Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Sterowania
i Elektroniki Przemysłowej
Zakład Sterowania

# Praca dyplomowa magisterska

na kierunku Automatyka i Robotyka

## Badania wydajności uczenia w oparciu o rozproszoną platformę Kubernetes

Szymon Krasuski
nr albumu 279103

promotor
dr inż. Waldemar Graniszewski

Warszawa 2020

## Badania wydajności uczenia w oparciu o rozproszoną platformę Kubernetes

# Streszczenie

Niniejsza praca zawiera analizę możliwości implementacji architektury rozproszonego systemu uczenia maszynowego. Pierwsza część pracy opisuje historię rozwoju idei i potrzeb uczenia maszynowego oraz przedstawia ewaluację sposobów podejścia do tego zagadnienia.
Druga część pracy poświęcona jest badaniu wydajności systemu rozproszonego zbudowanego przez autora w oparciu o platformę Kubernetes. Zaproponowane rozwiązanie zbudowane na dedykowanych kontenerach używających biblioteki Tensorflow i wdrożonych za pomocą serii manifestów przystosowanych do platformy Helm.
Przygotowane rozwiązanie przetestowano dla różnej liczby węzłów w rozproszonym systemie przy referencyjnym modelu sieci neuronowej uczonym pod standardową bibliotekę pisma odręcznego MNIST. Pod uwagę brano czas uczenia referencyjnego modelu przy stałej liczbie epok uczenia, stałej liczbie jednej z jednostek w samej implementacji infrastruktury oraz zmiennej liczbie drugiego typu jednostek infrastruktury. Badania przeprowadzono zarówno na lokalnej infrastrukturze, jak i platformie chmurowej Google Cloud.
Wyniki badań pokazały, że zwiększenie zasobów dostępnych w całym rozproszonym systemie pozwala równolegle rozłożyć proces uczenia z minimalną stratą mocy obliczeniowej na procesy związane z utrzymaniem samej infrastruktury. Wykazano także, że skalowanie rozproszonej platformy Kubernetes i związanych z nią procesów zarządzających ma minimalny wpływ na wydajność procesu uczenia.
Praca pokazuje kierunek, w którym w najbliższych latach być może będzie rozwijał się rynek usług i produktów związanych z uczeniem maszynowym, zarówno z perspektywy technicznej, jak i biznesowej niniejsza.

**Słowa Kluczowe**: Uczenie maszynowe, infrastruktura rozproszona, Kubernetes, Helm, Google Cloud Platform, Tensorflow

**Learnig performance study based on distributed Kubernetes platform**


# Abstract

This paper analizes possible implementation of distributed machine learning system architecture. Afterwards, research on perfomarance of machine learning process based on previously developed system was conducted.
Used machine learning infrastructure was based on Kubernetes platform. Proposed solution was developed with dedicated containers using Tensorflow library and deployed with series of manifests prepared for Helm platform.
Developed system architecture was tested for various number of nodes in distributed infrastructure with reference neural network model based on standard hand writing library MNIST. Analyzed was learning time of reference model with constant number of epochs, contant number of one of unit types used in system implementation and variable number of the other infrastructure unit type. Reaserch was conducted both on local infrastructure and also on Google Cloud Platform.
Results show that increase of computing resources available in whole distributed system allows to split machine learning process evenly on all nodes with minimal loss of computing power for infrastructure management processes. Moreover, research resulted with minimal impact of scaling distributed Kubernetes platform and involved management processes to performance of machine learning process.
This paper shows direction in which will lead market of machine learning services and products in the upcoming years either from technical or business perspective.



**Key words**: Machine learning, distributed infrastructure, Kubernetes, Helm, Google Cloud Platform, Tensorflow

Warszawa, 20.06.2020

POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY

**OŚWIADCZENIE**

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa inżynierska pt. Badania wydajności uczenia w oparciu o rozproszoną platformę Kubernetes:

- została napisana przeze mnie samodzielnie,
- nie narusza niczyich praw autorskich,
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej. Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Szymon Krasuski.....

# Spis treści

# Rozdział 1

# Wstęp

## 1.1 Wprowadzenie

Coraz bardziej dynamiczny rozwój technologii sprawił, że jest ona obecna praktycznie wszędzie dookoła człowieka w postaci wielu urządzeń oraz serwisów. Technologia ta agreguje potężne ilości danych każdego dnia. Według raportu z 2017 [1][1] każdego dnia w sieci produkowane jest 2.5 miliona terabajtów danych. Niemożliwe jest zatem, żeby człowiek lub grupa ludzi przeanalizowała każdego dnia tyle danych. Z tego powodu coraz popularniejsze stają się rozwiązania pozwalające na zautomatyzowanie analizy pozyskanych danych i wygenerowanie wzorców możliwych do późniejszego wykorzystania. Dynamicznie rozwijającą się gałęzią takich rozwiązań jest uczenie maszynowe, a zwłaszcza sieci neuronowe.

W 1943 roku W.McCulloch oraz W.Pitts przedstawili model matematyczny neuronu, który został następnie użyty w implementacji neuronów wykorzystywanych w sieciach neuronowych. Matematyczny opis działania tego neuronu bazuje na kilku sygnałach wejściowych $x$, gdzie każdy z sygnałów posiada swoją wagę $w$. W modelu występuje również waga zerowa, która jest stałą dodawaną do wartości wejścia neuronu tzw. *bias*. W centrum neuronu znajduje się funkcja aktywacji, która produkuje na wyjściu sygnał bazujący na sumie sygnałów wejściowych przemnożonych przez wagi każdego z nich. Tak zdefiniowany model wykorzystywany jest do budowania warstw sieci, gdzie znajduje się wiele neuronów, a kolejne warstwy sieci jako sygnał wejściowy przyjmują sygnał wyjściowy poprzedniej warstwy. Następnie taka sieć jest trenowana poprzez wielokrotną prezentację danych wejściowych razem ze

[1]Wykonanym i udostępnionym jako infografika w mediach społecznościowych przez firmę Domo

spodziewanym wyjściem sieci. Sieć w procesie uczenia dostosowuje wartości wag w taki sposób, żeby jak najlepiej odwzorować charakterystykę funkcji przejściowej.

Wraz z rozwojem sieci neuronowych zaczęto wykorzystywać coraz głębsze modele, czyli takie posiadające coraz więcej warstw. Niestety wraz ze wzrostem neuronów w sieci, należy wykonać więcej operacji w procesie uczenia i wyliczyć wartość większej liczby wag. Do bardzo dużych modeli zaczęto wykorzystywać wyspecjalizowane maszyny opierające się na kartach graficznych, których wielordzeniowe procesory pozwalają wykonać wiele operacji równolegle. Maszyny takie są stosunkowo drogie i nie są szeroko dostępne. Jednak wiele firm opiera swoje produkty o uczenie maszynowe - np. Spotify, firma słynąca z wykorzystania uczenia maszynowego do analizy muzyki.

## 1.2 Cel pracy

W odpowiedzi na potrzeby rynku, gdzie wymagane jest minimalizowanie czasu uczenia maszynowego (najczęściej ze względu na potrzebę zwiększania niezawodności i szybkości dostarczenia gotowego produktu oraz minimalizowaniu kosztów obsługi) zaczęto budować systemy uczenia rozproszonego.
W niniejszej pracy rozpatrywane są możliwości zbudowania infrastruktury do rozproszonego uczenia maszynowego oraz analizowany jest wpływ skalowania tej infrastruktury na czas uczenia referencyjnego modelu sieci neuronowej.
Rozwiązanie spełniające aktualne trendy rynkowe powinno spełniać dwa kryteria.
Pierwszym z nich jest maksymalne wykorzystanie dostępnych zasobów obliczeniowych do procesu uczenia. Jednocześnie procesy utrzymujące oraz zarządzające rozproszoną infrastrukturą powinny zajmować jak najmniejszą część dostępnych zasobów, a ich skalowanie nie powinno wpływać na wydajność procesu uczenia maszynowego.
Drugim kryterium jest swobodne skalowanie infrastruktury wykorzystanej do uczenia maszynowego. Jest to jednoznaczne z poprawą czasu uczenia przy zwiększeniu (zazwyczaj dodaniu kolejnych instancji do infrastruktury) dostępnych zasobów, pamiętając o ograniczeniach wynikających z architektury samego trenowanego modelu.
Jeśli powyższe kryteria zostaną spełnione, to rozwiązanie będzie odpowiednie do zastosowań biznesowych.

Struktura pracy została podzielona na trzy części.
Pierwsza z nich składa się na analizę produktów dostępnych na rynku w rozwiązaniu problemu omawianego w niniejszej pracy. Rozdział 2 skupia się na możliwościach technologii konteneryzacji w platformach rozproszonych. Rozdział 3 analizuje jakość zastosowania platformy Kubernetes do zarządzania kontenerami na rozproszonym środowisku. Rozdział 4 skupia się na sposobie dostarczenia finalnego rozwiązania ułatwiającego wdrożenie na różnorodne środowiska.
W kolejnej części zaprezentowano techniczny aspekt budowy rozwiązania uczenia maszynowego na rozproszonym środowisku. Rozdział 5 zawiera szczegółowy opis skryptów uczenia maszynowego, realizacji infrastruktury sieciowej oraz budowy środowisk testowych.
Ostatnia część pracy skupia się na badaniu zaprezentowanego rozwiązania i analizy wyników. Rozdział 6 opisuje zastosowaną strategię badań oraz otrzymane wyniki. W rozdziale 7 omówiono wyniki i zawarto wnioski.

# Rozdział 2

# Analiza możliwości wykorzystania technologii konteneryzacji do uczenia maszynowego

## 2.1 Przegląd Literatury

Konteneryzacja staje się coraz popularniejszą formą budowania architektury nowoczesnych systemów komputerowych [2]. Wykorzystanie takich narzędzi, jak Docker stało się wręcz standardem dla systemów bazowanych chmurowo. Dzięki wysokiemu stopniowi izolacji kontenerów od systemu zarządzającego połączonemu z względną lekkością operacyjną (która w porównaniu z np. tradycyjnymi maszynami wirtualnymi ma znacznie lepszą wydajność) częste stało się implementowanie kilku, a nawet kilkunastu kontenerów w ramach jednej aplikacji.

