczną komunikację pomiędzy serwisami, gwarantuje on dostarczenie zdarzeń systemu przynajmniej jednorazowo, otrzymujemy więc rozwiązanie systemu rozproszonego z *Eventual Consistency*. Równocześnie Apache Kafka stanowi platformę strumieniującą danych Fast Data w systemie – te natomiast będą wykorzystywane przez część analizy predykcyjnej.

Interpolator – część reprezentująca świat zewnętrzny wchodzący w interakcję z systemem, w scenariuszu rzeczywistym ta część byłaby przedstawiona w postaci interfejsu użytkownika UI, np. jako aplikacja webowa albo mobilna. Dla prostoty i wykazania celów bez angażowania ruchu rzeczywistego, formę interakcji użytkowników stanowi konsolowe narzędzie symulacyjne, opracowane w języku Golang. Wykonuje ono ruch w systemie wykonując zapytania na poszczególnych odnośnikach REST API, symulując w ten sposób faktyczne zachowania traderów, od rejestracji, zalogowania, po złożenie transakcji nabycia / sprzedaży, a następnie oczekiwania na rozwiązanie transakcji. Komponent ten określony jest jako interpolator, gdyż na etapie eksperymentu wykorzystuje on rzeczywiste dane archiwalne transakcji giełdowych, tak, aby odtworzyć w implementowanej giełdzie sytuację, która historycznie faktycznie wystąpiła w rzeczywistości, stąd też interpolacja przypadku z danych.

Analiza predykcyjna – wykorzystując Apache Spark MLlib przeprowadzane są procesy treningu, walidacji oraz ewaluacji modelu regresyjnego. Skonteneryzowane środowisko wirtualizowane jest za pomocą Docker, zawiera w sobie Apache Spark, Spark Streaming oraz Spark MLlib, a także skonstruowane pipeline'y modeli Machine Learningowych w języku Scala. Subskrybują one zdarzenia propagowane w systemie z Apache Kafka. Przeprowadzają na nich analizę danych czasu rzeczywistego i zwracają wynik natychmiastowo do systemu.

Całokształt rozwiązania składa się na rozproszony system reaktywny oparty na zdarzeniach, zbudowany na narzędziach Big Data, gotowych do operacyjnego obsługiwanie danych strumieniowanych Fast Data, zapewniających możliwie maksymalny poziom skalowalności, aż do poziomu *web-scale* (skali internetu). Analiza modelu uczenia maszynowego jest w nim przeprowadzana nieustannie i w czasie rzeczywistym, zwracając wartość biznesową płynącą z danych niemal od razu do systemu, umożliwiając nowe innowacyjne możliwości zastosowania podejścia *data-driven* (pol. napędzanych danymi). Jest to synteza strony operacyjnej oraz analitycznej w jednym rozwiązaniu.

### 3.2 Implementacja systemu reaktywnego

Logika biznesowa systemu reaktywnego zawarta jest w jego czterech mikroservisach, wykorzystują one bezpośrednie zapytania REST API do komunikacji ze światem zewnętrznym oraz subskrybują Event Bus do luźno powiązanej komunikacji pomiędzy sobą. Dodatkową formą komunikacji, ukrytą za warstwą abstrakcji jest komunikacja wewnętrzna serwisów pomiędzy swoimi tożsamymi instancjami, w ramach klastru aktorów Akka, wykorzystują one bezpośrednią komunikację przez *Akka Remoting*.

Punktem wiążącym budowę i kompilację całego systemu jest plik konfiguracyjny *build.sbt*, zawiera on kolekcję modułów projektu (mikroservisów) oraz ich wzajemne zależności w standardowym menadżerze projektów Scala – SBT (Listing nr 3.).

Listing nr 3. Fragment konfiguracji modułów projektu w *build.sbt*

```
organization in ThisBuild := "xarvalus"
version in ThisBuild := "0.1.0"

// ...

lazy val `reactive-stock` = (project in file("."))
  .aggregate(
    `common`,
    `asset-api`, `asset-impl`,
    `table-api`, `table-impl`, `table-stream-api`, `table-stream-impl`,
    `trader-api`, `trader-impl`,
```

```

    `transaction-api`, `transaction-impl`,
  )

  lazy val `asset-api` = (project in file("asset-api"))
    .settings(
      libraryDependencies ++= Seq(
        lagomScaladslApi
      )
    )
    .dependsOn(`common`)

  lazy val `asset-impl` = (project in file("asset-impl"))
    .enablePlugins(LagomScala)
    .settings(
      libraryDependencies ++= Seq(
        lagomScaladslPersistenceCassandra,
        lagomScaladslKafkaBroker,
        lagomScaladslTestKit,
        macwire,
        scalaTest
      )
    )
    .settings(lagomForkedTestSettings)
    .dependsOn(`common`, `asset-api`, `table-api`)

  // ...

```

Źródło: opracowanie własne

Pakiet *common* to współdzielony zestaw klas i metod wykorzystywanych powszechnie we wszystkich usługach systemu, zawiera m. in. logikę odnośnie weryfikacji klucza JWT używanego przy autoryzacji użytkowników, a także walidatory ogólnych błędów. Przy usłudze *Table Service* dodatkowo istnieje osobny mikroserwis *Table Stream Service*, który jest pochodny do oryginalnego serwisu – wyprowadza on dane odnośnie przetworzonych transakcji do świata zewnętrznego w postaci strumienia, obsługiwanego domyślnie przez protokół *WebSocket*.

Każda usługa jest konfigurowana w *application.conf* (Listing nr 4.). Zawiera m. in. wskazanie *Loadera*, który jest punktem wejściowym inicjalizującym usługę, a także konfigurację *Cassandry*, wskazanie *keyspace* i zdefiniowanie *ReadSide* w *Lagom*.

Dodatkowym elementem jest osobna serializacja dla komend w *akka.actor*, czyli *jackson-json*, wymagana ze względu na zmianę API od wersji Lagom 1.6.

#### Listing nr 4. Konfiguracja mikroserwisu *Transaction Service*

```
play.application.loader =
xarvalus.reactive.stock.transaction.impl.TransactionLoader

transaction.cassandra.keyspace = transaction

cassandra-journal.keyspace = ${transaction.cassandra.keyspace}
cassandra-snapshot-store.keyspace = ${transaction.cassandra.keyspace}
lagom.persistence.read-side.cassandra.keyspace =
${transaction.cassandra.keyspace}

akka.actor {
  serialization-bindings {

    "xarvalus.reactive.stock.transaction.impl.TransactionCommandSerializable"
    = jackson-json
  }
}
```

Źródło: opracowanie własne

Podstawową konwencją definiowania modułów projektu w DDD oraz architekturze mikroserwisowej, jest rozbijanie implementacji na interfejsy oraz klasy implementujące. Osiągnięte jest to w postaci dwóch modułów na mikroserwis: *api*, czyli interfejsu usługi, definiującego możliwe zapytania / kanały strumieni danych oraz *impl*, będącego faktyczną implementacją interfejsu. System identyfikuje mikroserwisy pomiędzy sobą za pomocą ich interfejsów *api*, co jest przedłużeniem jednej z głównych zasad pryncypiów SOLID. *Transaction Service* wykorzystuje moduł *transaction-api* (Listing nr 5.).

#### Listing nr 5. Interfejs (*trait*) *TransactionService* mikrousługi transakcji

```
object TransactionService {
  val TOPIC_NAME = "orders"
}

trait TransactionService extends Service {
  def order(): ServiceCall[Order, Done]
```

```

def ordersTopic(): Topic[PlacedOrder]

override final def descriptor: Descriptor = {
  import Service._
  // @formatter:off
  named("transaction-service")
    .withCalls(
      pathCall("/api/transaction/order", order _)
    )
    .withTopics(
      topic(TransactionService.TOPIC_NAME, ordersTopic _)
        .addProperty(
          KafkaProperties.partitionKeyStrategy,
          PartitionKeyStrategy[PlacedOrder](_.orderId)
        )
    )
    .withAutoAcl(true)
  // @formatter:on
}
}

sealed trait OrderType
sealed trait Buy extends OrderType
sealed trait Sell extends OrderType

// ...

case class Order(
  asset: String,
  price: BigDecimal,
  quantity: BigDecimal,
  orderType: OrderType
)

object Order {
  implicit val format: Format[Order] = Json.format
  implicit val orderValidator:
    ValidationTransform.TransformedValidator[Order] = validator[Order] {
    order =>
      order.asset is notEmpty
      order.price should be >= BigDecimal(0)
      order.quantity should be >= BigDecimal(0)
      order.orderType is notNull
    }
  }
}

```

```

case class PlacedOrder(
  orderId: String,
  asset: String,
  price: BigDecimal,
  quantity: BigDecimal,
  orderType: OrderType,
  user: UUID,
  timestamp: String
)

object PlacedOrder {
  implicit val format: Format[PlacedOrder] = Json.format
}

```

Źródło: opracowanie własne

Definiuje zapytanie *order* w postaci odnośnika „*/api/transaction/order*”, przyjmuje ono dane typu *Order*, a zwraca *Done*, czyli domyślną pustą odpowiedź. Zdefiniowany jest również *topic* Kafka o nazwie „*orders*”, emitujący wiadomości o typie *PlacedOrder*. Na zdefiniowanych klasach wprowadzone są transformatory do formatu JSON oraz walidacja za pomocą *Accord*.

Implementacja logiki definiowanej przez interfejs znajduje się w module *transaction-impl* (Listing nr 6.).

Listing nr 6. Implementacja *TransactionServiceImpl* mikrousługi transakcji

```

class TransactionServiceImpl(
  clusterSharding: ClusterSharding,
  persistentEntityRegistry: PersistentEntityRegistry
)(implicit ec: ExecutionContext)
  extends TransactionService {
  private def entityRef(id: String): EntityRef[TransactionCommand] =
    clusterSharding.entityRefFor(TransactionState.typeKey, id)

  implicit val timeout: Timeout = Timeout(5.seconds)

  override def order(): ServiceCall[OrderRequest, Done] = authenticated
  { (token, _) =>
    ServerServiceCall { request =>
      validate(request)
    }
  }
}

```



```

    val ref = entityRef(UUID.randomUUID().toString)
    Ref
      .ask[Confirmation](replyTo =>
        PlaceOrder(
          asset = request.asset,
          price = request.price,
          quantity = request.quantity,
          orderType = request.orderType,
          user = token.userId,
          replyTo
        ))
      .map {
        case Accepted => Done
        case _         => throw BadRequest("Cannot place order")
      }
  }
}

override def ordersTopic(): Topic[PlacedOrder] =
  TopicProducer.singleStreamWithOffset { fromOffset =>
    persistentEntityRegistry
      .eventStream(TransactionEvent.Tag, fromOffset)
      .map(ev => (convertEvent(ev), ev.offset))
  }

private def convertEvent(
  transactionEvent: EventStreamElement[TransactionEvent]
): PlacedOrder = {
  transactionEvent.event match {
    case OrderPlaced(orderId, asset, price, quantity, orderType,
user, timestamp) =>
      PlacedOrder(orderId.toString, asset, price, quantity,
orderType, user, timestamp)
  }
}

```

Źródło: opracowanie własne

Zapytanie *order* o stworzenie transakcji, po autoryzacji użytkownika oraz walidacji zawartości, tworzy *entityId*, reprezentujący konkretny *aggregate*, następnie poprzez *ref* do encji wykonuje się zapytanie typu *ask*, czyli wykonanie komendy oraz pozyskanie odpowiedzi. W tym przypadku jest to komenda *PlaceOrder* – czyli złożenie transakcji. Za logikę egzekwowania komendy oraz wyemitowanie odpowiedniego zdarzenia, wraz

z jego przetworzeniem odpowiada *TransactionAggregate*, i zdefiniowany w nim *TransactionBehavior* (Listing nr 7.).