Jednym z głównych wyzwań w procesie rozwijania technologii mikroserwisów opartej na konteneryzacji jest usystematyzowane zarządzanie kontenerami na wielu maszynach równocześnie [3][1]. Wiele niezależnych maszyn pozwala na szeroką dostępność (która jest ważna w przypadku, gdy jedna z maszyn zawiedzie). Również ze względów ekonomicznych, korzystna jest możliwość wykorzystania wielu maszyn o nieco gorszej specyfikacji, niż jednej maszyny o czołowych parametrach na rynku. Docker, który jest obecnie jednym z najpopularniejszych silników służącym do konteneryzacji niestety nie posiada odpowiednich narzędzi do obsługi systemów rozproszonych. W tym celu stworzona została platforma Kubernetes, która oparta jest na swoim przodku - platformie Borg. Platforma ta powstała na potrzeby wewnętrzne

[1] przygotowanym w 2015 na podstawie podcastu Software Engineering Radio

pracowników firmy Google w celu prostej implementacji aplikacji developerskich jeszcze na etapie testów.

Kubernetes jest bezsprzecznie jedną z najszybciej rozwijających się platform, która pozostawia w tyle inne konkurujące rozwiązania. Kubernetes pozwala optymalnie zarządzać rozlokowaniem kontenerów w infrastrukturze komputerowej oraz zapewnia wszelkie wymagane standardy dla nowoczesnych aplikacji wykorzystywanych w restrykcyjnych sektorach oprogramowania, jak na przykład bankowość [4].

Na podstawie badań z 2018 [5], gdzie zbadano wydajność sieciową grupy maszyn połączonych za pomocą platformy Kubernetes, wykazano że jest ona bardzo elastyczna sieciowo, a także nie wymaga znacząco większej przepustowości, niż takiej jaka jest choćby w sieciach domowych. Badania pokazują również, że wzrost ilości maszyn połączonych platformą tylko nieznacznie wpływa na wydajność sieci.

Przy wydatnej infrastrukturze sieciowej, która pozwala wykonywać aplikacje na wielu maszynach jednocześnie od razu nasuwa się na myśl o wykorzystaniu jej do celów uczenia maszynowego. Uczenie maszynowe jest znane z tego, że wymaga dużej pamięci obliczeniowej w celu nauczenia modelu. Przez ostatnie lata wiele firm inwestowało w maszyny z rozbudowanymi procesorami bazującymi na kartach graficznych, które rzeczywiście były wydajne, ale niestety były również bardzo drogie. Posiadając taką platformę, jak Kubernetes, możliwe staje się wykorzystanie wielu tańszych maszyn o gorszej specyfikacji i osiągnięciu podobnych wyników uczenia maszynowego.

W 2018 do platformy Kubernetes została w wersji testowej wprowadzona skrzynka narzędziowa Kubeflow, która ma za zadanie zarządzać wykonywaniem jednego zadania na wielu maszynach równolegle. W 2019 roku to rozwiązanie zostało oficjalnie wydane w wersji stabilnej i zostało zauważone jako doskonałe narzędzie do implementacji rozproszonego uczenia maszynowego. Narzędzie Kubeflow zostało opisane jako bardzo intuicyjne i pozwalające w prosty sposób konfigurować zaawansowane aspekty procesu uczenia maszynowego oraz uruchamiać sam proces w praktycznie dowolnym środowisku [6].

Niestety zastosowanie już nieco bardziej zaawansowanych metod uruchamiania aplikacji, niż proste kontenery wymagają uruchomienia kilku sekwencji lub aplikacji których nie można standardowo zawrzeć w jednym pliku konfiguracyjnym aplikacji czytanym przez platformę Kubernetes. W tym celu powstała platforma Helm, która służy do uruchamiania bardziej zaawanso-

wanych aplikacji za pośrednictwem platformy Kubernetes. Analiza jakości aplikacji wdrożonych za pomocą Helm Charts (Chart jest określany jako niezależna aplikacja lub zestaw aplikacji możliwe do wdrożenia za pomocą pojedynczej paczki z katalogu platformy Helm) pokazuje, że Helm pozwala w prosty sposób na wdrażanie aplikacji w wielu różnych środowiskach [7]. Wyraźnie widać również, że dodanie kolejnej warstwy abstrakcji, jaką jest Helm, do już i tak zbudowanej na wielu warstwach platformy Kubernetes nie wpływa na wydajność samej platformy, a wręcz w niektórych przypadkach może pozytywnie wpłynąć na jej wydajność.

Razem z odpowiednią architekturą nadającą się do wdrożenia najbardziej wymagających aplikacji potrzebne jest również zapewnienie odpowiednich narzędzi do uczenia maszynowego, które będą dawać możliwość wykorzystać w pełni możliwości infrastruktury. Na rynku aktualnie znajduje się wiele narzędzi pozwalających na budowanie aplikacji uczenia maszynowego. Jednym z najpopularniejszych z nich jest pakiet Tensorflow od firmy Google. Ponieważ firma Google jest twórcą platformy Kubernetes (chociaż sama platforma jest rozwijana na zasadach wolnego oprogramowania OpenSource), to pakiet Tensorflow również od tej firmy jest najlepiej zintegrowany z platformą. Od roku 2015 (gdy firma Google udostępniła narzędzie Tensorflow na prawach wolnego oprogramowania OpenSource) wokół narzędzia Tensorflow skupiła się bogata społeczność aktywnie rozwijająca narzędzie o nowe funkcjonalności [8]. Dzięki otwartości oprogramowania narzędzia współpracuje ono szeroko z całą gamą produktów również innych największych firm na rynku. Tensorflow ma swoje zastosowania w specjalistycznych aplikacjach zarówno w rynkach związanych z przemysłem i fabrykami, jak i w rynkach typowo informatycznych.

W roku 2017 firma Google udostępniła moduł do narzędzia Tensorflow o nazwie Tensorflow Serving, który pozwala na rozproszone wykorzystywanie modeli nauczonych za pomocą narzędzia Tensorflow [9]. Praca przedstawia możliwości tego modułu i wskazuje na nowe możliwości implementacji modeli uczenia maszynowego na systemach chmurowych. Moduł Tensorflow Serving pozwala na wykorzystanie np. pojedynczego modelu sieci neuronowej do analizy danych przez rozproszony system węzłów - utworzony np. w oparciu o platformę Kubernetes. Dzięki takiej architekturze możliwe jest równoległe analizowanie wielu danych oraz dynamiczne alokowanie i skalowanie zasobów w zależności od potrzeb aplikacji.

W momencie powstawania niniejszej pracy, w fazie początkowej prac nie istniał dedykowany moduł do narzędzia Tensorflow, który pozwalałby w pro-

sty sposób przeprowadzić proces uczenia jedynie dedykowane rozwiązania pozwalające na dystrybucję gotowego modelu do przewidywania danych i równoważenia zapytań do zwielokrotnionych węzłów. Analizując możliwe rozwiązania tego problemu oraz wyzwania, które musi spełnić odpowiednie narzędzie odpowiadające na aktualne potrzeby rynku uczenia maszynowego [10], jednym z głównych wyzwań jest równoległe uaktualnienie wag modelu uczenia maszynowego, które z pozoru wygląda na niemożliwe do zrównoleglenia. W samym artykule jednak przedstawione jest kilka możliwości pośredniego rozwiązania tego problemu i strategii łączenia ze sobą podmodeli wyuczonych na kilku węzłach grupy maszyn.

W czasie pisania niniejszej pracy zaczęły również pojawiać się rozwiązania pozwalające trenować model w architekturze rozproszonej. Sama dokumentacja narzędzia Tensorflow zyskała dodatkowy rozdział [11] z przykładowym programem pozwalającym na uruchomienie równoległego uczenia na tej samej maszynie, co może być wykorzystane jako baza do realizacji programu na rozproszoną platformę wielu węzłów.
Ze strony firmy Google Cloud Platform pojawił się również w dokumentacji przykład [12] z wdrożeniem pozwalającym na rozproszone uczenie maszynowe modelu. Architektura rozwiązania jest bardzo podobna do rozwiązania zaprezentowanego w niniejszej pracy, ale wykorzystuje wbudowane narzędzia Google Cloud Platform, co nie jest uniwersalnym rozwiązaniem i nie może być wykorzystane na innych platformach.
Bardziej uniwersalne rozwiązanie pojawiło się w projekcie KubeFlow [13], który dodał dedykowany operator kubernetesowy pozwalający na rozproszone uczenie maszynowe na dowolnym gronie węzłów platformy Kubernetes. Jest to jednak inne rozwiązanie architektoniczne, niż zaprezentowane w niniejszej pracy.

Jak widać rozproszone uczenie maszynowe dynamicznie rozwija się na rynku i jest to tylko kwestia czasu, kiedy pojawią się dedykowane narzędzia do tej funkcji. Przykładowo firma Banzai Cloud już teraz prezentuje prototypy rozwiązań uczenia maszynowego opartego o rozproszoną architekturę i prawdopodobnie w niedalekiej przyszłości będą oferować dedykowane rozwiązanie biznesowe.

## 2.2 Przegląd istniejących rozwiązań

Przez lata powstało wiele koncepcji dystrybucji oprogramowania jako spójnej całości. Aktualnie standardem dla systemów operacyjnych jest forma

serwisów zarządzanych przez wspólny system. Przykładem takiego systemu może być *systemd* [14]. Jest to popularne narzędzie zarządzające serwisami na systemie operacyjnym Linux. Wiele aplikacji jest dostarczanych właśnie jako serwis, czyli definicja programu, który zostaje uruchomiony i okresowo sprawdzany przez narzędzie zarządzające. Posiada również komendy do zatrzymania oraz zakończenia serwisu, które mogą być wywołane w wypadku awarii lub np. zamknięcia systemu operacyjnego w bezpieczny sposób Prostota działania takiego systemu długo była bardzo popularna w aplikacjach produkcyjnych klasy biznes. Aplikacje takie były dzielone na wiele tzw. mikroserwisów działających równolegle i komunikujące się ze sobą. Jednak nadal każdy z serwisów korzystał ze środowiska systemu operacyjnego działającego na serwerze. Niosło to za sobą wiele niebezpieczeństw, że nieodizolowane serwisy nie są dostatecznie bezpieczne dla aplikacji o podwyższonym ryzyku (przykładowo aplikacji obsługujących systemy bankowe). Działanie wielu serwisów w tym samym środowisku niesie za sobą niebezpieczeństwo, że w przypadku luki bezpieczeństwa w programie jednego z nich, wszystkie inne serwisy są również zagrożone.