Listing nr 7. Zawartość *TransactionAggregate*, *Behavior* oraz *State*

```
object TransactionBehavior {
  def create(entityContext: EntityContext[TransactionCommand]):
    Behavior[TransactionCommand] = {
    // ...

    private[impl] def create(persistenceId: PersistenceId, entityId:
    String) = EventSourcedBehavior
      .withEnforcedReplies[TransactionCommand, TransactionEvent,
    TransactionState](
        persistenceId = persistenceId,
        emptyState = TransactionState.initial,
        commandHandler = (state, cmd) => state.applyCommand(cmd,
    entityId),
        eventHandler = (state, evt) => state.applyEvent(evt)
      )
  }

  case class TransactionState(order: Option[Order]) {
    def applyCommand(cmd: TransactionCommand, entityId: String):
    ReplyEffect[TransactionEvent, TransactionState] =
      cmd match {
        case PlaceOrder(asset, price, quantity, orderType, user,
    replyTo) =>
          Effect
            .persist(
              OrderPlaced(
                orderId = UUID.fromString(entityId),
                asset,
                price,
                quantity,
                orderType,
                user,
                timestamp = LocalDateTime.now().toString
              )
            )
            .thenReply(replyTo) { _ =>
              Accepted
            }
      }
  }

  def applyEvent(evt: TransactionEvent): TransactionState =
```

```

    evt match {
      case OrderPlaced(_, asset, price, quantity, orderType, user,
_) =>
        TransactionState(
          Some(Order(
            asset,
            price,
            quantity,
            orderType,
            user
          ))
        )
    }
  }

  // ...

  case class OrderPlaced(
    orderId: UUID,
    asset: String,
    price: BigDecimal,
    quantity: BigDecimal,
    orderType: OrderType,
    user: UUID,
    timestamp: String
  ) extends TransactionEvent

  case class PlaceOrder(
    asset: String,
    price: BigDecimal,
    quantity: BigDecimal,
    orderType: OrderType,
    user: UUID,
    replyTo: ActorRef[Confirmation]
  ) extends TransactionCommand

```

Źródło: opracowanie własne

*TransactionState* przedstawia stan *agregate* dla danej encji wg jej ID, poprzez *TransactionBehavior* inicjalizowany jest pusty stan, handlersy komend i zdarzeń, oraz dane komendy i zdarzenia w postaci *PlaceOrder* (komenda), oraz *OrderPlaced* (zdarzenie). Gdy handler komendy konstruuje i przekazuje do przechowania zdarzenie, handler zdarzeń propaguje z zapisanej komendy zmianę stanu na stan danego agregatu.

W przypadku mikroservisu Trader oraz Asset logika ta jest zawarta w klasie Entity, która jest wersją API wykorzystywaną przed Lagom 1.6.

Oprócz zdefiniowania API oraz logiki przetwarzania komend i zdarzeń, przy scenariuszach pozyskiwania informacji wymagane jest skonstruowanie i aktualizacja strony odczytu, tak jak w przypadku *AssetService*, wykorzystuje się do tego celu *EventProcessor* oraz *Repository*, pierwsze propaguje zmianę stanu (wynikającą ze zdarzenia) na bazodanową stronę odczytu, gdy drugie zwraca dane z kolekcji w bazie (Listing nr 8.).

Listing nr 8. Implementacja *AssetEventProcessor* oraz *AssetRepository*

```
class AssetEventProcessor(  
  session: CassandraSession,  
  readSide: CassandraReadSide  
) (implicit ec: ExecutionContext)  
  extends ReadSideProcessor[AssetEvent] {  
    override def buildHandler(): ReadSideHandler[AssetEvent] = {  
      readSide.builder[AssetEvent]("AssetEventOffset")  
        .setGlobalPrepare(createTable)  
        .setPrepare { _ =>  
          prepareStatements()  
        }.setEventHandler[PriceUpdated](insertAsset)  
        .build()  
    }  
  
    // ...  
  
    private def createTable(): Future[Done] = {  
      session.executeCreateTable(  
        "CREATE TABLE IF NOT EXISTS assets(" +  
          "id          uuid," +  
          "asset       varchar," +  
          "price       decimal," +  
          "updated_at timestamp, PRIMARY KEY (id)" +  
        ");"  
      )  
    }  
  
    private def prepareStatements(): Future[Done] = {  
      session  
        .prepare("INSERT INTO assets(id, asset, price, updated_at)  
VALUES (?, ?, ?, ?)")
```

```

        .map(insertAsset => {
            insertAssetStatement = insertAsset
            Done
        })
    }

// ...

class AssetRepository(
    db: CassandraSession
)(implicit ec: ExecutionContext) {
    def findAllAssets(): Future[Option[Assets]] = {
        val assets = db.selectAll("SELECT id, asset, price, updated_at
FROM assets")
        .map { rows =>
            val names = rows.map(row => row.getString("asset"))

            if (names.isEmpty) {
                None
            } else {
                Some(
                    Assets(names))
            }
        }

        assets
    }
}

case class Assets(names: Seq[String])

```

Źródło: opracowanie własne

*AssetEventProcessor* reaguje na zdarzenie *PriceUpdated* (które jest emitowane na otrzymanie zdarzenia zamknięcia transakcji z Kafki), i wykonuje dla niego polecenie Cassandra wstawienia danych do kolumn tabeli *assets*, przechowujących cenę dla zasobów. Dodatkowo tabela tworzona jest, jeżeli wcześniej nie istniała. Polecenie *findAllAssets* w *AssetsRepository*, zwraca wszystkie zasoby z tabeli Cassandra.

Pozostałe mikroserwisy tożsamo implementują swoje API (Listing nr 9.).

Listing nr 9. Deskryptory pozostałych mikroserwisów: *Trader*, *Asset* oraz *Table*

```

trait TraderService extends Service {
  def register(): ServiceCall[Register, Done]
  def login(): ServiceCall[Login, LoginDone]
  def balance(): ServiceCall[NotUsed, BalanceDone]
  def assets(): ServiceCall[NotUsed, AssetsDone]
  def asset(asset: String): ServiceCall[NotUsed, AssetDone]
  def addAsset(asset: String): ServiceCall[AddAsset, Done]

  override final def descriptor: Descriptor = {
    import Service._
    // @formatter:off
    named("trader-service")
    .withCalls(
      restCall(Method.POST, "/api/trader/register", register _),
      restCall(Method.POST, "/api/trader/login", login _),
      restCall(Method.GET, "/api/trader/balance", balance _),
      restCall(Method.GET, "/api/trader/assets", assets _),
      restCall(Method.GET, "/api/trader/asset/:asset", asset _),
      restCall(Method.PUT, "/api/trader/asset/:asset", addAsset _)
    )
    .withAutoAcl(true)
    // @formatter:on
  }
}

// ...

trait AssetService extends Service {
  def assets(): ServiceCall[NotUsed, AssetsDone]
  def asset(asset: String): ServiceCall[NotUsed, AssetDone]

  override final def descriptor: Descriptor = {
    import Service._
    // @formatter:off
    named("asset-service")
    .withCalls(
      restCall(Method.GET, "/api/asset/assets", assets _),
      restCall(Method.GET, "/api/asset/:asset", asset _)
    )
    .withAutoAcl(true)
    // @formatter:on
  }
}

// ...

object TableService {

```

```

    val TOPIC_NAME = "resolved_transactions"
  }

  trait TableService extends Service {
    def resolvedTransactionsTopic(): Topic[ResolvedTransaction]

    override final def descriptor: Descriptor = {
      import Service._
      // @formatter:off
      named("table-service")
        .withTopics(
          topic(TableService.TOPIC_NAME, resolvedTransactionsTopic
        _
        )
        .addProperty(
          KafkaProperties.partitionKeyStrategy,
          PartitionKeyStrategy[ResolvedTransaction](_.transactionId)
        )
        .withAutoAcl(true)
      // @formatter:on
    }
  }

  // ...

  trait TableStreamService extends Service {
    def resolvedTransactionsStream: ServiceCall[NotUsed,
      Source[ResolvedTransaction, NotUsed]]

    override final def descriptor: Descriptor = {
      import Service._
      named("table-stream")
        .withCalls(
          namedCall("resolvedTransactionsStream",
            resolvedTransactionsStream)
        ).withAutoAcl(true)
    }
  }

```

Źródło: opracowanie własne

Złożone polecenie transakcji poprzez *TransactionService* jest weryfikowane i przekazywane do Kafka topic „orders”, na te zdarzenia subskrybuje *TableService*, który

przetwarza je w tablicy transakcji nabycia / sprzedaży, zamykając transakcje nakładające się oczekiwaniami. Transakcja zamknięta jest emitowana do drugiego Kafka topic „*resolved\_transactions*”, te natomiast są propagowane w systemie do *AssetService* na kalkulację aktualnej ceny zasobu, a także części analitycznej projektu Apache Spark MLlib oraz strumieniującego odpowiednika *TableStreamService*, emitującego informację o transakcjach do świata zewnętrznego poprzez WebSocket.

### 3.3 Interpolator – symulacja zdarzeń użytkowników

Interpolator, czyli konsolowe narzędzie symulacji ruchu użytkowników wykonuje zapytania na systemie reaktywnym, na podstawie wprowadzanych do niego danych transakcji. Odtwarza to stan giełdy, w taki sposób, że zamykane w niej transakcje nabycia / sprzedaży zasobu odzwierciedlają historię z danych wprowadzanych. Zachowują przy tym jednocześnie przyjęte zasady działania, takie jak składanie niezależnych transakcji przez wielu niepowiązanych traderów, czy wiązanie transakcji nakładających się w celu ich zamknięcia. Następnie, dane z systemu są wyprowadzane poprzez strumień, na który nasłuchuje narzędzie i prezentuje w czasie rzeczywistym przebieg działania giełdy na składane zamówienia (Listing nr 10.).

Listing nr 10. Klient REST wykonujący zapytania na API systemu reaktywnego

```
package main

// ...

type ReactiveStockSimulator struct {
    serviceGatewayUrl string
    endpoints          map[string]map[string]Endpoint
    traders            []Trader
}

// ...

func main() {
    log.Println("Starting Interpolator")

    reactiveStock := ReactiveStockSimulator{
        serviceGatewayUrl: "http://localhost:9000",
        endpoints: map[string]map[string]Endpoint{
```



```

        "TraderService": {
            "login": Endpoint{http.MethodPost, "/api/trader/login"},
            "register": Endpoint{http.MethodPost, "/api/trader/register"},
            "balance": Endpoint{http.MethodGet, "/api/trader/balance"},
            "asset": Endpoint{http.MethodGet, "/api/trader/asset/:asset"},
            "assets": Endpoint{http.MethodGet, "/api/trader/assets"},
            "putAsset": Endpoint{http.MethodPut,
"/api/trader/asset/:asset"},
        },
        "AssetService": {
            "asset": Endpoint{http.MethodGet, "/api/asset/:asset"},
            "assets": Endpoint{http.MethodGet, "/api/asset/assets"},
        },
        "TransactionService": {
            "order": Endpoint{http.MethodPost, "/api/transaction/order"},
        },
        "TableService": {
            "resolvedTransactionsStream":
                Endpoint{http.MethodGet, "/resolvedTransactionsStream"},
        },
    },
}

log.Println("Registering Traders")
for i := 1; i <= NumOfTraders; i++ {
    go reactiveStock.register(i)
}

// ...

log.Println("Logging traders in")
for i := 1; i <= NumOfTraders; i++ {
    reactiveStock.login(i)
}

log.Println("Subscribing to resolved transactions stream")
go reactiveStock.resolvedTransactions()

log.Println("Placing orders")
for i := 0; i < NumOfTransactions; i++ {
    go reactiveStock.placeOrder("BUY")
    go reactiveStock.placeOrder("SELL")
}

// Keep Interpolator running
select {}

```

}

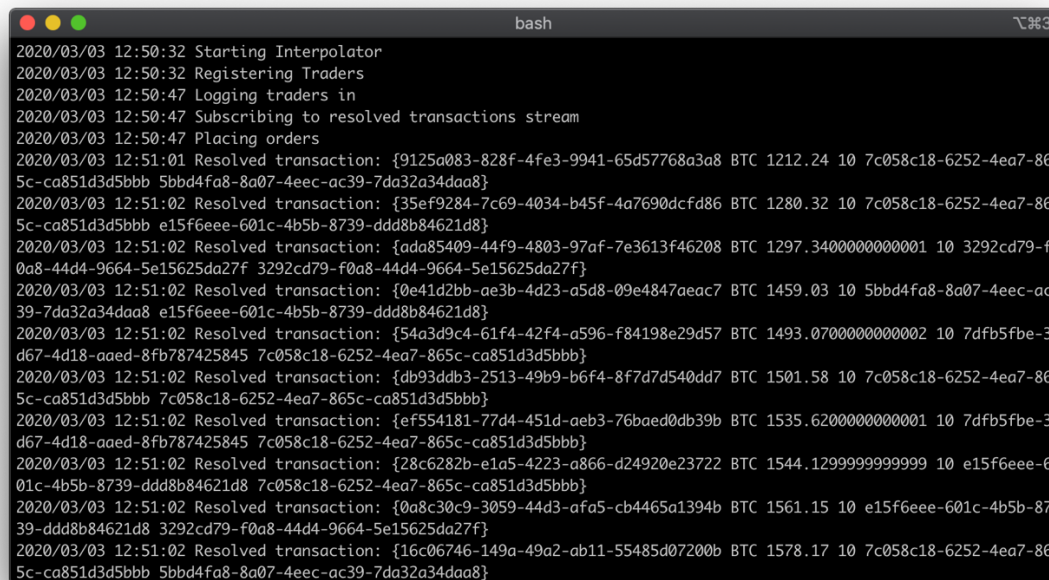
Źródło: opracowanie własne

Interpolator posiada zdefiniowane API systemu reaktywnego w postaci ścieżek do wykonywania zapytań oraz głównego *Service Gateway*, przez którego się z nim komunikuje. Procedura przebiega następująco:

- rejestracja handlarzy poprzez */register*
- zalogowanie na wcześniej podane dane w */login*
- zasubskrybowanie do strumienia zamkniętych transakcji (autoryzowane)
- złożenie transakcji kupna / sprzedaży zgodnie z interpolowanymi danymi, rozdystrybuowanymi poprzez zarejestrowanych handlarzy (autoryzowane)

Przebieg procesu symulacji możemy obserwować w konsoli, podczas uruchamiania interpolatora (Rys.19).

Rys.19. Uruchomienie interpolatora oraz obserwacja zamkniętych transakcji



```
2020/03/03 12:50:32 Starting Interpolator
2020/03/03 12:50:32 Registering Traders
2020/03/03 12:50:47 Logging traders in
2020/03/03 12:50:47 Subscribing to resolved transactions stream
2020/03/03 12:50:47 Placing orders
2020/03/03 12:51:01 Resolved transaction: {9125a083-828f-4fe3-9941-65d57768a3a8 BTC 1212.24 10 7c058c18-6252-4ea7-86
5c-ca851d3d5bbb 5bbd4fa8-8a07-4eec-ac39-7da32a34daa8}
2020/03/03 12:51:02 Resolved transaction: {35ef9284-7c69-4034-b45f-4a7690dcfd86 BTC 1280.32 10 7c058c18-6252-4ea7-86
5c-ca851d3d5bbb e15f6eee-601c-4b5b-8739-ddd8b84621d8}
2020/03/03 12:51:02 Resolved transaction: {ada85409-44f9-4803-97af-7e3613f46208 BTC 1297.3400000000001 10 3292cd79-f
0a8-44d4-9664-5e15625da27f 3292cd79-f0a8-44d4-9664-5e15625da27f}
2020/03/03 12:51:02 Resolved transaction: {0e41d2bb-ae3b-4d23-a5d8-09e4847aeac7 BTC 1459.03 10 5bbd4fa8-8a07-4eec-ac
39-7da32a34daa8 e15f6eee-601c-4b5b-8739-ddd8b84621d8}
2020/03/03 12:51:02 Resolved transaction: {54a3d9c4-61f4-42f4-a596-f84198e29d57 BTC 1493.0700000000002 10 7dfb5fbe-3
d67-4d18-aaed-8fb787425845 7c058c18-6252-4ea7-865c-ca851d3d5bbb}
2020/03/03 12:51:02 Resolved transaction: {db93ddb3-2513-49b9-b6f4-8f7d7d540dd7 BTC 1501.58 10 7c058c18-6252-4ea7-86
5c-ca851d3d5bbb 7c058c18-6252-4ea7-865c-ca851d3d5bbb}
2020/03/03 12:51:02 Resolved transaction: {ef554181-77d4-451d-aeb3-76baed0db39b BTC 1535.6200000000001 10 7dfb5fbe-3
d67-4d18-aaed-8fb787425845 7c058c18-6252-4ea7-865c-ca851d3d5bbb}
2020/03/03 12:51:02 Resolved transaction: {28c6282b-e1a5-4223-a866-d24920e23722 BTC 1544.1299999999999 10 e15f6eee-6
01c-4b5b-8739-ddd8b84621d8 7c058c18-6252-4ea7-865c-ca851d3d5bbb}
2020/03/03 12:51:02 Resolved transaction: {0a8c30c9-3059-44d3-afa5-cb4465a1394b BTC 1561.15 10 e15f6eee-601c-4b5b-87
39-ddd8b84621d8 3292cd79-f0a8-44d4-9664-5e15625da27f}
2020/03/03 12:51:02 Resolved transaction: {16c06746-149a-49a2-ab11-55485d07200b BTC 1578.17 10 7c058c18-6252-4ea7-86
5c-ca851d3d5bbb 5bbd4fa8-8a07-4eec-ac39-7da32a34daa8}
```

Źródło: opracowanie własne

### 3.4 Konstrukcja modelu predykcyjnego

Giełda opiera się na danych finansowych, które w dużej mierze są zmiennymi ciągłymi, np. w kursie cen akcji w danym okresie czasu. Pytania więc, na które chcemy poznać odpowiedzi często również opierają się na uzyskaniu wyniku w postaci ciągłej, czyli np. prognoza nadchodzących cen akcji. Do predykcji liczbowych wartości rzeczywistych stosuje się metodę uczenia maszynowego – analizę regresji – będącą formą uczenia nadzorowanego, mającą głębokie korzenie w statystyce.

Ogólna postać modelu regresji przedstawia się następująco:

- $\beta$  – nieznane parametry, wektor współczynników regresji
- $X$  – niezależne zmienne, wektor zmiennych objaśniających
- $Y$  – zmienne zależne, zmienna objaśniana
- $\varepsilon$  – błąd losowy, jego rozkład może, lecz nie musi być zależny od  $X$

$$Y = f(X, \beta) + \varepsilon$$

Gdzie  $f(X, \beta)$  to funkcja regresji, przedstawia wartości w liczbach rzeczywistych. Model zakłada, że predykowana przez nas zmienna  $Y$  (znana w procesie treningu, nieznana w ewaluacji) jest funkcją niezależnych zmiennych  $X$  oraz ich parametrów  $\beta$ , w raz z  $\varepsilon$  reprezentującym addytywny błąd losowy przeprowadzanej predykcji. Celem badacza jest estymacja funkcji regresji  $f(X, \beta)$  możliwie jak najbliżej dopasowując ją do danych (mając jednak na uwadze problemy takie, jak *overfitting*).<sup>69</sup> W procesie treningu modelu wykorzystywane są rozmaite miary błędu, m. in. RMSE, czyli *root mean square error*, określają one jak dobrze model dopasował się do danych.

$$RMSE = \sqrt{\frac{\sum_{t=1}^T (\hat{y}_t - y_t)^2}{T}}$$

Pierwiastek błędu średniokwadratowego (RMSE):

- $\hat{y}_t$  – wartości po predykcji dla czasu  $t$

---

<sup>69</sup> *Regression analysis* [online] Wikipedia [dostęp: 08.03.2020]. Dostępny w Internecie: [https://en.wikipedia.org/wiki/Regression\\_analysis](https://en.wikipedia.org/wiki/Regression_analysis)

- $y_t$  – zmienna zależna dla czasu  $t$
- $T$  – wielokrotne obserwacje

Miara ta wylicza błędy dla poszczególnych obserwacji i agreguje je do średniej wspólnej miary dającej informację, w jakim stopniu model myli się w dokonywanej predykcji.

W badaniu dla opracowania analizy regresji ceny transakcji mającej miejsce w systemie, wybrano model wywodzący się od drzew decyzyjnych (ang. *decision tree*). Algorytmy drzewa decyzyjnego stanowią formę drzewa, w którym obserwacje reprezentowane są na gałęziach, a konkluzje o wartości docelowej w liściach. Mogą przyjmować wartości zarówno dyskretne, jak i ciągłe. Stosuje się je w przypadkach klasyfikacji (mówimy wtedy o *Classification tree*), a także w regresji, czyli tzw. *Regression tree*.<sup>70</sup> Dla ograniczenia wady drzew decyzyjnych, jaką jest tendencja do przeuczenia oraz w poszukiwaniu jeszcze lepiej predykującej formy modelu wykorzystano *Random forest* (pol. las losowy), jest to jeden z algorytmów typu *ensemble*, czyli zestaw wielu algorytmów uczenia maszynowego, w tym wypadku jest to, kolekcja drzew decyzyjnych (tu: regresyjnych).

Środowisko analityczne projektu, czyli Apache Spark, Spark MLlib, a także połączenie z Apache Kafka jest wirtualizowane za pomocą Docker (Listing nr 11.). Na podstawie konfiguracji instancjonowany jest kontener, który wykonuje pipeline machine learningowy na danych pochodzących z systemu.

Listing nr 11. Konfiguracja środowiska analitycznego w *docker-compose.yml*

```
version: '3'

# Based on https://github.com/Gradient/dockerized-spark
services:
  spark:
    image: gradient/spark:2.4.4-alpine
    volumes:
      - ./opt/spark/spark-ml-analysis
    stdin_open: true
    tty: true
```

---

<sup>70</sup> *Decision tree learning* [online] Wikipedia [dostęp: 08.03.2020]. Dostępny w Internecie: [https://en.wikipedia.org/wiki/Decision\\_tree\\_learning](https://en.wikipedia.org/wiki/Decision_tree_learning)

```

command: spark-submit
--packages org.apache.spark:spark-streaming-kafka-0-
10_2.11:2.4.4,org.json4s:json4s-native_2.11:3.6.7
--master local[*]
--class "TransactionRegressorModel" /opt/spark/spark-ml-
analysis/target/scala-2.11/spark-ml-analysis_2.11-0.1.0.jar

```

Źródło: opracowanie własne

Model regresyjny to projekt Scala / sbt (aplikacja Sparkowa), który po zbudowaniu do pliku *jar* jest uruchamiany na Apache Spark w kontenerze poprzez *spark-submit*.

*TransactionRegressorModel* zawiera kod modelu predykcyjnego regresji ceny transakcji (Listing nr 12.).

Listing nr 12. Model uczenia maszynowego regresji ceny transakcji

```

// ...

// Messages passed by Kafka from reactive-stock
case class ResolvedTransaction(
  transactionId: String,
  asset: String,
  price: BigDecimal,
  quantity: BigDecimal,
  timestamp: String,
  buyer: String, // UUID
  seller: String // UUID
) extends Serializable

// Topic from reactive-stock
object ResolvedTransaction {
  val Topics = Array("resolved_transactions")

  val KafkaParams: Map[String, Object] = Map(
    "bootstrap.servers" -> "host.docker.internal:9092",
    "key.deserializer" -> classOf[StringDeserializer],
    "value.deserializer" ->
classOf[ResolvedTransactionDeserializer].getCanonicalName,
    "group.id" -> "spark-ml-analysis",
    "auto.offset.reset" -> "latest",
    "enable.auto.commit" -> (false: java.lang.Boolean)
  )

```

```

}

object TransactionRegressorModel {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession
      .builder
      .appName("TransactionRegressorModel")
      .getOrCreate()

    val streamingContext = new StreamingContext(spark.sparkContext,
Seconds(5))

    val stream = KafkaUtils.createDirectStream[String,
ResolvedTransaction](
      streamingContext,
      PreferConsistent,
      Subscribe[String, ResolvedTransaction](
        (ResolvedTransaction.Topics,
ResolvedTransaction.KafkaParams)
      )
    )

    stream.foreachRDD(record => {
      if (record.count() == 0) {
        streamingContext.stop()
      }

      record.foreach(data =>
        val featureIndexer = new VectorIndexer()
          .setInputCol("features")
          .setOutputCol("indexedFeatures")
          .setMaxCategories(4)
          .fit(data)

        // Data split into training and test sets (30% held for
testing)
        val Array(trainingData, testData) =
data.randomSplit(Array(0.7, 0.3))

        // A RandomForest model
        val rf = new RandomForestRegressor()
          .setLabelCol("label")
          .setFeaturesCol("indexedFeatures")

        val pipeline = new Pipeline()
          .setStages(Array(featureIndexer, rf))

```

```

// Train model
val model = pipeline.fit(trainingData)

// Make predictions
val predictions = model.transform(testData)

predictions.select("prediction", "label",
"features").show(100)

// Evaluate model
val evaluator = new RegressionEvaluator()
    .setLabelCol("label")
    .setPredictionCol("prediction")
    .setMetricName("rmse")
val rmse = evaluator.evaluate(predictions)
println(s"Root Mean Squared Error (RMSE) on test data =
$rmse")

val rfModel =
model.stages(1).asInstanceOf[RandomForestRegressionModel]
println(s"Learned regression forest model:\n
${rfModel.toDebugString}")
})

streamingContext.start()

// ...
}
}

```

Źródło: opracowanie własne

Inicjalizacja modelu zaczyna się od *SparkSession*, a jej następnym elementem jest *StreamingContext* (pochodzący ze Spark Streaming), który strumieniuje dane pochodzące z Kafki, z kanału *resolved\_transactions*, subskrybując do niego podobnie jak każdy niezależny mikroservis. Informacje pochodzące z Kafki są tożsame z wiadomościami przekazywanymi w systemie reaktywnym, mamy więc dostęp do zdarzeń z latencją w okolicach mniej niż jednej sekundy. Wszelkie informacje pochodzące z wiadomości typu *ResolvedTransaction*, m. in. cena transakcji, jej wolumen, data, itp. służą do modelowania analizy regresyjnej. Dane te są następnie przekazywane do *RandomForestRegressor*, tworzone w pipeline, którego metoda *fit* odpowiada za przeprowadzenie treningu modelu – w ramach niego inteligentnie

wybierane są cechy istotne dla predykcji z wprowadzanych danych oraz rozkład drzew decyzyjnych i wag ich poszczególnych predykcji dla zbiorczego wyniku całego regresyjnego lasu losowego. Metoda *model.transform()* wykonuje predykcję na wytrenowanym modelu posiłkując się danymi testowymi, gdy *RegressionEvaluator* przetwarza uzyskane predykcje obliczając ich błędy predykcji RMSE w metodzie *evaluate*. Tak uzyskany model regresyjny przeprowadza proces treningu na danych płynących bezpośrednio z części operacyjnej, w czasie rzeczywistym. Uzyskana w ten sposób predykcja może np. ponownie zostać publikowana do topicu Kafki, aby została natychmiastowo użyta w podejmowaniu decyzji części operacyjnej systemu. Ostatnia linijka z użyciem *RandomForestRegressionModel* prezentuje w czytelnej formie przyjaznej badaczowi postać modelu lasu losowego razem z dobranymi parametrami.