Powstawały więc inne metody budowania architektury aplikacyjnej, która była już bardziej odizolowana. Kolejnym punktem zwrotnym rozwoju infrastruktur serwerowych były rozproszone maszyny wirtualne. Był to ogromny krok technologiczny, który pozwolił wyzwolić się od ograniczeń narzuconych przez fizyczne urządzenia. Technologia maszyn wirtualnych polega na programowej wirtualizacji sprzętu, a następnie na instalację na nim systemu operacyjnego. Pozwala to na utworzenie na bazie systemu nadrzędnego wielu maszyn wirtualnych pracujących na różnych systemach operacyjnych (przykładowo na fizycznym sprzęcie pracującym pod systemem Linux można uruchomić zarówno maszyny wirtualne z system Windows, jak i Linux). Charakterystyczne dla maszyn wirtualnych jest fakt, że mogą one pracować połączone w podsieć obsługiwaną przez system bazowy lub jako samodzielny obiekt w sieci. W przypadku obsługi podsieci na systemie bazowej, tworzy się sieć z adresem prywatnym, a system bazowy działa jako router krańcowy tej sieci. W sytuacji, gdy maszyna jest ustawiona w tryb 'Bridged Adapter' - czyli adaptera mostkowego - jest podłączona logicznie jako oddzielne urządzenie w sieci systemu bazowego na równych prawach i z osobnym adresem IP.

Przez lata powstało wiele rozwiązań pozwalających na zarządzanie i automatyzację wirtualizacji maszyn, jak VirtualBox, czy VMware.Rozwiązania te są dedykowane głównie dla systemów serwerowych, gdzie możliwe jest ustawienie fizycznej architektury w dowolną konfigurację, a następnie progra-

mowe budowanie pożądanej infrastruktury. Technologia wirtualizacji istniała na rynku od dłuższego czasu, ale dopiero w przeciągu ostatnich lat zaczęto ją wykorzystywać do budowania rozproszonych systemów na wielu serwerach - platformy takie jak np. Openstack [15] (który pierwszą datę wydania miał w 2010 roku) pozwoliły na wdrożenie wielu serwisów w odizolowanych środowisku maszyn wirtualnych. Dzięki temu rozwiązaniu zaczęły powstawać nowe gałęzie rynku, gdzie firma buduje centrum serwerowe i oferuje klientowi dostęp do zwirtualizowanych zasobów, które mogą być skalowane według potrzeb i być dowolnie dobierane.

Podstawową zaletą opisanych narzędzi jest swoboda wyboru systemu operacyjnego maszyny wirtualnej, ponieważ zwirtualizowany osprzęt [16] jest optymalnie dobierany pod docelowy system. Pozwala to na uruchamianie aplikacji w bardzo uniwersalnym środowisku, gdzie nie wystąpi problem np. braku odpowiednich sterowników do obsługi jednej z kart na płycie głównej. Pozwala to także na uruchamianie systemów i aplikacji napisanych na sprzęt, który już dawno nie jest produkowany.
Kolejną zaletą tego rozwiązania jest niski koszt implementacja systemu jako maszyny wirtualnej. Oznacza to, że mają dostęp do zasobów obliczeniowych możliwe jest uruchomienie maszyny wirtualnej bez ponoszenia dodatkowych kosztów na osprzęt fizyczny. Jedynym kosztem może być jedynie koszt licencji na oprogramowanie systemu operacyjnego. Również w tym przypadku maszyny wirtualne mają swoje zalety, ponieważ najczęściej taka licencja może być wielokrotnie wykorzystywana do tworzenia kolejnych maszyn wirtualnych (przy uprzednim usunięciu poprzedniej maszyny).
Maszyny wirtualne charakteryzują się tym, że są bardzo dobrze izolowane od systemu bazowego. Systemy maszyny wirtualnej nie jest świadomy, że operuje na zwirtualizowanym osprzęcie, a program wirtualizujący maszyny nie zapewnia połączeń programowych z systemem bazowym. Nawet udostępnianie folderów i plików między systemem bazowym, a maszyną wirtualną jest rozwiązane na bazie dysków sieciowych. Pozwala to na bezpieczne uruchamianie aplikacji (nawet tych niekoniecznie bezpiecznych) i testowanie w bezpiecznym środowisku.
Obrazy maszyn wirtualnych są trzymane w systemie bazowym jako plik, co pozwala na szybkie i bezproblemowe przenoszenie maszyn wirtualnych między systemami bazowymi, a także tworzenie wielu klonów takich maszyn. Fakt ten jest często wykorzystywany przez firmy oferujące wirtualizację w swoich centrach danych, które oferują szereg obrazów maszyn często już z prekonfigurowanym środowiskiem.

Podstawową wadą maszyn wirtualnych jest stałe zajęcie zasobów fizycz-

nych maszyny [17], na której zwirtualizowano system. Oznacza to, że przydzielając maszynie wirtualnej daną ilość pamięci RAM i dysku roboczego, będą one zajęte całkowicie przez maszynę wirtualną podczas pracy nawet wtedy, gdy maszyna korzysta jedynie z niewielkiej części tych zasobów. Generuje to problem, kiedy maszyna sporadycznie zwiększa zapotrzebowanie na zasoby (przykładowo, gdy więcej ludzi, niż zazwyczaj, próbuje dostać się do serwera www), to musi na stałe mieć przydzieloną największą liczbę zasobów jaką będzie potrzebować i będą one zajmowane cały czas. Blokuje to wówczas możliwość swobodnego skalowania zużywanych zasobów. Co prawda możliwa jest zmiana przydzielonych zasobów, ale wymaga to ponownego uruchomienia maszyny wirtualnej, co przy systemach serwerowych nie jest zalecaną operacją.
Inną wadą jest zależność maszyn wirtualnych od systemu bazowego. W przypadku, gdy system bazowy wymaga modernizacji lub naprawy, wymagane jest wyłączenie wszystkich wirtualnych serwerów. Generuje to też dużo większą ilość elementów do utrzymania. W przypadku dedykowanych serwerów do utrzymania był tylko jeden system, a w technologii wirtualizacji wymagane jest utrzymanie zarówno systemu na maszynie fizycznej, jak i zwirtualizowanych systemów. Oczywiście narzędzia do zarządzania wirtualizacją znacznie zmniejszają potrzebny nakład pracy, ale nadal wymaga to bardziej skomplikowanej wiedzy.
Wadą jest też konieczność przejścia procesu instalacji systemu operacyjnego na maszynie wirtualnej, co często jest czasochłonne, a uwzględnić trzeba dodatkowo czas potrzebny na instalację komponentów do danej aplikacji. Rozwiązaniem tego problemu po części jest dystrybuowanie gotowych obrazów dysków, co znacznie ułatwia pracę w przypadku identycznych ustawień maszyny do kilku aplikacji. W przypadku, gdy jednak aplikacja wymaga zmian, to obraz maszyny nie jest przydatny.

Nową popularną technologią, która zaczyna zastępować maszyny wirtualne, jest konteneryzacja [18]. Kontener jest również sposobem wirtualizacji środowiska do uruchomienia aplikacji, ale w przeciwieństwie do maszyn wirtualnych rezygnuje z wirtualizacji warstwy sprzętowej i samego systemu operacyjnego. Procesy uruchamiane w kontenerze są widoczne jako zwykłe procesy bazowego systemu operacyjnego, ale kontener zapewnia ich izolację. Jest to słabsza izolacja od tej stosowanej w maszynach wirtualnych, ale często wystarczająca. Zrezygnowanie z wirtualizacji sprzętu i systemu operacyjnego wiąże się z tym że uruchamiana aplikacja musi być kompatybilna z fizycznym sprzętem i systemem bazowym.
Technicznie kontenery korzystają z bazowych obrazów systemów operacyjnych, które są bardzo okrojone, a przez to bardziej lekkie. Dopasowanie

konfiguracji takiego systemu bazowego jest możliwe za pomocą Dockerfile. Dockerfile jest skryptem określającym szereg operacji do wykonania na danym obrazie systemu i zwrócenia nowego obrazu systemu już z potrzebną konfiguracją.
Kontenery są skupione w podsieć na systemie bazowym, ale możliwa jest również w prosty sposób przekierowania portów warstwy transportowej kontenera do dowolnie wybranych portów systemu bazowego. Oznacza to, że za pomocą adresu systemu bazowego i odpowiedniego portu można komunikować się z kontenerem z poziomu Internetu.
Obecnie najbardziej popularnym narzędziem do konteneryzacji jest Docker. Projekt rozpoczęty w 2013 roku zrewolucjonizował rynek i obecnie jest wykorzystywana w całym szeregu zastosowań. Często na jednym systemie działa kilkanaście, a nawet kilkadziesiąt kontenerów z różnymi aplikacjami. W ostatnim czasie technologia konteneryzacji otwiera się również dla zwyczajnych konsumentów w postaci dodatków do przeglądarek [19], gdzie każda zakładka otwierana jest w oddzielonym kontenerze. Dzięki temu np. programy śledzące aktywność w przeglądarce ze stron Facebook'a są zamknięte w kontenerze i nie mają dostępu do innych aktywności poza stroną Facebook'a.

Pamięć zajmowana przez kontener jest na poziomie kilkudziesięciu MB. Jest to ogromna oszczędność miejsca w porównaniu z maszynami wirtualnymi [20], gdzie system zajmował często po kilka lub kilkadziesiąt GB pamięci. Zaleta ta przekłada się na większą ilość kontenerów uruchomionych na jednym systemie bazowym.
Kontenery zużywają dużo mniej pamięci, niż maszyny wirtualne [18]. Poprawia to wydajność sprzętu, na którym uruchamiane są kontenery oraz pozwala na szybsze działanie uruchomionej aplikacji. Również działanie aplikacji jako proces w systemie bazowym pozwala na dynamiczne wykorzystywanie zasobów. Oznacza to, że nie trzeba na stałe przypisywać do kontenera odpowiedniej ilości zasobów, tylko będzie on dynamicznie zajmował potrzebną ilość obiektów potrzebnych do poprawnego działania aplikacji. Niestety oznacza to dużo mniejszą kontrolę nad zużyciem zasobów, ale w ostatnim czasie developerzy oprogramowania do konteneryzacji starają się rozwiązać ten problem przez ręczne dopisywanie ograniczeń możliwych obiektów do zajęcia.
Czas potrzebny na uruchomienie i konfigurację kontenera, to kilka lub kilkanaście sekund. Jest to ogromne usprawnienie wirtualizacji i pozwala na tworzenie spersonalizowanych kontenerów 'na żywo', czyli przy zapytaniu o aplikację można w kilka sekund skonfigurować środowisko i uruchomić aplikację.
Kontenery są bardzo proste do migracji między urządzeniami. Jest to wypadkowa wynikająca z małego rozmiaru, jak i krótkiego czasu potrzebnego na

uruchomienie. Oprogramowanie typu docker oferuje wręcz portal [21] bazy skompilowanych obrazów kontenerów lub wręcz samych Dockerfile'ów [22] do samodzielnego skompilowania na swojej maszynie. Systemy testowe (jak np. Zuul) często opierają się na pobieraniu kilku kontenerów, uruchomieniu ich i przeprowadzeniu testów nowego oprogramowania. Jest to szybki sposób na sprawdzenie, czy nowy kod nie zawiera błędów.
Kontenery pomagają znacznie ograniczyć koszty. Nie trzeba płacić za licencję za system operacyjny [23], a ograniczenie potrzebnych zasobów pozwala na ograniczenie wydatków na fizyczne zasoby.

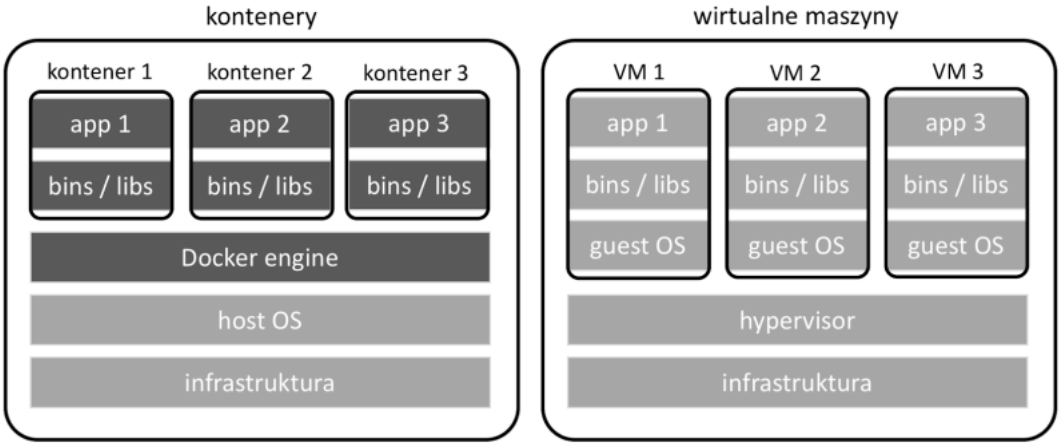
Podstawową wadą konteneryzacji jest słabsze bezpieczeństwo [20] w porównaniu do maszyn wirtualnych. Kontenery pracują na tym samym jądrze systemu i sprzęcie, co system bazowy. Wszystkie podatności w systemie bazowym lub kernelu są także obecne w kontenerach. Jest to jednak nadal bezpieczeństwo na wyższym poziomie, niż stosowanie dedykowanych serwerów i często wystarczające dla stosowanych aplikacji.
Kontenery mają mniejszą elastyczność obsługiwanego systemu operacyjnego [20]. System pracujący w kontenerze musi być taki sam, jak system bazowy. Wymaga to zakupu i uruchomienia innej maszyny fizycznej, jeśli wymagane jest korzystanie z innego systemu operacyjnego. Nie jest to zazwyczaj problem dla dostawców rozwiązań serwerowych, ponieważ korzystają oni zazwyczaj wyłącznie z systemu Linux, choć może to być problematyczne w zastosowaniach dla użytkowników indywidualnych. Projekt Docker oferuje uruchomienie 'kontenera' z systemem Linux na systemie Windows, ale jest to w rzeczywistości uruchomienie maszyny wirtualnej, która jest możliwa do zarządzania z poziomu Docker'a. Jest to spowodowane tym, że środowisko biznesowe Microsoft'u próbuje zdobyć wszelkimi siłami rynek serwerowy, który jest zajęty w większości przez systemów typu Linux.
Kolejną wadą jest połączenie kontenerów w sieć. Kontener działa w podsieci i jedyną możliwością komunikacji z Internetem jest połączenie portów kontenera z portami systemu bazowego. Często jest duże ograniczenie. Najczęściej komunikację z kontenerem ogranicza się do kilku portów i większe aplikacji zastępuje się kilkoma mniejszymi kontenerami obsługujących mniejsze moduły.
Inną wadą jest sposób trzymania danych [24]. W kontenerach nie powinno trzymać się danych. Zamiast tego dane powinny być zapisane na systemie bazowym, który współdzieli folder z danymi kontenera. Jest to spowodowane tym, że kontenery są zaprojektowane aby często je przenosić i kasować. Powstaje przez to problem, że dane nie są odizolowane i przy wrażliwych danych trzymanie ich na systemie bazowym nie jest zalecane.
Kontenery są lekkie, ponieważ posiadają jedynie bazę systemu operacyj-

nego [20]. Oznacza to, że system w kontenerze nie posiada wielu funkcjonalności, które odnajdziemy w standardowej wersji systemu. Jeśli następuje potrzeba modyfikacji uruchamianej aplikacji i skorzystania z funkcjonalności, która nie była dodana przy kompozycji kontenera, to należy dopisać dodatkowa konfigurację, która doda tę funkcjonalność.
Kontenery zostały zaprojektowane tak, żeby obsługiwać tylko jeden proces [20]. Oznacza to, że komponując w kontenerze dane środowisko nie można wykonywać kilku zadań na raz . Jeśli projekt wymaga wykorzystania kilku procesów w takim samym środowisku, to konieczne jest utworzenie nowego kontenera ze sklonowanym środowiskiem i obsługiwać projekt z kilku kontenerów. Zajmuje to więcej zasobów i może być niewygodne w utrzymaniu, ale najczęściej projekty o takiej specyfikacji są rzadko spotykane. Zazwyczaj w takich sytuacjach wykorzystuje się maszynę wirtualną.
Adres IP przypisany kontenerowi nie jest stały [25]. Przy każdym kolejnym uruchomieniu kontener otrzymuje nowy adres IP. Może to być uciążliwe przy oprogramowywaniu struktur sieciowych, które wymagają komunikacji po stałych adresach IP. Najczęściej polecane jest posługiwanie się nazwami kontenerów (każdy kontener musi mieć niepowtarzalną nazwę, które jeśli nie jest nadana przez człowieka, a jest generowana automatycznie) lub korzystanie z wystawiania portów przez system bazowy.

Jądro systemu bazowego odpowiada na izolację kontenera [26] i wszelkie podatności jądra automatycznie są też podatnościami kontenera. Jest to o tyle niebezpieczne, że potencjalny atak na aplikację działającą w kontenerze może dostać się do poziomu jądra systemu bazowego, gdzie ma dostęp do wyższego poziomu uprawnień kontroli działania systemu.
Kontener jest pojedynczym procesem w systemie bazowym, który zarządzany jest przez serwis demona (np. Docker). Generuje to dodatkowy potencjalny punkt ataku, gdzie wymagane jest zabezpieczenie nie tylko systemów wirtualnych, ale też serwisu demona kontrolującym je.
Konteneryzacja nie jest dobrym rozwiązaniem dla wrażliwych aplikacji lub danych [26]. Kontener jest odizolowany, ale jest łatwo osiągalny z poziomu systemu bazowego (istnieje wręcz narzędzie do zdalnego wiersza poleceń) i sam w sobie posiada kilka możliwości komunikacji z systemem bazowym. Również zalecaną praktyką w kontenerach jest korzystanie z folderów w systemie bazowym do trzymania danych. Taka praktyka w trzymaniu danych np. klientów banku jest niedopuszczalna i najczęściej dyskwalifikuje konteneryzację.

Maszyna wirtualna będzie dobrym rozwiązaniem dla projektów wymagających dużej izolacji pracy - maszyna jest odizolowana od fizycznego sprzętu,

a także od zewnętrznych systemów. Przy okazji maszyna wirtualna zapewnia sporą elastyczność skalowania wirtualnych zasobów, co jest na pewno odpowiednie dla projektów wymagających sporadycznych zmian w infrastrukturze.
Konteneryzacja będzie dobrym rozwiązaniem dla projektów wymagających dużej elastyczności i częstego skalowania. Kontener jest lekki i szybki do utworzenia, migracji, klonowania. Kontenery działają jako procesy, więc samoczynnie skalują potrzebne im zasoby do obsługi aplikacji. Konteneryzacja będzie też właściwym wyborem dla prostych aplikacji, które nie wymagają dużo konfiguracji. Umieszczenie takich aplikacji w kontenerze będzie szybkie do uruchomienia i kontroli. Również z tego samego względu konteneryzacja jest idealna dla projektów zakładających uruchomienia wielu takich samych aplikacji. Narzędzie typu Kubernetes pozwala zarządzać powielaniem i uruchamianiem takich samych aplikacji w wielu kontenerach i łączenie ich usług w sieci.




Rysunek 2.1: Porównanie architektury działania wirtualnych maszyn z konteneryzacją
**Źródło:** docker.com

# Rozdział 3

# Analiza wykorzystania platformy Kubernetes do zarządzania uczeniem maszynowym

Wraz z rozwojem technologii konteneryzacji pojawił się problem związany z zarządzaniem kontenerami na wielu maszynach. Pierwsze próby zautomatyzowania uruchamiania skonteneryzowanych aplikacji były realizowane za pomocą podstawowych komend systemowych wywołanych za pomocą struktur (ang. framework) zdalnego zarządzania. Przykładem takiej struktury może być Ansible [27]. Jest to struktura, która jest zorganizowana w książki zadań (ang. playbooks) zawierające zadania utworzone z modułów dostępnych w bazowej instalacji lub otwartoźródłowych dystrybuowanych np. przez platformę GitHub. Struktura Ansible zawiera również moduły dedykowane konteneryzacji opartej na platformie Docker. Istnieją moduły, które pozwalają uruchamiać kontenery, zbierać dane diagnostyczne, zarządzać stanem samego silnika platformy Docker. Dzięki temu możliwe jest proste uruchomienie pożądanych kontenerów na wielu systemach jednocześnie. Niestety wiąże się to z problemem. O ile struktura Ansible zapewnia wykonanie wszystkich zadań na systemach lub sygnalizuje błąd w przypadku niepowodzenia, to nie posiada mechanizmów pozwalających na monitorowanie systemu w czasie rzeczywistym.

Powstało więc narzędzie *docker-compose* [28], które jest narzędziem dedykowanym do platformy konteneryzacyjnej Docker. Narzędzie to pozwala zdefiniować pożądany stan kontenerów w systemie (również z ich opcjami dodatkowymi) w postaci pliku w formacie YAML, a następnie wywołać polecenie *docker-compose up . -d*. Zakłada się, że polecenie jest uruchomione z folderu zawierającego definicję pożądanego stanu o nazwie *docker-compose.yml*

(w przeciwnym wypadku wymagane są dodatkowe flagi konfiguracyjne). Następnym krokiem jest uruchomienie obserwatora systemowego, który realizuje zadeklarowany stan w platformie Docker, a następnie zbiera dane diagnostyczne i obserwuje stan rzeczywisty. Jeśli stan kontenerów zmieni się i będzie odbiegał od zadeklarowanego, to narzędzie *docker-compose* podejmie działania mające na celu ponowny powrót do oczekiwanego stanu. Narzędzie *docker-compose* sprawdza się doskonale w aplikacjach statycznych, czyli posiadających jeden niezmienny stan. Nawet po restarcie maszyny serwis *docker-compose* od razu aplikuje konfigurację, żeby utrzymać zadeklarowany stan.