W rozdziale omówiono badaną implementację systemu reaktywnego strumieniującego dane Fast Data z analizą predykcyjną Machine Learning. Skupiono się na najistotniejszych szczegółach kodowych oraz punktów komunikacji. Przedstawiono również drogę od zainicjowania zdarzenia w systemie, przejścia kompletnej logiki, po część analityczną, która dokonuje analizy operacyjnej i zwraca dane do systemu. Kompletny kod źródłowy systemu został opublikowany na portalu Github do wglądu zainteresowanych badaczy – referencje można znaleźć w zakończeniu niniejszej pracy.



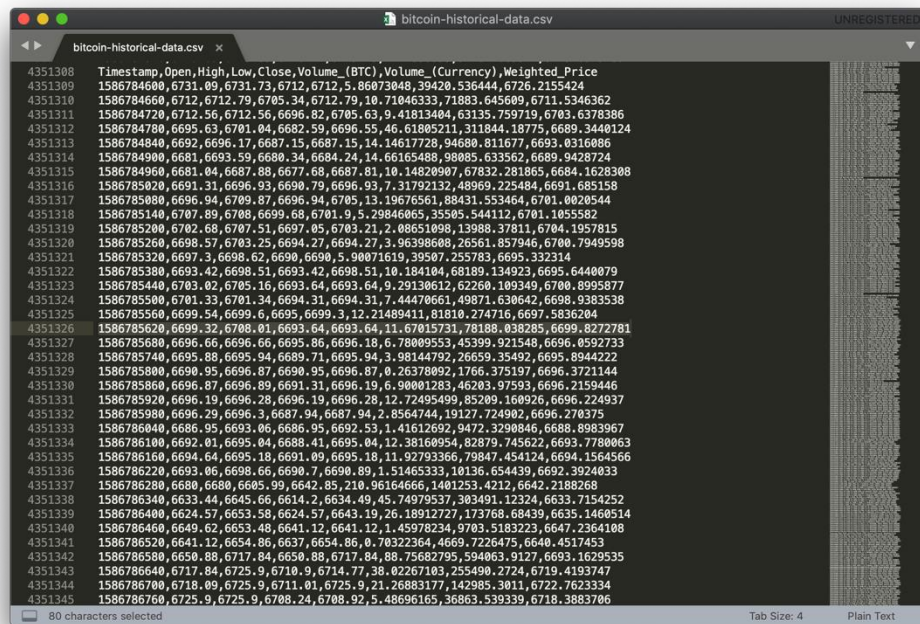
## 4. Analiza predykcyjna czasu rzeczywistego regresji ceny transakcji

Rozdział dotyczy części analizy operacyjnej omówionej implementacji systemu reaktywnego. Przedstawiono w nim przebieg procesu predykcji, wraz z selekcją danych do interpolatora, trening modelu uczenia maszynowego z wynikiem i komentarzem oraz ewaluację modelu w celu oceny poziomu błędów. Omówiono również czym jest kontekstowy re-trening. W ostatniej sekcji przedstawiono finalne wnioski przeprowadzonej analizy przypadku oraz eksperymentu, zwrócono także uwagę na znaczenie biznesowe oraz rekomendacje dla adaptujących.

### 4.1 Przebieg procesu predykcji

Na przeprowadzenie eksperymentu zaimplementowanej analizy predykcyjnej wykorzystano historyczne dane cen transakcji kryptowaluty Bitcoin (BTC). Wybór oparto o powszechny dostęp do cyfrowych danych, zawierających szczegóły odnośnie kwot podlegających wymianie, wolumenów, daty transakcji oraz innych istotnych informacji przy procesie tworzenia rzeczywistego przypadku wykorzystania modeli uczenia maszynowego dla scenariuszy giełdowych (Rys.20).

Rys.20. Dane kryptowaluty Bitcoin przetwarzane przez interpolator



*Źródło: opracowanie własne na bazie danych kryptowaluty Bitcoin*

Wycinek pobranych danych stanowi zakres czasowy pomiędzy marcem, a grudniem w 2019 roku, składający się w sumie z ponad 100 000 transakcji. W procesie treningu wybrano jedynie fragment ok. 70% jego zakresu jako dane treningowe, pozostałe 30% stanowi dane testowe do ewaluacji. Ideą podziału danych historycznych jest ocena retrospektywna, czy jeżeli dane byłyby istotnie z czasu teraźniejszego, to czy wytrenowany model prawidłowo predykowałby dane przyszłe. Na potrzeby eksperymentu więc zamiast przeprowadzać analizę na świeżo powstających danych, symuluje się ich przeszłe odwzorowanie i porównuje, czy dalsza część danych historycznych pokrywa się z predykcją zwróconą przez model – udowadnia to skuteczność prowadzonej analizy w warunkach wygodnych do oceny badawczej.

Przygotowane dane wczytano do interpolatora, który wykorzystuje informacje o zawartych transakcjach, aby odwzorować je w faktycznych operacjach zakupu / sprzedaży w systemie reaktywnym giełdy. Tym sposobem z pewną dokładnością odtwarzana jest sytuacja historyczna mająca miejsce w rzeczywistości, na potrzeby eksperymentu. Interpolator wykonuje odpowiednie operacje rejestracji handlarzy, logowania, interpolowania oraz wykonania operacji, nasłuchuje również, czy operacje faktycznie powiodły się (poprzez strumień na WebSocket wystawiany z serwisu, który pobiera te dane z Kafki). Wprowadzone transakcje są weryfikowane, przetwarzana jest na nich zdefiniowana logika biznesowa pomiędzy mikroserwisami, ostatecznie transakcja jest zamykana, a informacja o tym trafia do Kafki, do kanału *resolved\_transactions*.

Następnym ogniwem jest część analityczna systemu, która nasłuchuje na strumieniowane zdarzenia nadchodzące z kanału Kafki w Apache Spark. Aplikacja Sparkowa subskrybuje na każde zdarzenie, będące informacją o pojedynczej zawartej transakcji i przeprowadza na niej trening oraz ewaluację modelu uczenia maszynowego. Kontekstowo prowadzona jest ocena poziomu błędów predykcji modelu i ponowny jego re-trening, jeśli błędy były zbyt znaczące.

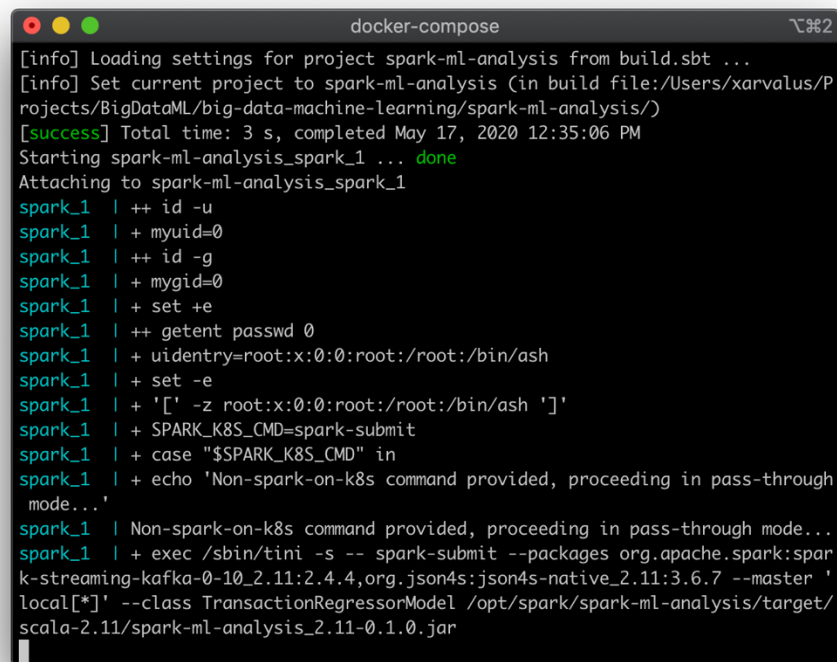
Wyniki predykcji następnie są wykorzystywane operacyjnie w systemie reaktywnym, będąc publikowanymi z powrotem do kanału Kafki, z którego każda

subskrybująca część systemu ma natychmiastowy dostęp do wyniku ewaluacji modelu uczenia maszynowego.

## 4.2 Trening modelu uczenia maszynowego

Uruchomienie eksperymentu rozpoczyna się od komendy *make train* w projekcie, odpowiada to za wirtualizację środowiska analitycznego poprzez *docker-compose* – ustawienie Apache Spark z connectorem Apache Kafka, Spark MLlib oraz niezbędnych zależności. Wykonuje się również budowę aplikacji Sparkowej *TransactionRegressorModel*, która zawiera kod uczenia maszynowego, będącego obiektem przeprowadzanego badania. Następnie polecenie *spark-submit* wraz z parametrami inicjuje rozpoczęcie działania modelu regresyjnego (Rys.21).

Rys.21. Inicjacja procesu treningu modelu uczenia maszynowego



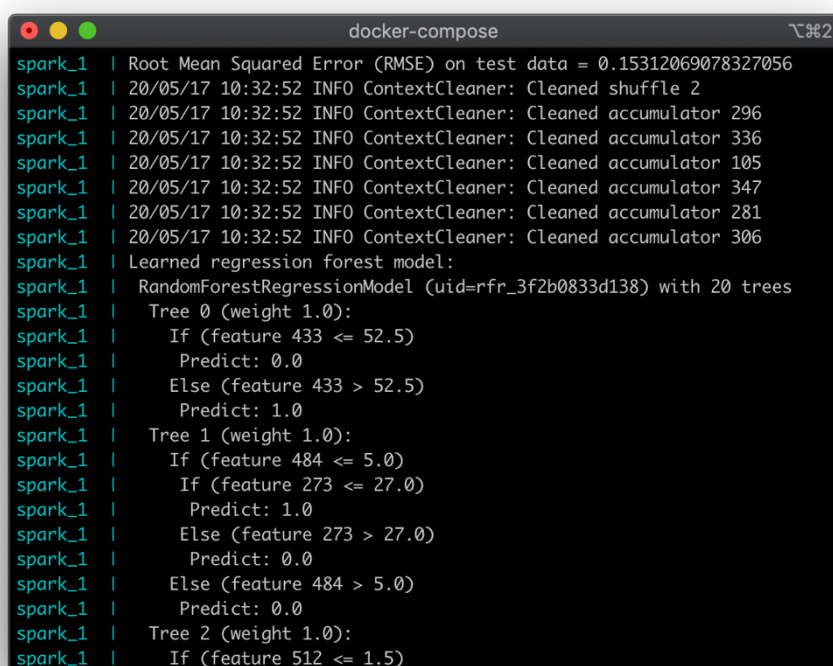
```
docker-compose
[info] Loading settings for project spark-ml-analysis from build.sbt ...
[info] Set current project to spark-ml-analysis (in build file:/Users/xarvalus/P
rojects/BigDataML/big-data-machine-learning/spark-ml-analysis/)
[success] Total time: 3 s, completed May 17, 2020 12:35:06 PM
Starting spark-ml-analysis_spark_1 ... done
Attaching to spark-ml-analysis_spark_1
spark_1 | ++ id -u
spark_1 | + myuid=0
spark_1 | ++ id -g
spark_1 | + mygid=0
spark_1 | + set +e
spark_1 | ++ getent passwd 0
spark_1 | + uidentry=root:x:0:0:root:/root:/bin/ash
spark_1 | + set -e
spark_1 | + '[' -z root:x:0:0:root:/root:/bin/ash ']'
spark_1 | + SPARK_K8S_CMD=spark-submit
spark_1 | + case "$SPARK_K8S_CMD" in
spark_1 | + echo 'Non-spark-on-k8s command provided, proceeding in pass-through
mode...'
spark_1 | Non-spark-on-k8s command provided, proceeding in pass-through mode...
spark_1 | + exec /sbin/tini -s -- spark-submit --packages org.apache.spark:spar
k-streaming-kafka-0-10_2.11:2.4.4,org.json4s:json4s-native_2.11:3.6.7 --master '
local[*]' --class TransactionRegressorModel /opt/spark/spark-ml-analysis/target/
scala-2.11/spark-ml-analysis_2.11-0.1.0.jar
```