Wraz z rozwojem technologii konteneryzacyjnej pojawiły się jednak nowe potrzeby, które wymagały dynamicznego zarządzania stanem kontenerów na wielu maszynach, a także ujednoliconego systemu zarządzania stosem sieciowym pomiędzy maszynami. Firma Google jako pierwsza podjęła próby utworzenia takiego systemu w ramach wewnętrznej firmowej platformy Borg. Platforma ta powstała z potrzeby pracowników, którzy tworzyli nowe aplikacje i często nie posiadali wiedzy technicznej pozwalającej skonfigurować serwery w sposób odpowiedni dla potrzeb testowania aplikacji. Platforma Borg zapewniała środowisko do uruchomienia testowych aplikacji na skonsolidowanych serwerach testowych dostępnych dla pracowników. Korzystając z zalet izolacji kontenerów od systemu bazowego rozwiązano również problem zachowania czystości serwerów udostępnionych do testowania, ponieważ pracownik nie instalował niczego na głównym systemie, a jedynie w kontenerze, który w prosty sposób mógł być usunięty. W roku 2014 firma Google bazując na platformie Borg stworzyła produkt komercyjny pod nazwą Kubernetes [29], która zapewniała podobną funkcjonalność. Platforma Kubernetes została udostępniona na zasadach otwartoźródłowych (OpenSource) i przekazana pod opiekę fundacji CNCF (ang. Cloud Native Computing Foundation). Od tego czasu platforma Kubernetes jest dynamicznie rozwijana przez społeczność (średnio co 2-3 miesiące wydawana jest nowa wersja platformy).

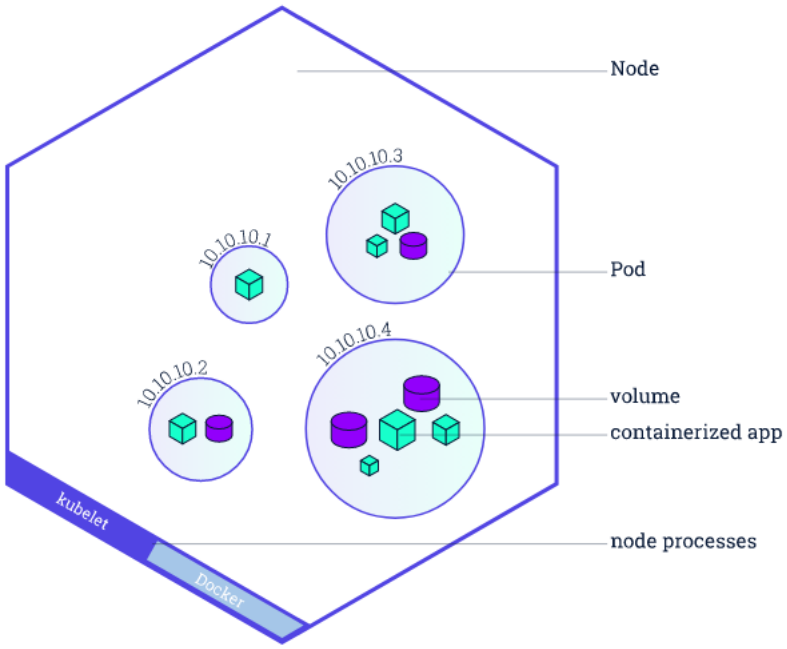
Platforma Kubernetes jest nazywana często orkiestratorem (ang. orchestrator) do platform konteneryzacji - najczęściej Docker. Pozwala ona organizować wiele maszyn w węzły (ang. nodes), które są ze sobą połączone logicznie za pomocą platformy. Węzły mogą pełnić funkcję *worker* oraz *master*. Węzeł o funkcji *worker* jest dedykowany do uruchamiania skonteneryzowanych aplikacji pracujących. Węzły o funkcji *master* z drugiej strony są dedykowane do zarządzania całym gronem węzłów (ang. cluster). Platforma Kubernetes jest oparta o bazę danych *etcd* [30] komunikując się z bazą oraz resztą węzłów zarządza całą platformą. Najczęściej pojawia się w gronie tylko jeden

węzeł o funkcji *master*, ale możliwe jest dostawienie większej liczby węzłów o takiej roli, żeby zapewnić wysoką dostępność (ang. High Availability). Standardowo węzły o roli *master* nie powinny uruchamiać kontenerów i domyślnie posiadają skazy (ang. taints), które blokują tworzenie kontenerów z poziomu platformy. Jednak czasami specjalistyczne aplikacje wymagają uruchomienia kontenerów na węzłach typu *master*. Wtedy możliwe jest nadanie konkretnym wdrożeniom tolerancji do uruchomienia ich na węzłach zarządzających.

Posiadając gotowe grono węzłów możliwe jest wykorzystanie API (Interfejs Programowania Aplikacji - ang. Aplication Programming Interface) [31] do uruchamiania kontenerów na węzłach oraz zarządzanie ich siecią. Zarządzanie kontenerami odbywa się poprzez definiowanie zasobów w platformie Kubernetes. Zazwyczaj są to definicje parametrów w postaci plików w standardzie YAML. Platforma Kubernetes posiada bogaty katalog zasobów pozwalających osiągnąć skomplikowane wdrożenia aplikacji. Od 2019 roku popularne stało się bardziej zaawansowane tworzenie aplikacji poprzez bezpośrednie tworzenie kodu pracującego z API platformy Kubernetes pod postacią Operator Framework (Struktura Operatorowa) [32]. Skomplikowane wdrożenia aplikacji mogą zostać zaprogramowane jako dodatkowy zasób w platformie Kubernetes (standardowo jest to kod napisany w języku Go, na bazie którego napisana jest cała platforma Kubernetes). Tak napisany operator może zostać uruchomiony w ramach platformy przez wdrożenie prostego pliku konfiguracyjnego w formacie YAML. Jest to możliwe, ponieważ wszystkie zasoby w platformie korzystają z zasobu nazywanego przez twórców Pod. Pod [33] jest najmniejszym rozpoznawalnym zasobem w platformie Kubernetes. Jest to Definicja jednego lub więcej kontenerów pracujących w ramach jednego węzła. Zasób Pod może posiadać również dodatkowe konfiguracje około kontenerowe, jak przykładowo dyski zewnętrzne zamontowane do kontenerów.


Rysunek 3.1: Schematy poglądowe idei zasobu Pod w platformie Kubernetes
**Źródło:** [https://kubernetes.io/docs/tutorials/kubernetes-basics]

W ramach jednego węzła może pracować wiele Pod'ów. Każdy z Pod'ów otrzymuje w ramach systemu platformy Kubernetes niepowtarzalny adres IP. Z poziomu węzła pracującego w trybie *master* widoczne są Pod'y, a nie poszczególne kontenery. Z punktu widzenia węzła obsługującego dane kontenery są one widoczne jako pojedyncze kontenery uruchomione w platformie Docker, ale ich konfiguracja jest zarządzana przez platformę Kubernetes. Niemożliwe jest zagnieżdżanie Pod'ów w ramach jednego grona węzłów. Czyli uruchamiania kontenerów w innych kontenerach. Istnieją natomiast kontenery inicjalizujące, które są uruchamiane przed startem właściwych kontenerów. Ich zadaniem jest wykonanie operacji przygotowawczych, które następnie zostaną przekazane do kontenerów zaprojektowanych już do pracy ciągłej.




Rysunek 3.2: Schemat poglądowy idei istnienia wielu Pod'ów w ramach węzła w platformie Kubernetes
**Źródło:** https://kubernetes.io/docs/tutorials/kubernetes-basics

Domyślnie jednak komunikacja pomiędzy Pod'ami jest utrudniona. Adresy IP są przyznawane losowo, a jeśli grono węzłów jest rozpięte na maszynach, które w swojej topologii sieciowej posiadają urządzenia typu router, to niemożliwe będzie uruchomienie Pod'u w węźle pracującym w sieci za routerem. W takiej sytuacji stosowane jest CNI (Interfejs Sieci Kontenerowej - ang. Container Network Interface), który zarządza komunikacją sieciową pomiędzy Pod'ami. CNI korzysta również z podstawowych zasobów platformy Kubernetes, co oznacza że w implementacji nie różni się od standardowych aplikacji i może częściowo być implementowane ręcznie. Możliwe jest też uzyskanie częściowej implementacji komunikacji CNI lub specyficznego zachowania sieciowego dzięki zasobowi Service. Zasób Service pozwala proste implementowanie rozwiązań domenowych dla Pod'ów oraz mechanizmów sieciowych, jak sterowanie obciążeniem pomiędzy wieloma kontenerami.

Platforma Kubernetes posiada również wiele innych zasobów, które pozwalają dynamicznie konfigurować wiele Pod'ów. Podstawowymi takimi zasobami są Set'y [34], czyli zbiory Pod'ów. Zasób ten pozwala uruchomić zdefiniowaną liczbę Podów gdziekolwiek w gronie węzłów. Platforma Kubernetes

zapewnia mechanizm, który automatycznie skieruje zapytania do odpowiednich węzłów, tak aby jak najlepiej zarządzać wykorzystaniem pamięci węzłów. Dodatkowo jeśli stan jakiekolwiek węzła się zmieni - czy to programowo, czy komunikacja z węzłem zaniknie na skutek np. wyłączenia maszyny, to wtedy platforma przeniesie, najszybciej jak to możliwe, Pod'y na inne dostępne węzły. Innym przydatnym zasobem jest DaemonSet [35], czyli zbiór Pod'ów, ale uruchamianych po jednym na każdym węźle w gronie (lub na wybranych węzłach za pomocą ograniczeń przydzielonych etykiet). Pozwala on na maksymalne wykorzystanie zasobów grona węzłów, a także uruchomienie aplikacji na każdym węźle i jest to równoznaczne z tym że aplikacja jest dostępna z każdego węzła.

Dzięki mechanizmowi automatycznego zarządzania wieloma węzłami, a także ciągłej obserwacji i analizy stanów na każdym z węzłów możliwe jest wykorzystanie platformy do wdrożenia aplikacji rozproszonego uczenia maszynowego. Uczenie maszynowe wymaga intensywnego wykorzystania zasobów obliczeniowych węzłów. Oznacza to, że mechanizm wykorzystujący jak najlepiej pamięć obliczeniową węzłów przez ustawianie kolejnych Pod'ów pozwoli osiągnąć maksimum wydajności. Dzięki centralnemu zarządzaniu gronem węzłów możliwe jest proste wdrożenie wielu kontenerów na węzłach za pomocą jednego pliku konfiguracyjnego. Również możliwości sieciowe dostarczane z platformą ułatwiają znacznie tworzenie architektury połączeń pomiędzy kontenerami znajdującymi się w Pod'ach. Analizując powyższe zalety platformy Kubernetes, wybrano ją do ostatecznej realizacji projektu.

# Rozdział 4

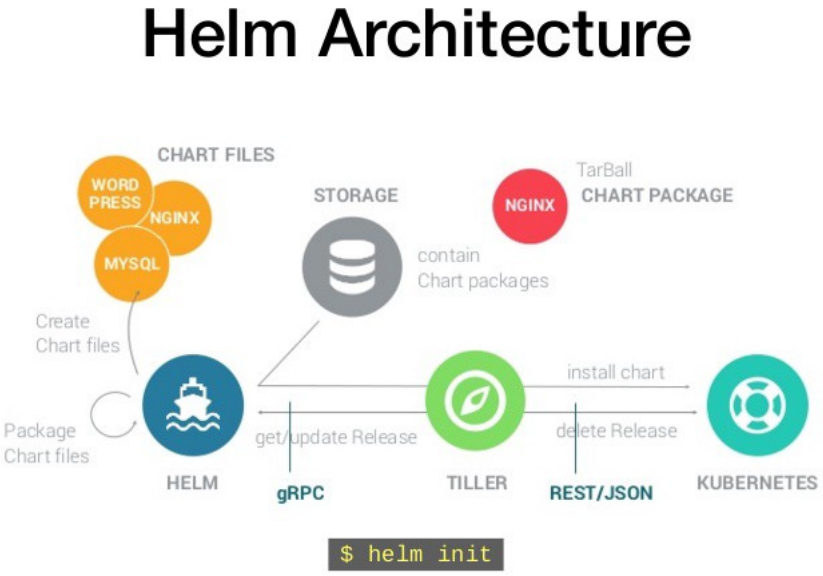
# Wykorzystanie narzędzia Helm do dystrybuowania zadań na wybrane węzły obliczeniowe

Wdrożenia gotowych aplikacji na platformie Kubernetes wymaga najczęściej utworzenia wielu zasobów z konkretną konfiguracją dostosowaną pod użytkownika. Poprawne wdrożenie wymagałoby wywołania wielu komend a także zmiany parametrów w wielu miejscach plików konfiguracyjnych. Wymagałoby to od operatora doskonałej znajomość architektury aplikacji oraz jej działania na samej platformie Kubernetes. W takiej sytuacji dystrybucja aplikacji byłaby utrudniona pod względem biznesowym ponieważ każdorazowo wymagałaby asysty eksperta który generuje dodatkowe koszty. Również posiadając już asystę eksperta należy wziąć pod uwagę ryzyko popełnienia ludzkiego błędu podczas edycji plików konfiguracyjnych. Zmiana jednego parametru może być równoznaczna ze zmianą wartości w np. 3 plikach konfiguracyjnych i operator może zapomnieć zmienić wartość w jednym z plików. Wtedy tracony jest dodatkowy czas potrzebny na zdiagnozowanie problemów z aplikacją oraz niebezpieczeństwo że aplikacja produkcyjna działa nieprawidłowo.

Z powyższych powodów powstał system obudowania gotowych wdrożeń platformy Kubernetes. Helm [36] bo tak nazywa się ten system pozwala na utworzenie tzw. Chart'u (po polsku dosłownie wykres) który zawiera szablony plików konfiguracyjnych które dynamicznie są wypełniane przy wdrożeniu aplikacji. Za pomocą jednej komendy *helm install* w terminalu uruchamiany jest mechanizm wypełniania plików konfiguracyjnych, a następnie uruchomienie w platformie Kubernetes wszystkich plików konfiguracyjnych zawartych w odpowiednim folderze szablonów. Dane potrzebne do wypełnie-

nia szablonów są predefiniowane w pliku *values.yml*. Jest to plik w formacie YAML posiadający pola klucz-wartość ze zmiennymi zdefiniowanymi w szablonach. Standardowo każdy Chart zawiera już wbudowany plik *values.yml* z wartościami domyślnymi, które zostaną wykorzystane jeśli nie został zdefiniowany inny plik z wartościami. Jednak jeśli użytkownik zdefiniuje swój własny plik i poda go do komendy za pomocą odpowiedniej flagi, to wtedy Chart wypełni szablony wartościami z tego pliku. Należy jedynie pamiętać, żeby plik zawierał wszystkie wymagane dane, aby poprawnie uruchomić aplikację. Dzięki temu użytkownik nie musi wiedzieć w jaki sposób zaprojektowane jest wdrożenie na platformie Kubernetes jego aplikacji, a ma możliwość w prosty sposób wdrażać spersonalizowane aplikacje.

Architektura systemu Helm [37] działa przy pomocy pośredniego serwera nazywanego Tiller. Serwer ten działa jako Pod w platformie Kubernetes. Jest to aplikacja, która komunikuje się z interfejsem platformy Kubernetes oraz przyjmuje polecenia od aplikacji klienckich dla systemu Helm (przykładowo narzędzia linii poleceń *helm*). Tiller posiada dostęp do kontrolera grona węzłów i może wysyłać zapytania HTTPS do np. utworzenia zasobów. Tiller służy także do uruchamiania zasobów platformy Kubernetes w odpowiedniej kolejności i do obsługi błędów. Aplikacje klienckie wysyłają polecenia do Tiller'a. Zazwyczaj są to proste komendy utworzenia lub usunięcia Chart'u, ale może też służyć do pobierania danych analitycznych. Jednak sam mechanizm wypełniania szablonów w odpowiedni sposób oraz pobieranie plików Chart'u jest wykonywany po stronie aplikacji klienckiej. Oznacza to, że chcąc zaimplementować własnego klienta korzystającego z komunikacji z Tiller'em należy zaimplementować własny system tworzenia plików konfiguracyjnych.



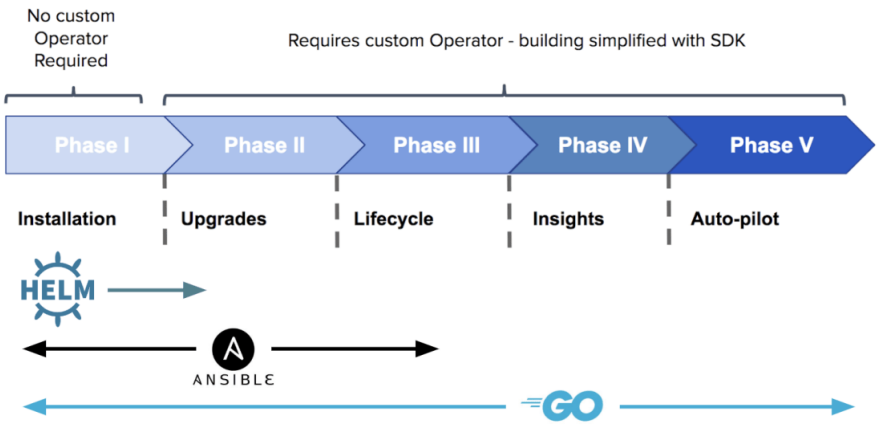

Rysunek 4.1: Schemat architektury działania systemu Helm
**Źródło:** https://blog.francium.tech

System Helm umożliwia również dystrybucję gotowych wdrożeń na system w własnym katalogu dostępnym na stronie www. Pierwotnie było to repozytorium na platformie GitHub, które zawierało zbiór gotowych wdrożeń. Aby zainstalować takie wdrożenie wystarczyło wtedy w komendzie *helm install* podać jedynie odpowiednią gałąź wdrożeń (np. *stable* dla wdrożeń stabilnych), a następnie nazwę. Potem wprowadzono jednak katalog z przystępną stroną internetową *hub.helm.sh*, gdzie dostępne są nie tylko Chart'y zawarte w repozytorium na GitHub'ie. Oczywiście system Helm zapewnia również wsparcie dla instalacji Chart'ów spoza oficjalnego katalogu. Możliwe jest zainstalowanie Chart'u z prywatnego repozytorium na praktycznie dowolnej platformie kontroli wersji (nie tylko platforma GitHub, ale też między innymi platforma GitLab, BitBucket itp.). W pracy nad utworzeniu własnego Chart'em przydatna jest również opcja instalacji Chart'u z plików lokalnych, co wiąże się z podaniem lokalnej ścieżki zamiast adresu repozytorium w komendzie.

System Helm jest doskonały jeśli celem jest jedynie wdrożenie aplikacji, która nie wymaga dalszych uaktualnień lub monitoringu. Jeśli aplikacja wymaga uaktualnień, które nie mogą być wykonane w samej aplikacji, a muszą zmieniać architekturę wdrożenia w platformie Kubernetes, to należy wyko-

rzystać narzędzie Ansible. Jest to narzędzie, które pozwala wykonywać zadania zdalnie na wielu maszynach. Dzięki temu można sekwencjonować listę komend systemowych, które muszą być wykonane, aby przeprowadzić uaktualnienie wdrożenia na platformie Kubernetes. Wdrożenie może być nadal zrealizowane w formie Chart'u na systemie Helm, albo może być po prostu wywołaniem serii komend bezpośrednio odwołujących się do platformy Kubernetes.

Jeśli wymogiem wdrożenia byłoby również zbieranie danych analitycznych i automatyczne reagowanie na zmiany środowiska, to wtedy wymagane jest zastosowanie bardziej skomplikowanych narzędzi. Od 2019 roku coraz popularniejsze staje się tworzenie tzw. Operatorów pisanych w języku Go (tym samym, w którym napisany jest mechanizm platformy Kubernetes). Operator [32] niczym innym, jak dostosowany do konkretnej aplikacji dedykowany zasób platformy Kubernetes. Wszystkie zasoby w platformie są pisane w podobny sposób w języku Go, więc znając standardową strukturę i sposoby odwoływania się do interfejsu platformy można tworzyć w pełni zautomatyzowane wdrożenia. Jednak tworzenie operatora ma sens tylko w przypadku bardzo rozbudowanych aplikacji i z dedykowanymi mechanizmami analitycznymi. W przypadku niniejszego projektu możliwe byłoby utworzenie operatora, ale nie jest to konieczne i ten sam efekt można osiągnąć prostszymi metodami, jak wykorzystanie systemu Helm, który będzie całkowicie wystarczający.