Źródło: *opracowanie własne*

Otrzymywane zdarzenia z Kafki zawierają informacje o zamkniętej transakcji w systemie z opóźnieniem rzędu pojedynczej sekundy, najistotniejsze dane z punktu

widzenia modelu predykcyjnego to cena, po której transakcję zawarto, data zamknięcia oraz wolumen wymiany. Są one wykorzystywane bezpośrednio w modelu do otrzymania predykcji regresji, choć pozostałe informacje mogą być równie istotne z punktu widzenia analizy, takie jak np.: tożsamość nabywcy i kupca, typ zasobu itp. w zależności od rodzaju wykonywanego badania. Wprowadzenie danych do modelu lasu losowego w procesie treningu skutkuje rozdystrybuowaniem wag przy wielu drzewach losowych stosowanych wewnątrz algorytmu, z których następnie brany jest ostateczny werdykt predykcji co do przyszłych wartości cenowych transakcji (Rys.22). W danym momencie przyjęto, że próbujemy dowiedzieć się, jakie wartości cenowe można oczekiwać po nadchodzących przyszłych transakcjach, na bazie tych, które są w danym okresie czasowym już znane (wykorzystane do treningu modelu).

Rys.22. Wynik treningu modelu z podglądem w strukturę lasu losowego oraz poziomem błędów RMSE



```
spark_1 | Root Mean Squared Error (RMSE) on test data = 0.15312069078327056
spark_1 | 20/05/17 10:32:52 INFO ContextCleaner: Cleaned shuffle 2
spark_1 | 20/05/17 10:32:52 INFO ContextCleaner: Cleaned accumulator 296
spark_1 | 20/05/17 10:32:52 INFO ContextCleaner: Cleaned accumulator 336
spark_1 | 20/05/17 10:32:52 INFO ContextCleaner: Cleaned accumulator 105
spark_1 | 20/05/17 10:32:52 INFO ContextCleaner: Cleaned accumulator 347
spark_1 | 20/05/17 10:32:52 INFO ContextCleaner: Cleaned accumulator 281
spark_1 | 20/05/17 10:32:52 INFO ContextCleaner: Cleaned accumulator 306
spark_1 | Learned regression forest model:
spark_1 | RandomForestRegressionModel (uid=rfr_3f2b0833d138) with 20 trees
spark_1 | Tree 0 (weight 1.0):
spark_1 |   If (feature 433 <= 52.5)
spark_1 |     Predict: 0.0
spark_1 |   Else (feature 433 > 52.5)
spark_1 |     Predict: 1.0
spark_1 | Tree 1 (weight 1.0):
spark_1 |   If (feature 484 <= 5.0)
spark_1 |     If (feature 273 <= 27.0)
spark_1 |       Predict: 1.0
spark_1 |     Else (feature 273 > 27.0)
spark_1 |       Predict: 0.0
spark_1 |   Else (feature 484 > 5.0)
spark_1 |     Predict: 0.0
spark_1 | Tree 2 (weight 1.0):
spark_1 |   If (feature 512 <= 1.5)
```

Źródło: opracowanie własne

Wynik procesu treningu przedstawia wartość RMSE, czyli poziom błędów predykcji, jaki istnieje w uzyskanym modelu wynoszący ok. 0,1531 – wynikający z

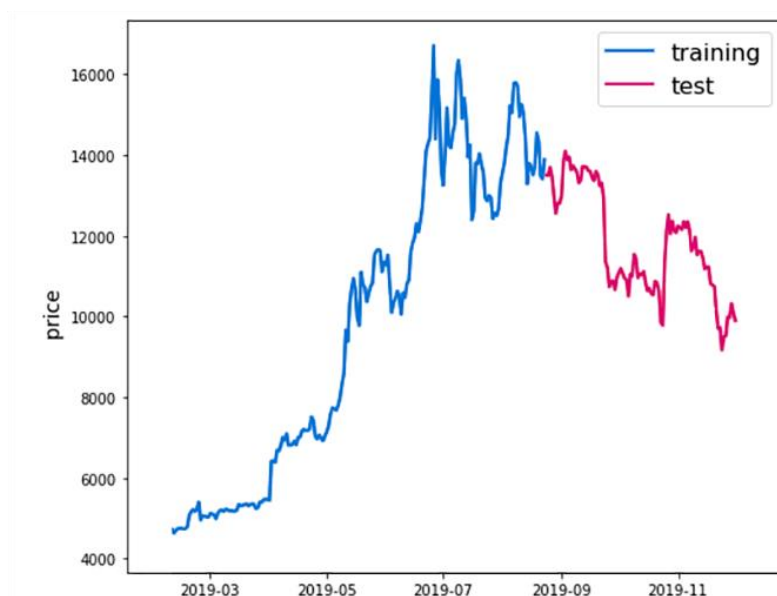
przeanalizowania zbioru danych testowych, czyli horyzontu czasowego, który chcemy zaprognozować w ramach eksperymentu.

#### **4.3 Ewaluacja z obserwacją błędów predykcji oraz kontekstowym re-treningiem**

Wytrenowany model z powodzeniem może zostać wykorzystany do ewaluacji w środowisku produkcyjnym. Na tym etapie dane, które przychodzą z subskrybowanego strumienia Kafki są przekazywane bezpośrednio do modelu uczenia maszynowego, który zwraca predykcję cen na nadchodzący okres. Predykcje te następnie mogą zostać ponownie wysłane do kanału Kafki bezpośrednio po ich uzyskaniu (najnowszej iteracji ewaluacji modelu, czyli najbardziej aktualnej wersji). Z poziomu Kafki system reaktywny może swobodnie uzyskać dostęp do aktualnej predykcji dla dowolnego mikroservisu. Informacja ta pozwala na zaproponowanie inteligentniejszych funkcjonalności aplikacji, podnosząc kluczową dla użytkownika użyteczność usługi. W badanym przypadku jest to predykcja przyszłych cen zawieranych w transakcji kupna / sprzedaży, ale z powodzeniem mogą to również być przykłady sugestii poszukiwanych produktów, personalizacji, wykrywania anomalii i innych – wykorzystywanych całościowo od części operacyjnej po analityczną i z powrotem, w czasie rzeczywistym.

Poniższy wykres obrazuje dane treningowe oraz testowe, które są horyzontem czasowym, na którym przeprowadzono predykcję (Rys.23).

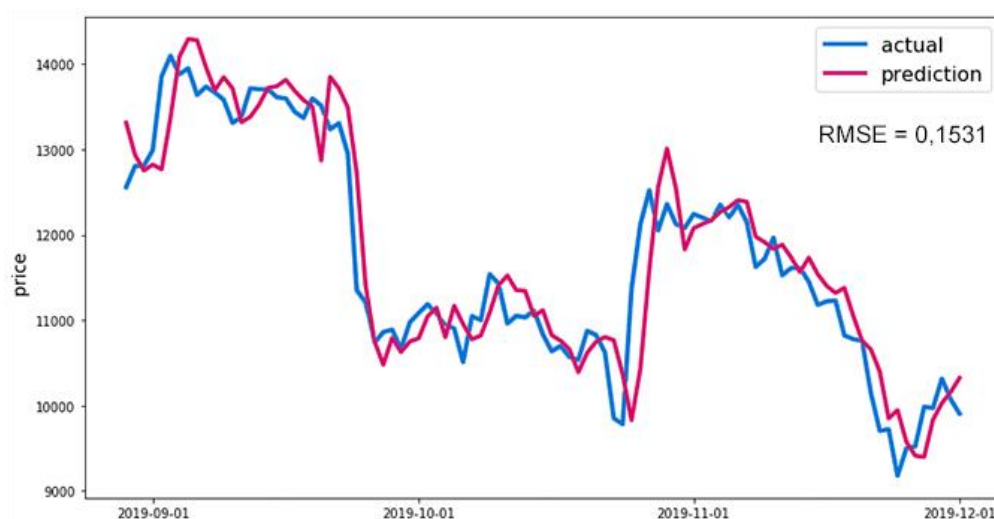
Rys.23. Wykres danych wykorzystanych w procesie treningu oraz ewaluacji modelu uczenia maszynowego



Źródło: opracowanie własne

Uzyskany w procesie treningu model regresyjny przedstawia predykcje dla nadchodzących transakcji, bazując na zdarzeniach systemu, interpolowanych z wybranych danych kursów kryptowaluty. Dzięki zastosowanym warunkom badawczym porównujemy uzyskaną predykcję, z faktycznymi danymi historycznymi na nadchodzący okres (Rys.24).

Rys.24. Wykres uzyskanej predykcji oraz danych rzeczywistych (historycznych)



Źródło: opracowanie własne

Po analizie wykresów widoczna jest minimalna różnica w predykcji od uzyskanych rzeczywistych cen transakcji. Wynosi ona wg.  $RMSE = 0,1531$ , oceniony ekspercko

poziom dokładności predykcji można uznać za dobry. Tak opracowany model zgodnie z oczekiwaniami predykuje nadchodzące ceny, a wartość uzyskana z analizy machine learning może z powodzeniem zostać przekazana do części operacyjnej systemu, wspierając użytkowników oraz pośrednio zwiększając zyski generowane z przedsięwzięcia.

Analizy predykcyjne, jednakże mają charakter krótko istotnych, im bardziej długoterminowe skutki próbuje się przewidzieć, tym bardziej mylą się w przewidywaniach. Należy kontrolować więc horyzont oraz poziom błędów, przy których opracowany model uczenia maszynowego nie generuje już dostatecznie dobrego wyniku przewidywania i wymaga ponownego wytrenowania, aby lepiej dopasował się do aktualnych trendów rynkowych. Proces ten określa się kontekstowym retrainingiem – aplikacja Sparkowa obserwuje jaki poziom błędów obecnie jest obserwowany na bazie aktualnej ewaluacji modelu czasu rzeczywistego. W naszym wypadku jest to RMSE, gdy przekroczy ono poziom 0,25 model uznawany jest za nieefektywny, a jego użycie produkcyjne przechodzi z ewaluacji w ponowny proces treningu na bazie istniejących historycznych danych zaktualizowanych o nowo wygenerowane dane i przechowywane w systemie w postaci zdarzeń w Kafce. Po przejściu procesu treningu osiągamy ponownie predykcję, która powinna być lepiej dopasowana do predykowanego horyzontu (niższy poziom błędów RMSE). Retrening na bazie błędów pozwala na jeszcze większą elastyczność i formę inteligentnego dopasowania w analizie w sposób automatyczny, w miejscu, gdzie dotychczas potrzebny był zespół badaczy i ponowne opracowywanie modelu, który lepiej opisywałby rzeczywistość. Nie jest to naturalnie metoda zastępująca kreatywnych specjalistów data science, wbrew pozorom wprowadza to większe ryzyko, że świeżo opracowany model ad-hoc będzie błędnie predykował. Utrzymanie przypadku użycia machine learning z fast data oraz kontekstowym retrainingiem wymaga o wiele więcej dbałości, testów i obserwacji, niż wynikałoby to w podejściu standardowym.

#### **4.4 Wnioski, znaczenie biznesowe oraz rekomendacje dla adaptujących**

Stosowana technologia i rozwiązania wywodzą się od tzw. *modern enterprise*, nowoczesnych korporacji, które powstały na fазie ostatniego boomu technologicznego i

w krótkim czasie urosły do zasięgu *web-scale* oraz tysięcy pracowników. Jako pierwsze rozwiązywały problemy skalowalności, w dużej mierze zawdzięczając swój sukces przetwarzanym danym – firmy takie, jak Spotify czy Netflix charakteryzują się agresywną innowacją i szerokim wykorzystaniem modeli uczenia maszynowego stosowanym w czasie rzeczywistym na danych generowanych przez użytkowników. Dzięki ich przełomowym wysiłkom powstają inteligentniejsze aplikacje: skalowalne, czasu rzeczywistego i napędzane danymi.

Opracowana architektura odpowiada na najwyższe wymagania skalowalności, niezawodności, przetwarzania danych klasy Big Data / Fast Data, korzystając z dekad postępu w technologiach informatycznych, będąc jednocześnie nowoczesną formą oprogramowania, która czerpie z zalet nowatorskości i innowacji. Możliwość wykorzystania jej w rzeczywistości biznesowej stanowi przewagę konkurencyjną dla firm, które mogą dzięki niej uzyskać wartość z danych w czasie rzeczywistym bezpośrednio w części operacyjnej serwowanych usług. Wraz z nadejściem rewolucji danych, przetwarzania w chmurze, doprowadziło to do powstania nowych nisz, branż teleinformatycznych, które zaczęły specjalizować się w omawianej tematyce – sukcesy te przekładają się na odkrywanie nowych scenariuszy zastosowania informatyki w biznesie, w efekcie tworząc oprogramowanie jeszcze inteligentniejsze, bliżej użytkownika i będącego gotowym na wyzwania nowoczesnej ekonomii, w której bycie podłączonym w sieci stanowi główny priorytet operacyjności. Świat generuje masywne wolumeny danych, działa w czasie rzeczywistym, połączenie części operacyjnej z analityczną w zunifikowanym rozwiązaniu umożliwia jeszcze szersze dotarcie do potrzeb klientów i inteligentne jej zaspokajanie. Dotyczy to danych wewnętrznych, tak jak omawiany przypadek zdarzeń zawierania transakcji, jak i w luźno pojętych danych z usług instytucji trzecich, social media, artykułów, wyników wyszukiwania, głębokiego Internetu itp. W przyszłości oczekuje się emergencji jeszcze większej ilości źródeł i danych, które będą mogły zostać przetwarzane przez oprogramowanie, liderem tej zmiany będą te przedsiębiorstwa, które będą potrafiły uzyskać dostęp do jak największej ich części i wykorzystać wartość z nich płynącą do zyskania przewagi nad konkurentami.

Przeprowadzony eksperyment wykorzystania uczenia maszynowego w opracowanej analizie przypadku reaktywnego systemu strumieniującego fast data udowodnił, że



możliwym jest uzyskanie wartości dodanej wynikającej z zastosowania analiz Machine Learningu w czasie rzeczywistym. Stworzony system reaktywny odpowiada wymaganiom skalowalności, dogodnej natychmiastowej komunikacji pomiędzy jego komponentami oraz podejścia napędzanego danymi. Stanowi również stabilną podstawę rozbudowy systemu o dodatkowe funkcjonalności i przypadki użycia, zgodnie z wymaganiami biznesu. Stosując zaproponowane podejście możliwym jest uzyskanie nieskrępowanego wglądu we wszelkie zdarzenia zachodzące w systemie, które są kolekcjonowane w strumień danych czasowych. Menedżerowie, analitycy oraz badacze data science posiadają lepszy ogłód na całość funkcjonującego rozwiązania, uzyskując możliwość kreatywnej dedukcji przy odpowiedzi na pytanie – *co można jeszcze uzyskać w oparciu o dane generowane przez użytkowników naszego produktu* – wielu z nich upatruje obecny rozwój tego podejścia w dynamicznym rozwoju technologii sztucznej inteligencji w ostatnich latach. Do jej efektywnego wykorzystania potrzebne są duże ilości danych – przedstawiony system skupia się na zebraniu ich dla nowych przypadków zastosowania uczenia maszynowego. Efektem ich stosowania jest dalej idąca automatyzacja, optymalizacja, podniesienie użyteczności w wykorzystaniu przez użytkowników. Choć często nie są to cele same w sobie istnienia zysku w przedsięwzięciu IT, to jednak stanowią one przewagę, która może decydować o wzroście bądź upadku konkurujących przedsiębiorstw. Nowoczesne korporacje „modern enterprise” reprezentują szereg funkcjonalności, od automatycznej translacji tekstów, rozpoznawania i klasyfikacji obiektów, autonomicznego przetwarzania danych tekstowych i liczbowych, systemów rozpoznawania dźwięku i reakcji na komendy głosowe, proponowanie użytkownikom gotowych rozwiązań na bazie ich aktualnych zachowań i przechowywanych informacji, czy też bardziej fizycznych zastosowań takie, jak autonomiczne pojazdy. Obecny kierunek rozwoju IT skupia się na tych trudnych do rozwikłania problemach, przodują w nich firmy technologiczne, które mogą pozwolić sobie na duże nakłady na badania i rozwój (R&D). Ich efekty są trudne do przewidzenia, a także o wiele trudniejsze do wdrożenia. Niemniej jednak firmy takie jak Microsoft, Apple, Facebook, Google, Uber, Netflix, Spotify, Amazon i inne mogą pochwalić się istnieniem takich rozwiązań – wprowadzenie ich zwiększa ich dominację w obecnie zajętych niszach konsumenckich, a także daje pole dla wielu nowopowstających start-up’ów, które próbują wygrać tą część rynku, która została pominięta przez powyższe molochy branży IT. Z czasem, jak AI / ML / DL stają się coraz bardziej pospolite, ich najbardziej ogólne zastosowania są popularyzowane jako

usługi, takie jak AWS Rekognition, Lex, Personalize, Transcribe itd., czy też wydawane jako open source'owe frameworki, np. Tensorflow. Trudno przewidzieć ile tak naprawdę z marzenia o SI uda się obecnie osiągnąć, czy doczekamy się szerokiej adaptacji w wielu sferach życia, czy pozostaną one jedynie jako rozwiązania największych z instytucji. Niemniej jednak drzemie w nich ogromny potencjał, który pozwoli na jeszcze dalsze przekroczenie granicy postępu w IT. Być może doświadczymy tego teraz, a być może w nadchodzących dekadach.

Firmy, chcące uzyskać postęp wprowadzając omawiane rozwiązania muszą skupić się na danych jakie posiadają oraz jakie są w stanie uzyskać. Przekłada się to w dużej mierze na ruch, jaki generują użytkownicy w ich rozwiązaniach, oraz jakie elementy działania są monitorowane i przekazują informacje istotne biznesowo. Architektura systemu reaktywnego odpowiada najwyższym potrzebom skalowalności, dla największych przedsięwzięć skali Internetu. Prawdopodobnie więc przeciętna instytucja nie potrzebuje działać w tak szerokiej skali. Niemniej jednak opracowanie rozwiązania jest dla nich możliwe, wiąże się to z dużymi nakładami na zasoby, przede wszystkim doświadczonych software engineerów, architektów, ekspertów dziedziny oraz opłaty za infrastrukturę chmurową. Znalezienie odpowiednich osób jest wyzwaniem, szczególnie, że Scala i pochodne jej technologie są stosunkowo niszowe (systemy rozproszone oraz Big Data), i nie należą do zasobów tanich. Nierzadko wykonanie projektu w omawianych technologiach wymaga zbudowania zespołu od zera, rekrutując ludzi dla potencjału, pomagając im rozwinąć się pod okiem mentorów, mogąc dojść do swojej świetności budując system, który dla wielu stanowi wyzwanie – jest to niezwykle pasjonujące zajęcie i szereg ambitnych specjalistów poszukuje takich doświadczeń. To m. in. dlatego Scala uchodzi za technologię wokół której istnieje mnóstwo utalentowanych osobistości.

Z pomocą firmom adaptującym służą usługi partnerskie, m. in. wsparcie firmy Lightbend, która jest inicjatorem omawianych transformacji. Zapewnia ona konsultację ekspertów, programy certyfikacyjne oraz materiały i dokumentacje dla poszerzania wiedzy oraz know-how. Wielu wybitnych ekspertów tej dziedziny można znaleźć również na konferencjach poświęconych Big Data, AI, danym czy chmurze. W Polsce działają również firmy specjalizujące się w w/w. tematyce, a także Blockchain oraz Fintech budujące rozwiązania oparte na ekosystemie Scali – są to VirtusLab, Scalac

oraz SoftwareMill. Szczególnie Ci ostatni cenieni są na rynku międzynarodowym, jako niezwykle uzdolnieni eksperci w swojej branży oraz efektywni we współpracy.

W praktyce biznesowej / technologicznej szczególnie nabrał tempa przypadek użycia IoT (ang. *Internet of Things*, pol. Internet Rzeczy). Scenariusz ten zakłada istnienie wielu czujników, np. w każdym pojedynczym wyprodukowanym samochodzie, który ma połączenie z Internetem, GPS itd. Czujniki te zbierają dane o pokonanej drodze, zużyciu paliwa, aktualnej pozycji, stylu jazdy, właścicielu, ilości pasażerów itp. Możliwość zebrania oraz przetworzenia tych danych wymaga wysoce skalowalnej i wydajnej infrastruktury – takiej jak rozproszony system reaktywny oparty o Akka (Lagom) oraz narzędzia Big Data, takie jak Cassandra czy Spark. Auta produkowane masowo na całym świecie użytkowane są w setkach tysięcy egzemplarzy, a każdy z nich posiada wiele czujników, które w czasie rzeczywistym nadają informację o swoich pojazdach. Zebranie, przetworzenie oraz przechowanie tych danych stanowi monumentalne wyzwanie. Niech zobrazowany przykład zilustruje przed jakimi zadaniami staje ów technologia oraz ludzie, którzy ją reprezentują – i jak niezwykle istotnym jest przekroczenie granicy postępu w skali operacyjnej współczesnych rozwiązań IT.

W rozdziale omówiono przebieg oraz wyniki realizacji eksperymentu analizy predykcyjnej Machine Learning implementowanego systemu reaktywnego. Prócz części technicznej oraz wykonania analizy, skupiono się również na osiągniętych rezultatach, a także na zrozumieniu kontekstu biznesowego. Rozważono jakie są efekty zrealizowanej analizy przypadku pod kątem technicznym, jak i biznesowym. Zaprezentowano również jaki jest dostęp do omawianych technologii, specjalistów i czym kierować się w próbie adaptacji niniejszego systemu.

## **Zakończenie**

Celem pracy było badanie rozwiązań umożliwiających transformację tradycyjnej analizy ETL / Big Data na korzyść natychmiastowej analizy operacyjnej Fast Data, z wykorzystaniem algorytmów Machine Learning w czasie rzeczywistym. Cel ten został zrealizowany w oparciu o:

- opracowanie kompletnej infrastruktury skalowalnego reaktywnego systemu rozproszonego strumieniującego Fast Data do ewaluacji czasu rzeczywistego modeli uczenia maszynowego
- uzyskanie wysokiego poziomu dokładności predykcji w stworzonym modelu regresji uczenia maszynowego
- identyfikacja technologicznych czynników sukcesu firm wykorzystujących wartość z danych do osiągnięcia przełomowych innowacji.

Rozdział pierwszy wprowadza do zagadnień omawianych w całości pracy oraz stanowi podstawę pod realizację przyjętych celów. W szczególności ważny jest dla identyfikacji istotnych czynników sukcesu firm wykorzystujących wartość z danych. Przedstawiono w nim czym jest Big Data, Fast Data, sztuczna inteligencja, Machine Learning, system reaktywny oraz ich szczegółowe koncepcje, takie jak systemy rozproszone, programowanie asynchroniczne, konkurencyjne z paralelizacją, Actor model, wykorzystanie chmury oraz architektury mikroserwisowej, a także posiłkowanie się nierelacyjnymi bazami danych oraz bazami NewSQL.

Drugi rozdział skupiając się na narzędziach oraz architekturze przyczynił się do realizacji celu opracowania kompletnej infrastruktury systemu reaktywnego, zwrócił również uwagę na wiele istotnych technologicznych czynników sukcesu w postaci innowacyjnych rozwiązań, z jakich korzystają firmy omawianego kierunku rozwoju. Opisano w nim narzędzia SMACK stack, takie, jak Apache Spark, Lagom framework (Akka), Cassandra i Apache Kafka. Przedstawiono również bibliotekę uczenia maszynowego Spark ML oraz wzorce projektowe CQRS, ES, DDD.

W rozdziale trzecim zrealizowano cel w postaci opracowanej infrastruktury systemu reaktywnego strumieniującego dane Fast Data z ewaluacją regresyjnej analizy Machine Learning w czasie rzeczywistym. Omówiono architekturę rozwiązania,

implementację, opis jej komponentów, przebieg procesu działania oraz przedstawienie relacji pomiędzy elementami. Rozdział zawiera wiele listingów kodu, wraz z opisem zasady funkcjonowania systemu.