Rysunek 4.2: Porównanie możliwości wdrożeń za pomocą systemu Helm, narzędzia Ansible oraz Operatork SDK napisanego w języku Go
**Źródło:** https://blog.openshift.com/make-a-kubernetes-operator-in-15-minutes-with-helm/

# Rozdział 5

# Architektura badanego rozwiązania

## 5.1 Wybór biblioteki do uczenia maszynowego

Na rynku istnieje wiele bibliotek umożliwiających w prosty sposób pisanie oprogramowania do uczenia maszynowego. Jednak tylko kilka z nich posiada gotowe metody zrównoleglenia procesu uczenia oraz dystrybucji procesu na wiele węzłów. Przy wyborze należy pamiętać o dostępności danej biblioteki, obecnie panujących standardach na rynku oraz uniwersalności oprogramowania. Po analizie dostępnych rozwiązań, wybrano 3 potencjalne biblioteki do wyboru.

Pierwszą z nich jest biblioteka Gluon [38], która zapewnia wysokopoziomowe budowanie architektury modelu uczenia maszynowego oraz posiada rozbudowany system zrównoleglania procesu uczenia. Rozwiązanie dystrybucji uczenia w tej bibliotece jest zrealizowane bardzo prosto i intuicyjnie, co mogłoby obiecująco zapowiadać wykorzystanie właśnie tej biblioteki w architekturze projektu. Niestety Gluon jest zaprojektowany do wydajnego uczenia prostych modeli i przewidziany raczej do prototypowania rozwiązania, które potem docelowo jest implementowane na bardziej zaawansowanej bibliotece. Z tego powodu biblioteka ta nie została wykorzystana w projekcie, ponieważ nie mogłaby być zastosowana produkcyjnie do uczenia skomplikowanych modeli już do konkretnych aplikacji.

Kolejną biblioteką jest PyTorch [39]. Jest to biblioteka rozwijana głównie przez firmę Facebook. Pozwala ona na budowanie skomplikowanych modeli uczenia maszynowego, a także ich dynamiczną edycję nawet podczas procesu

uczenia. PyTorch posiada wbudowane mechanizmy zrównoleglania i dystrybuowania procesu uczczenia. Dodatkowo PyTorch jest jednym z głównych standardów przemysłowych uczenia maszynowego, który jest wykorzystywany często przez projekty biznesowe, jak i naukowe. Z tego powodu wykorzystanie tej biblioteki w projekcie pozwalałoby na uruchomienie wielu już istniejących modeli dostępnych np. na platformie GitHub.

Ostatnią biblioteką jest Tensorflow [40], który w chwili pisania tej pracy jest najpopularniejszą biblioteką uczenia maszynowego. Tensorflow jest rozwijany przez firmę Google, czyli porównywalnie dużego gracza na rynku, co Facebook. Tensorflow pozwala budować skomplikowane modele oraz posiada pochodną bibliotekę Keras [41], która pozwala budować modele wysokopoziomowe Tensorflow - podobnie, jak Gluon. Biblioteka posiada również mechanizmy zrównoleglania i dystrybucji procesu nauczania, ale również jest wzbogacona o dodatkowe narzędzia ułatwiające wykorzystanie tego mechanizmu. Firma Google co chwilę prezentuje nowe narzędzia do wykorzystania biblioteki Tensorflow. Ponieważ firma Google posiada również platformę chmurową GCP [42], a także rozwija platformę Kubernetes, to udostępnia gotowe kontenery i rozwiązania pozwalające w prosty sposób uruchomić skrypt stworzony za pomocą biblioteki Tensorflow. Google Cloud Platform pozwala wręcz utworzyć klaster Kubernetes i z interfejsu użytkownika wybrać gotowe wdrożenia uczenia maszynowego. Niestety w chwili pisania tej pracy firma Google udostępnia jedynie gotowe wdrożenia wykorzystania gotowych modeli, a niniejsza praca skupia się na uczeniu tych modeli. Jednak dzięki zaimplementowanym mechanizmom możliwe jest dostosowanie gotowych rozwiazań na potrzeby celu projektu jako najbardziej przydatną do realizacji celów jakie postawił sobie autor tej pracy. Z powyższych powodów do realizacji projektu wybrano bibliotekę Tensorflow.

## 5.2 Realizacja środowiska do uruchomienia biblioteki uczenia maszynowego

W celu uruchomienia skryptów wykorzystujących bibliotekę potrzebne jest środowisko. Na szczęście firma Google udostępnia gotowy kontener z zainstalowanymi wszystkimi zależnościami. Kontener ten bazuje na systemie Ubuntu 18.04, który standardowo posiada w sobie system DKMS (ang. Dynamic Kernel Module Support). System ten pozwala na dynamiczne wsparcie modułów jądra systemu. Oznacza to, że niezależnie od tego na jakiej wersji jądra systemu bazowana jest maszyna gospodarza (co oznacza, że kontener

będzie miał taką samą wersję jądra systemu), to moduły systemowe zadziałają poprawnie. Jest to ważne w sytuacji, gdy klaster będzie zbudowany na maszynach z różnymi systemami. Przykładowo system RedHat Enterprise Linux 7.x aktualnie bazuje na jądrze systemu w wersji 3.10 i jest szeroko wykorzystywany w przemyśle. Z kolei nowsza wersja systemu RHEL w wersji 8.x, która jest naturalnym następcą wersji 7.x bazuje już na jądrze w wersji 4.18. Inny system szeroko wykorzystywany w przemyśle to Ubuntu, który w swojej najnowszej wersji LTS (Long Time Support) wykorzystuje jądro systemu w wersji 5.3. Jak widać więc, nie można jednoznacznie określić wersji jądra systemu dla której aplikacja będzie uruchamiana. Wykorzystanie systemu z DKMS pozwala ujednolicić aplikację dla różnych systemów.
Kontener dystrybuowany za pomocą platformy Docker Hub posiada zainstalowane wszelkie potrzebne narzędzia oraz skonfigurowaną bibliotekę pod język programowania Python. Dostępne są kontenery wspierające zarówno wersję 2, ja i 3 języka Python. Pomimo tego, że od końca 2019 roku oficjalnie zakończono wsparcie dla języka Python w wersji 2, to nadal wiele przemysłowych aplikacji korzysta z właśnie tej wersji, dlatego wsparcie tej wersji jest również ważne. Biblioteka Tensorflow w swojej strukturze jest identyczna dla obu wersji języka Python (oczywiście różni się implementacja samej biblioteki, ale dla użytkownika biblioteki składnia jest ta sama).

Przy pomocy narzędzia do budowania własnych kontenerów bazujących na innych kontenerach możliwe jest proste zbudowanie kolejnego własnego kontenera. Kontener ten posiada wszystkie funkcjonalności obrazu dystrybuowanego przez firmę Google oraz wszystkie elementy specyficzne dla projektu - w tym zainstalowane dodatkowe pakiety systemowe oraz skopiowane skrypty.
Własny kontener można utworzyć za pomocą wiersza poleceń przez uruchomienie obrazu bazowego kontenera z odpowiednimi argumentami - narzędzie Docker posiada wbudowane funkcje do uruchamiania kontenerów z dołączonymi plikami systemu gospodarza, przekierowanie portów i wielu innymi. Następnie użytkownik stosując polecenie *docker exec <nazwa kontenera> sh* może ręcznie skonfigurować z wiersza poleceń kontenera docelowy system - przykładowo zainstalować potrzebne pakiety lub skonfigurować pliki potrzebne do poprawnego działania kontenera. Po skonfigurowaniu wszystkich parametrów kontenera należy oznaczyć obraz odpowiednim tag'iem. W systemie Docker każdy kontener posiada swój numer ID, który pozwala zidentyfikować dany obraz. Na warstwie powyżej numerów ID funkcjonują tagi. Tag jest wskaźnikiem na konkretny obraz. Zazwyczaj tag składa się z trzech elementów: *<repozytorium>/<nazwa>:<wersja>*. Repozytorium wskazuje na domenę, która przetrzymuje zapisane obrazy kontenerów możliwe do pobra-

nia. Może to być domena publiczna lub ograniczona hasłem (w przypadku zabezpieczenia domeny hasłem wymagane jest uprzednie zalogowanie lokalne do repozytorium przez komendę *docker login*). Jeśli nie zostanie podane repozytorium, to automatycznie przypisane jest repozytorium publiczne na platformie Docker Hub. Nazwa obrazu kontenera może być dowolna i często zawiera również wskaźnik na użytkownika, który jest autorem danego obrazu (Wtedy nazwa składa się z *<nazwa użytkownika>/<nazwa obrazu>*). Wersje obrazów mogą mieć dowolną formę. Mogą to być ciągi znaków, liczby lub często spotykane kilka liczb oddzielonych kropkami oznaczających główną wersję obrazu oraz podwersję. Standardowo najnowsze wersje kontenerów są oznaczane jako *latest*. Dlatego często konkretne ID kontenera jest oznaczone przez kolejną wersję oraz przez wersję latest, aby wskazać, że jest to wersja najnowsza. Oznaczony kontener należy wysłać do repozytorium przez komendę *docker push*. Jeśli użytkownik ma dostęp do repozytorium wespecyfikowanego w tag'u, to kontener zostanie wysłany na serwer. Przy tworzeniu kolejnych wersji tego samego kontenera wygodny jest mechanizm, który wykrywa istniejący już kontener na serwerze zewnętrznym i przesyła tylko nowe warstwy różniące się od poprzednika.

Narzędzie Docker umożliwia takie zautomatyzowane budowanie kontenerów przez komendę *docker build*, która wykorzystuje Dockerfile. Dockerfile jest to plik sformatowany według kolejnych instrukcji dla kontenera. Typowa składnia zawiera format *<Komenda> <Argumenty>*. Między innymi możliwe komendy to:

- FROM - komenda służąca do zdefiniowania obrazu bazowego kontenera. Notacja identyczna, jak w sytuacji tag'u.
- RUN - komenda służąca do wykonania podczas budowania kontenera komendy linii poleceń - może to być np. pobranie programu przez pakiet apt, czy yum.
- ADD - komenda służąca do dodania do systemu plików kontenera plików z systemu, na którym budowany jest kontener. Często wykorzystywana do kopiowania skryptów do wnętrza kontenera.
- CMD - komenda do określenia komendy wykonanej przy starcie kontenera po wyjściu z stanu entrypoint. Standardowo kontener powinien działać z procesem, który się nie kończy (nie zwraca kodu wyjściowego komendy). W sytuacji, gdy proces zostaje zakończony, kontener przestaje działać.
- ENTRYPOINT - komenda służąca do określenia komendy wykonywanej przy uruchomieniu kontenera przed wykonaniem komend określonej

w CMD. Powinna to być komenda, który zwraca kod wyjścia i służy do przygotowania uruchomionego kontenera do pracy.