Czwarty rozdział w pełni poświęcono analizie predykcyjnej Machine Learning czasu rzeczywistego, opracowanej w poprzednim rozdziale. Przedstawiono przebieg procesu predykcji, selekcję danych, przeprowadzenie procesu treningu oraz wykonanie eksperymentu z kontekstowym re-treningiem. Zrealizowano cel o wysokim poziomie dokładności predykcji w stworzonym modelu regresji uczenia maszynowego. Pod koniec rozdziału skupiono się na aspektach biznesowych, rozważaniach nad dobranymi narzędziami i koncepcjami, które pozwoliły na wypracowanie nowej wartości dodanej. Realizując tym samym cel identyfikacji technologicznych czynników sukcesu firm wykorzystujących wartość z danych do osiągnięcia przełomowych innowacji, razem z syntezą informacji z poprzednich rozdziałów.

Dzięki realizacji powyższych celów z sukcesem zrealizowano cel pracy, jakim była transformacja tradycyjnej analizy ETL / Big Data na korzyść natychmiastowej analizy operacyjnej Fast Data, z wykorzystaniem algorytmów Machine Learning w czasie rzeczywistym. Przedstawiona analiza przypadku implementacji systemu reaktywnego giełdy *reactive-stock*, wraz z eksperymentem ewaluacji modelu maszynowego regresji ceny transakcji udowodniły, że możliwym jest transformacja obecnych przypadków analitycznych istniejących w odseparowaniu od części operacyjnej. Po zastosowaniu przedstawionych rozwiązań, dotychczasowe analizy offline staną się analizami operacyjnymi online, przeprowadzane będą w ułamku sekundy od zaistnienia odpowiedniego zdarzenia w systemie, działając jeszcze bliżej użytkowników, zwracając natychmiastową wartość wynikającą z danych od razu do aplikacji w systemie.

Możliwość nieograniczonego wglądu w dane systemu, generowane przez zdarzenia użytkowników pozwala na jeszcze kreatywniejsze sposoby wykorzystania algorytmów sztucznej inteligencji w celu usprawnienia procesów biznesowych produktów. Natychmiastowa analiza umożliwi wyprzedzenie konkurencji oraz gotowość na przeprowadzanie analiz ad-hoc, prezentując użytkownikom wyniki predykcji oraz wyższą użyteczność usług. Zaprojektowana architektura pod kątem działania w skali Internetu odpowiednio zareaguje w czasie normalnego ruchu, jak i wzmożonego, przez

krótkie, jak i długie okresy czasu, a także w sytuacjach eksplozji ruchu w systemie, tak jak ma to miejsce na przykład w święta okresowe w sprzedaży e-commerce. Integrując prezentowane rozwiązania tworzy się produkty skalowalne, czasu rzeczywistego i napędzane danymi. Podejścia te stanowią obecne wyzwanie, przed jakim stoi branża IT, a ich zrealizowanie pozwoli na przekroczenie granicy postępu w skali operacyjnej współczesnych rozwiązań technologicznych.

Do celów dalszych badań kod źródłowy opracowanej implementacji został udostępniony na portalu Github – projekt systemu reaktywnego z analizą uczenia maszynowego „*reactive-stock*”<sup>71</sup>. Kod źródłowy dostępny jest na otwartej licencji MIT, dla każdego zainteresowanego badacza. Udostępniono również wyniki pracy badawczej w projekcie „*big-data-machine-learning-thesis*”<sup>72</sup>.

---

<sup>71</sup> <https://github.com/Xarvalus/big-data-machine-learning>

<sup>72</sup> <https://github.com/Xarvalus/big-data-machine-learning-thesis>

## Bibliografia

1. *What is Big Data?* [online] SAS Institute Inc.  
[dostęp: 20.08.2019]. Dostępny w Internecie:  
[https://www.sas.com/en\\_us/insights/big-data/what-is-big-data.html](https://www.sas.com/en_us/insights/big-data/what-is-big-data.html)
2. *What is Big Data?* [online] Oracle Corporation  
[dostęp: 20.08.2019]. Dostępny w Internecie: <https://www.oracle.com/big-data/guide/what-is-big-data.html>
3. *Analyzing Big Data in MicroStrategy* [online] MicroStrategy, Inc.  
[dostęp: 20.08.2019]. Dostępny w Internecie:  
[https://www2.microstrategy.com/producthelp/archive/10.7/WebUser/WebHelp/Lang\\_1033/Content/mstr\\_big\\_data.htm](https://www2.microstrategy.com/producthelp/archive/10.7/WebUser/WebHelp/Lang_1033/Content/mstr_big_data.htm)
4. Goczyła K., *Big Data i 5V – Nowe wyzwania w świecie danych VII Krajowa Konferencja Naukowa INFOBAZY 2014. Inspiracja - Integracja – Implementacja* Gdańsk: Politechnika Gdańska, Centrum Informatyczne TASK, 2014, s.21-23
5. *Big Data: The 5 Vs Everyone Must Know* [online] Bernard Marr  
[dostęp: 26.08.2019]. Dostępny w Internecie:  
<https://www.linkedin.com/pulse/20140306073407-64875646-big-data-the-5-vs-everyone-must-know>
6. *A Short History of Big Data* [online] Dr Mark van Rijmenam  
[dostęp: 26.08.2019]. Dostępny w Internecie: <https://datafloq.com/read/big-data-history/239>
7. *MapReduce: Simplified Data Processing on Large Clusters* [online] Google AI  
[dostęp: 26.08.2019]. Dostępny w Internecie: <https://ai.google/research/pubs/pub62>
8. *The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.* [online] The Apache Software Foundation.  
[dostęp: 27.08.2019]. Dostępny w Internecie: <https://hadoop.apache.org>
9. *Hadoop vs. Spark: The New Age of Big Data* [online] Ken Hess  
[dostęp: 27.08.2019]. Dostępny w Internecie: <https://www.datamation.com/data-center/hadoop-vs.-spark-the-new-age-of-big-data.html>
10. *Apache Spark™ is a unified analytics engine for large-scale data processing.* [online] The Apache Software Foundation.  
[dostęp: 27.08.2019]. Dostępny w Internecie: <https://spark.apache.org/>
11. *What Is Fast Data And Why Is It Important?* [online] Lightbend, Inc.  
[dostęp: 28.08.2019]. Dostępny w Internecie:  
<https://www.lightbend.com/blog/executive-briefing-what-is-fast-data-and-why-is-it-important>

12. *What Is Spark Streaming?* [online] Databricks Inc.  
[dostęp: 01.09.2019]. Dostępny w Internecie: <https://databricks.com/glossary/what-is-spark-streaming>
13. *Understanding Database Sharding* [online] DigitalOcean, LLC  
[dostęp: 03.09.2019]. Dostępny w Internecie:  
<https://www.digitalocean.com/community/tutorials/understanding-database-sharding>
14. *Breaking Down ACID Databases* [online] Zack Busch  
[dostęp: 04.09.2019]. Dostępny w Internecie: <https://learn.g2.com/acid-database>
15. *CAP Theorem and Distributed Database Management Systems* [online] Syed Sadat Nazrul [dostęp: 03.09.2019]. Dostępny w Internecie:  
<https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e>
16. *Sharding — MongoDB Manual* [online] MongoDB, Inc  
[dostęp: 04.09.2019]. Dostępny w Internecie:  
<https://docs.mongodb.com/manual/sharding/>
17. *What Is New About NewSQL?* [online] Gokhan Simsek  
[dostęp: 04.09.2019]. Dostępny w Internecie:  
<https://softwareengineeringdaily.com/2019/02/24/what-is-new-about-newsql/>
18. *Don't be confused between Concurrency and Parallelism* [online] Skrew Everything  
[dostęp: 07.09.2019]. Dostępny w Internecie: <https://medium.com/from-the-scratch/dont-be-confused-between-concurrency-and-parallelism-eac8e703943a>
19. *What is the difference between concurrency and parallelism?* [online] Apurva Thorat  
[dostęp: 07.09.2019]. Dostępny w Internecie:  
<https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism>
20. *Async and non blocking concepts in java. Is it possible to be asynchronous and blocking together?* [online] Alexandre  
[dostęp: 07.09.2019]. Dostępny w Internecie:  
<https://stackoverflow.com/questions/25316798/async-and-non-blocking-concepts-in-java-is-it-possible-to-be-asynchronous-and-b>
21. *I/O-bound vs CPU-bound in Node.js* [online] Panu Pitkamaki  
[dostęp: 07.09.2019]. Dostępny w Internecie: <https://bytearcher.com/articles/io-vs-cpu-bound/>
22. *Asynchronous programming with Go* [online] Gaurav Singha Roy  
[dostęp: 08.09.2019]. Dostępny w Internecie:  
<https://medium.com/@gauravsingharoy/asynchronous-programming-with-go-546b96cd50c1>



23. *What are microservices?* [online] Chris Richardson, Microservices.io  
[dostęp: 08.09.2019]. Dostępny w Internecie: <https://microservices.io/>
24. *The Reactive Manifesto* [online] reactivemanifesto.org  
[dostęp: 08.09.2019]. Dostępny w Internecie: <https://www.reactivemanifesto.org/>
25. *Reactive Programming versus Reactive Systems* [online] Lightbend, Inc.  
[dostęp: 09.09.2019]. Dostępny w Internecie: <https://www.lightbend.com/white-papers-and-reports/reactive-programming-versus-reactive-systems>
26. *Reactive Streams* [online] reactive-streams.org  
[dostęp: 09.09.2019]. Dostępny w Internecie: <https://www.reactive-streams.org/>
27. *Customer Case Studies* [online] Lightbend, Inc.  
[dostęp: 09.09.2019]. Dostępny w Internecie: <https://www.lightbend.com/case-studies>
28. *Why modern systems need a new programming model* [online] Akka  
[dostęp: 13.09.2019]. Dostępny w Internecie:  
<https://doc.akka.io/docs/akka/current/guide/actors-motivation.html>
29. *How the Actor Model Meets the Needs of Modern, Distributed Systems* [online] Akka [dostęp: 13.09.2019]. Dostępny w Internecie:  
<https://doc.akka.io/docs/akka/current/guide/actors-intro.html?language=scala>
30. *How are Akka actors different from Go channels? How are two related to each other?* [online] Rick Beton [dostęp: 25.09.2019]. Dostępny w Internecie:  
<https://www.quora.com/How-are-Akka-actors-different-from-Go-channels-How-are-two-related-to-each-other>
31. *Erlang programming language* [online] Erlang  
[dostęp: 25.09.2019]. Dostępny w Internecie: <https://www.erlang.org/>
32. *Akka Streams Introduction* [online] Akka  
[dostęp: 25.09.2019]. Dostępny w Internecie:  
<https://doc.akka.io/docs/akka/current/stream/stream-introduction.html>
33. *Artificial Intelligence (AI) vs. Machine Learning vs. Deep Learning* [online] Skymind [dostęp: 27.09.2019]. Dostępny w Internecie: <https://skymind.ai/wiki/ai-vs-machine-learning-vs-deep-learning>
34. *Scikit-learn Tutorial – Choosing the right estimator* [online] scikit-learn  
[dostęp: 27.09.2019]. Dostępny w Internecie: [https://scikit-learn.org/stable/tutorial/machine\\_learning\\_map/index.html](https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html)
35. *Models for machine learning* [online] IBM Developer  
[dostęp: 27.09.2019]. Dostępny w Internecie: <https://developer.ibm.com/articles/cc-models-machine-learning/>
36. *Machine Learning Algorithm - Backbone of emerging technologies* [online] Vishakha Jha [dostęp: 03.10.2019]. Dostępny w Internecie:

<https://www.techleer.com/articles/203-machine-learning-algorithm-backbone-of-emerging-technologies/>

37. *What is the difference between test set and validation set?* [online] Alexander Galkin [dostęp: 03.10.2019]. Dostępny w Internecie: <https://stats.stackexchange.com/questions/19048/what-is-the-difference-between-test-set-and-validation-set>
38. *Deep learning* [online] Sven Behnke [dostęp: 04.10.2019]. Dostępny w Internecie: [https://en.wikipedia.org/wiki/Deep\\_learning](https://en.wikipedia.org/wiki/Deep_learning)
39. *Artificial neural networks are changing the world. What are they?* [online] Graham Templeton [dostęp: 04.10.2019]. Dostępny w Internecie: <https://www.extremetech.com/extreme/215170-artificial-neural-networks-are-changing-the-world-what-are-they>
40. *Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning* [online] Vibhor Nigam [dostęp: 05.10.2019]. Dostępny w Internecie: <https://towardsdatascience.com/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90>
41. *Fifth generation computer* [online] Wikipedia [dostęp: 07.10.2019]. Dostępny w Internecie: [https://en.wikipedia.org/wiki/Fifth\\_generation\\_computer](https://en.wikipedia.org/wiki/Fifth_generation_computer)
42. *Artificial intelligence can 'evolve' to solve problems* [online] Matthew Hutson [dostęp: 07.10.2019]. Dostępny w Internecie: <https://www.sciencemag.org/news/2018/01/artificial-intelligence-can-evolve-solve-problems>
43. *What Are Overfitting and Underfitting in Machine Learning?* [online] Anas Al-Masri [dostęp: 07.10.2019]. Dostępny w Internecie: <https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690>
44. *What is Kubernetes* [online] The Kubernetes Authors [dostęp: 11.10.2019]. Dostępny w Internecie: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>
45. *The Advantages of Using Kubernetes and Docker Together* [online] Stackify [dostęp: 11.10.2019]. Dostępny w Internecie: <https://stackify.com/kubernetes-docker-deployments/>
46. *Building Microservices: Inter-Process Communication in a Microservices Architecture* [online] NGINX Inc. [dostęp: 12.10.2019]. Dostępny w Internecie: <https://www.nginx.com/blog/building-microservices-inter-process-communication/>
47. *gRPC Concepts* [online] gRPC [dostęp: 12.10.2019]. Dostępny w Internecie: <https://grpc.io/docs/guides/concepts/>
48. Kamil Owczarek, *Spark and Akka for Big Data Systems, in practice* Scalar Conference 2018 by SoftwareMill. Warszawa, 2018

49. *What is scalable programming language?* [online] Simone Pezzano [dostęp: 24.10.2019]. Dostępny w Internecie: <https://www.quora.com/What-is-scalable-programming-language>
50. *The Scala Programming Language* [online] École Polytechnique Fédérale Lausanne (EPFL) [dostęp: 26.10.2019]. Dostępny w Internecie: <https://docs.scala-lang.org/tour/tour-of-scala.html>
51. *Scala—Part of Lightbend Platform* [online] Lightbend [dostęp: 26.10.2019]. Dostępny w Internecie: <https://www.lightbend.com/scala-part-of-lightbend-platform>
52. *The SMACK Stack is the New LAMP Stack* [online] Edward Hsu, D2IQ [dostęp: 29.10.2019]. Dostępny w Internecie: <https://d2iq.com/blog/smack-stack-new-lamp-stack>
53. *Lagom Framework—Part of Lightbend Platform* [online] Lightbend [dostęp: 29.10.2019]. Dostępny w Internecie: <https://www.lightbend.com/lagom-framework-part-of-lightbend-platform>
54. *Lagom - Microservices Framework* [online] Lightbend [dostęp: 30.10.2019]. Dostępny w Internecie: <https://www.lagomframework.com>
55. *Cassandra – A structured storage system on a P2P Network* [online] Facebook Engineering [dostęp: 08.11.2019]. Dostępny w Internecie: <https://www.facebook.com/notes/facebook-engineering/cassandra-a-structured-storage-system-on-a-p2p-network/24413138919/>
56. *Introduction to Apache Cassandra's Architecture* [online] Akhil Mehra [dostęp: 09.11.2019]. Dostępny w Internecie: <https://dzone.com/articles/introduction-apache-cassandras>
57. *Five Steps to an Awesome Data Model in Apache Cassandra* [online] Scotch.io [dostęp: 09.11.2019]. Dostępny w Internecie: <https://scotch.io/tutorials/five-steps-to-an-awesome-data-model-in-apache-cassandra>
58. *Apache Cassandra – Manage massive amounts of data, fast, without losing sleep* [online] The Apache Software Foundation [dostęp: 12.11.2019]. Dostępny w Internecie: <http://cassandra.apache.org/>
59. *Big Data: Examples and Guidelines for the Enterprise Decision Maker* [online] MongoDB, Inc. [dostęp: 13.11.2019]. Dostępny w Internecie: <https://www.mongodb.com/collateral/big-data-examples-and-guidelines-enterprise-decision-maker>
60. *Apache Spark - Unified Analytics Engine for Big Data* [online] The Apache Software Foundation [dostęp: 14.11.2019]. Dostępny w Internecie: <https://spark.apache.org/>
61. *Spark Streaming Programming Guide* [online] The Apache Software Foundation [dostęp: 14.11.2019]. Dostępny w Internecie: <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

62. *MLlib is Apache Spark's scalable machine learning library*. [online] The Apache Software Foundation [dostęp: 18.11.2019]. Dostępny w Internecie: <https://spark.apache.org/mllib/>
63. *Moving from Big Data to Fast Data? Here's How To Pick The Right Streaming Engine* [online] Lightbend [dostęp: 22.11.2019]. Dostępny w Internecie: <https://www.lightbend.com/blog/moving-from-big-data-to-fast-data-heres-how-to-pick-the-right-streaming-engine>
64. *Deep Learning With Apache Spark — Part 1* [online] Favio Vázquez [dostęp: 22.11.2019]. Dostępny w Internecie: <https://towardsdatascience.com/deep-learning-with-apache-spark-part-1-6d397c16abd>
65. *Deep Learning with Apache Spark and TensorFlow* [online] Databricks [dostęp: 22.11.2019]. Dostępny w Internecie: <https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html>
66. *Industrializing AI & Machine Learning Applications with Kubeflow* [online] Tianxiang (Ivan) Liu [dostęp: 22.11.2019]. Dostępny w Internecie: <https://towardsdatascience.com/industrializing-ai-machine-learning-applications-with-kubeflow-5687bf56153f>
67. *Different kinds of service bus: command bus, service bus and query bus*. [online] Barry van Veen [dostęp: 01.12.2019]. Dostępny w Internecie: <https://barryvanveen.nl/blog/59-different-kinds-of-service-bus-command-bus-service-bus-and-query-bus>
68. *Apache Kafka – a distributed streaming platform* [online] Apache Software Foundation [dostęp: 02.12.2019]. Dostępny w Internecie: <https://kafka.apache.org/>
69. *Thorough Introduction to Apache Kafka™* [online] Stanislav Kozlovski [dostęp: 02.12.2019]. Dostępny w Internecie: <https://hackernoon.com/thorough-introduction-to-apache-kafka-6fbf2989bbc1>
70. *Apache Kafka – documentation* [online] Apache Software Foundation [dostęp: 02.12.2019]. Dostępny w Internecie: <https://kafka.apache.org/documentation.html>
71. *Modern Open Source Messaging: NATS, RabbitMQ, Apache Kafka, hmbdc, Synapse, NSQ and Pulsar* [online] Philip Feng Ph.D [dostęp: 03.12.2019]. Dostępny w Internecie: <https://medium.com/@philipfeng/modern-open-source-messaging-apache-kafka-rabbitmq-nats-pulsar-and-nsq-ca3bf7422db5>
72. *Domain-Driven Design – Object-Orientation Done Right* [online] Aslam Khan, Obi Oberoi [dostęp: 04.12.2019]. Dostępny w Internecie: <https://dzone.com/refcardz/getting-started-domain-driven>
73. *A Decade of DDD, CQRS and Event Sourcing* [online] tacta.io [dostęp: 04.12.2019]. Dostępny w Internecie: <https://tacta.io/a-decade-of-ddd-cqrs-and-event-sourcing/>

74. *1 Year of Event Sourcing and CQRS* [online] Teiva Harsanyi [dostęp: 05.12.2019]. Dostępny w Internecie: <https://hackernoon.com/1-year-of-event-sourcing-and-cqrs-fb9033ccd1c6>
75. *CQRS, Event Sourcing and DDD FAQ* [online] Edument AB [dostęp: 05.12.2019]. Dostępny w Internecie: <https://cqrs.nu/Faq>
76. *Are CQRS commands part of the domain model?* [online] Vladimir Khorikov [dostęp: 05.12.2019]. Dostępny w Internecie: <https://enterprisecraftsmanship.com/posts/cqrs-commands-part-domain-model/>
77. *Best Practices for Event-Driven Microservice Architecture* [online] Jason Skowronski [dostęp: 05.12.2019]. Dostępny w Internecie: <https://dzone.com/articles/best-practices-for-event-driven-microservice-archi>
78. *Domain Events vs. Event Sourcing – Why domain events and event sourcing should not be mixed up* [online] Christian Stetter [dostęp: 06.12.2019]. Dostępny w Internecie: <https://www.innoq.com/en/blog/domain-events-versus-event-sourcing/>
79. *Regression analysis* [online] Wikipedia [dostęp: 08.03.2020]. Dostępny w Internecie: [https://en.wikipedia.org/wiki/Regression\\_analysis](https://en.wikipedia.org/wiki/Regression_analysis)
80. *Decision tree learning* [online] Wikipedia [dostęp: 08.03.2020]. Dostępny w Internecie: [https://en.wikipedia.org/wiki/Decision\\_tree\\_learning](https://en.wikipedia.org/wiki/Decision_tree_learning)

## Spis rysunków

Rys.1. Model 5V – pięć obszarów Big Data .....	9
Rys.2. Komponenty strumieniowego systemu danych Fast Data opartego o Spark Streaming .....	14
Rys.3. Operacje zapisu / odczytu w <i>Sharded cluster</i> poprzez router <i>mongos</i> .....	17
Rys.4. Konkurencyjność osiągnięta samodzielnie oraz wraz z paralelizacją .....	19
Rys.5. Cechy systemu rozproszonego <i>Reactive System</i> .....	25
Rys.6. Funkcjonowanie aktorów oraz <i>message passing</i> w <i>Actor model</i> .....	29
Rys.7. Mechanizm nadzoru oraz struktura hierarchiczna drzewa aktorów .....	30
Rys.8. Główne modele uczenia maszynowego.....	35
Rys.9. Drzewo decyzyjne doboru metody ML dla danego problemu wg Scikit-learn .....	37
Rys.10. Głęboka sieć neuronowa, a rozpoznawanie wzorców poprzez warstwy .....	38
Rys.11. Ewolucja wdrożeń w kierunku wirtualizacji w chmurze .....	43
Rys.12. Dawny LAMP stack, nowy SMACK stack dla aplikacji biznesowych .....	53
Rys.13. Przechowywanie danych w pierścieniu węzłów Cassandra .....	56
Rys.14. Scenariusze zastosowania Big Data operacyjnego razem z analitycznym .....	57
Rys.15. Rodzaje silników strumieniujących wg <i>batch</i> , <i>mini-batch</i> i <i>low latency</i> ....	60
Rys.16. Rozproszony trening modeli TensorFlow oparty o Apache Spark .....	62
Rys.17. Ogólna architektura Publish / Subscribe w Apache Kafka .....	65
Rys.18. Architektura rozwiązania projektu <i>reactive-stock</i> .....	73
Rys.19. Uruchomienie interpolatora oraz obserwacja zamkniętych transakcji .....	90
Rys.20. Dane kryptowaluty Bitcoin przetwarzane przez interpolator .....	97
Rys.21. Inicjacja procesu treningu modelu uczenia maszynowego .....	99
Rys.22. Wynik treningu modelu z podglądem w strukturę lasu losowego oraz poziomem błędów RMSE .....	100
Rys.23. Wykres danych wykorzystanych w procesie treningu oraz ewaluacji modelu uczenia maszynowego.....	101
Rys.24. Wykres uzyskanej predykcji oraz danych rzeczywistych (historycznych) .....	102

## Spis kodów źródłowych

Listing nr 1. Konkurencyjność modelu <i>message passing</i> w Golang.....	22
Listing nr 2. Wybrane funkcjonalności oraz implementacje w języku Scala .....	50
Listing nr 3. Fragment konfiguracji modułów projektu w <i>build.sbt</i> .....	76
Listing nr 4. Konfiguracja mikroservisu <i>Transaction Service</i> .....	78
Listing nr 5. Interfejs ( <i>trait</i> ) <i>TransactionService</i> mikrouслуги transakcji .....	78
Listing nr 6. Implementacja <i>TransactionServiceImpl</i> mikrouслуги transakcji .....	80
Listing nr 7. Zawartość <i>TransactionAggreagate</i> , <i>Behavior</i> oraz <i>State</i> .....	82
Listing nr 8. Implementacja <i>AssetEventProcessor</i> oraz <i>AssetRepository</i> .....	84
Listing nr 9. Deskryptory pozostałych mikroservisów: <i>Trader</i> , <i>Asset</i> oraz <i>Table</i> ..	85
Listing nr 10. Klient REST wykonujący zapytania na API systemu reaktywnego..	88
Listing nr 11. Konfiguracja środowiska analitycznego w <i>docker-compose.yml</i> .....	92
Listing nr 12. Model uczenia maszynowego regresji ceny transakcji.....	93