Do projektu wykorzystano poniższy Dockerfile:

Listing 5.1: Plik Dockerfile do budowy kontenera wykorzystanego w pracy

```
FROM tensorflow/tensorflow:1.6.0-devel

ADD . /root

WORKDIR /root
ENTRYPOINT ["python", "train_distributed.py"]
```

Linia *FROM tensorflow/tensorflow:1.6.0-devel* oznacza, że obrazem bazowym będzie obraz z pakietem Tensorflow dostarczony przez firmę Google w wersji 1.6.0-devel. Linia *ADD . /root* oznacza, że wszystkie skrypty zawarte w folderze, gdzie umieszczony jest Docerfile zostaną przekopiowane pod ścieżką /root zachowując hierarchię plików. Linia *WORKDIR /root* oznacza, że proces uruchomiony przez kontener będzie traktował folder /root jako folder, z którego uruchamiane są polecenia. Linia *ENTRYPOINT ["python", "train_distributed.py"]* specyfikuje, że po uruchomieniu kontenera należy uruchomić skrypt train_distributed.py, który zawiera już w sobie dalszą logikę działania.

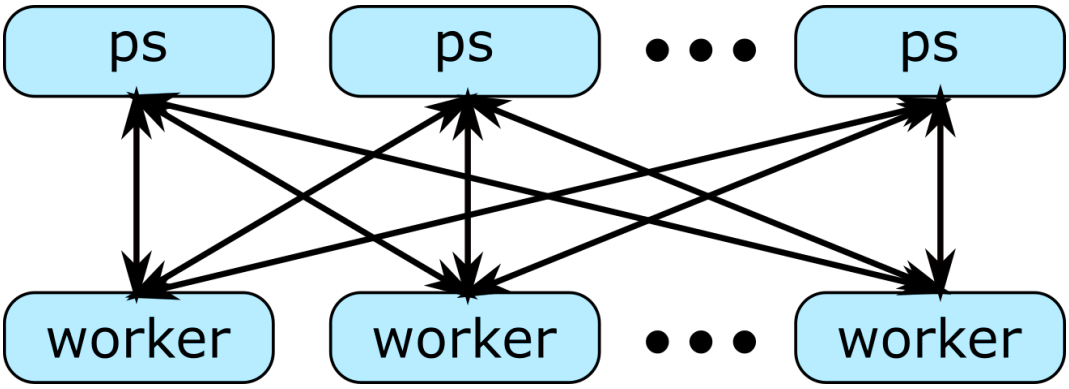
Tak przygotowany obraz kontenera został umieszczone na platformie Docker Hub, skąd był pobierany w dalszej części projektu.

## 5.3 Realizacja skryptów uczenia maszynowego

Logika procesu uczenia maszynowego została zawarta w skrypcie napisanym w języku Python w wersji 2. Skrypt jest przystosowany do działania w dwóch trybach. Trybie *ps* oraz trybie *worker*. Są to dwa tryby wprowadzone do biblioteki tensorflow razem z funkcjonalnością Distributed Tensorflow.

Tryb *ps* jest rozwijany do nazwy *Parameter Server* i oznacza to typ serwera, która zawiera w sobie zmienne modelu sieci neuronowej (np. wagi kolejnych węzłów), które są aktualizowane podczas procesu uczenia. Dzięki temu zmienne modelu są umieszczone w jednym miejscu i jednostki przeprowadzające sam proces uczenia nie muszą posiadać zduplikowanych danych lokalnie.
Tryb *worker* oznacza typ jednostki, która wykonuje obliczenia i zwraca uaktualnione wartości zmiennych do jednostki w trybie *ps*. *Worker* dostaje od *ps* wartości zmiennych (np. wag) startowych. Dane uczące mogą być umiesz-

czone lokalnie na jednostkach w trybie *worker* lub dostępne z oddzielnego serwera bazy danych. W przypadku lokalnego przechowywania danych jest to mało efektywne, ponieważ kopia danych musi być wielokrotnie kopiowana. W przypadku, gdyby dane były regularnie uaktualniane, jak może to mieć miejsce w środowisku produkcyjnym, gdzie chcemy dynamicznie dostosowywać model do aktualnie panujących warunków - wielokrotnie wymagane byłoby duże obciążenie sieci przesyłając wielokrotnie te same dane. Gdyby w tym samym czasie trwał również proces uczenia, to możliwe byłoby zanotowanie drastycznego spadku szybkości procesu.




Rysunek 5.1: Schemat architektury rozwiązania do rozproszonego uczenia maszynowego
**Źródło:** Opracowanie własne

W przypadku niniejszego projektu wykorzystane zostaną dane z biblioteki MNIST, która dostępna jest już w samym pakiecie Tensorflow. Tym samym każdy kontener z zainstalowanym pakietem Tensorflow posiada te dane i wystarczy zaledwie odpowiednia funkcja importująca w skrypcie. Biblioteka MNIST została wybrana jako przykładowy problem uczenia maszynowego, który jest szeroko stosowany w większości przykładach poruszających rozwiązywanie problemów rzeczywistych. Biblioteka MNIST zawiera bazę cyfr zapisanych odręcznie w różnych stylach. Problemem do rozwiązania jest model sieci neuronowej, który analizując odręcznie zapisane kształty zwróci odpowiadające im cyfry.

Rysunek 5.2: Przykładowe dane zawarte w bibliotece MNIST
**Źródło:** yann.lecun.com/exdb/mnist/

Ponieważ dane uczące nie będą uaktualniane (dane w bibliotece są statyczne i wystarczające na potrzeby projektu) i są zawarte w już zainstalowanej bibliotece, to wykorzystanie danych zablokowanych lokalnie nie stanowi większego problemu i będzie odpowiednie na potrzeby projektu.

Po aktualizacji wag przez jednostkę *worker* uaktualnione wagi są zwracane do jednostki w trybie *ps*, gdzie ostateczne dane są uaktualniane analizując wyniki otrzymane z wielu różnych jednostek *worker*. Możliwe jest multiplikowanie jednostek w trybie *ps* i jest to rozwiązanie często wykorzystywane ze względu na zmniejszenie obciążenia sieci między jednostkami. Przykładowo biorąc pod uwagę zadanie uczenia maszynowego, gdzie zdefiniowane jest 250 milionów zmiennych analiza danych na jednostkach typu *worker* wynosi 1 sekundę. W pierwszej konfiguracji, gdzie zdefiniowana jest tylko jedna jednostka typu *ps* i 10 jednostek typu *worker*, każda jednostka w trybie *worker* musi przesyłać co sekundę dane o wielkości 1GB. Przy istnieniu jedynie jednej jednostki w trybie *ps* potrzebna jest sieć potrafiąca obsłużyć przepustowość łącza 80Gbps. Dodając do konfiguracji kolejną jednostkę działającą w trybie *ps* pozwalamy podzielić obsługiwane zmienne na dwa serwery, a co za tym idzie ograniczyć potrzebne dane do przesłania na linii *ps-worker*. Dzięki temu potrzebna szerokość sieci zmniejsza się o połowę do 40Gbps. Oczywiście przy dzisiejszych standardach nie ma większego problemu ze znalezieniem łącza, które zapewni taką przepustowość danych. Jednak coraz częściej infrastruk-

tury sieciowe budowane są w serwisach chmurowych, gdzie cena zależy od wykorzystanych zasobów chmurowych, a opłata jest pobierana za ilość przesłanych danych w zależności od obciążenia sieci. Oznacza to, że ta sama ilość danych przesłana łączem o większej przepustowości będzie droższa od danych przesłanymi dwoma łączami o mniejszej przepustowości.

Struktura skryptu składa się z wstępnej konfiguracji na podstawie flag zdefiniowanych przez użytkownika przy wywołaniu skryptu, a następnie jednej z dwóch roli w zależności od trybu. Możliwe opcje do przekazania jako argumenty to:

- **data_dir** - folder do przetrzymania danych wykorzystanych w procesie uczenia
- **log_dir** - folder do przetrzymywania logów z pracy skryptu.
- **num_gpus** - liczba procesorów GPU na maszynie wykorzystanej do uczenia (nie wykorzystywane w tych badaniach, ale możliwe przy wykorzystaniu z np. specjalistycznymi komputerami posiadających kilka kart graficznych)
- **hidden_units** - liczba warstw ukrytych w modelu sieci
- **train_steps** - liczba kroków do wykonania w procesie uczenia
- **batch_size** - Rozmiar pakietu danych analizowanych w pojedynczym kroku uczenia
- **learning_rate** - wskaźnik uczenia
- **ps_hosts** - Lista serwerów w trybie *ps* [w formacie *hostname:port*]. Jeśli flaga nie zostanie podana, to skrypt weźmie listę z zmiennej środowiskowej *PS_HOSTS*
- **worker_hosts** - Lista serwerów w trybie *worker* [w formacie *hostname:port*]. Jeśli flaga nie zostanie podana, to skrypt weźmie listę z zmiennej środowiskowej *WORKER_HOSTS*
- **job_name** - pełniona rola (*worker*|*ps*). Jeśli flaga nie jest zdefioniowana, to rola będzie ustalona na podstawie zmiennej środowiskowej *JOB_NAME*
- **existing_servers** - Flaga definiująca, czy utworzyć nowy serwer do uczenia maszynowego, czy podłączyć się też do już istniejącego.

Następnie wszystkie argumenty są analizowane i dostosowywane do biblioteki Tensorflow. Ważnym krokiem jest definicja obiektu serwera, gdzie łączone są ze sobą jednostki pracujące w trybach *ps* oraz *worker*. Najpierw

definiowany jest obiekt *cluster*, który zawiera w sobie dane dotyczące wszystkich jednostek. Następnie jeśli nie jest stworzony już serwer, do którego dana jednostka może się dołączyć, to tworzony jest nowy serwer korzystający z infrastruktury grupy zdefiniowanej w obiekcie *cluster*. Na koniec sprawdzany jest warunek, czy jednostka uruchamiająca skrypt powinna pracować w trybie *ps* i jeśli tak, to jest dołączana do serwera.

Listing 5.2: Definicja serwera do uczenia modelu

```
ps_spec = FLAGS.ps_hosts.split(",")
worker_spec = FLAGS.worker_hosts.split(",")

cluster = tf.train.ClusterSpec({"ps": ps_spec,
                                "worker": worker_spec})

if not FLAGS.existing_servers:
    # Not using existing servers. Create an in-process server.
    server = tf.train.Server(cluster,
                             job_name=FLAGS.job_name,
                             task_index=FLAGS.task_index)
if FLAGS.job_name == "ps":
    server.join()
```

W kolejnym kroku definiowane są elementy do wyświetlenia w narzędziu Tensorfboard. Jest to narzędzie pozwalające wyświetlić wyniki procesu uczenia w estetycznym interfejsie użytkownika. Zdefiniowane jest zbieranie jak największej ilości danych, które mogą być przydatne w analizie gotowego modelu.

Listing 5.3: Uruchomienie narzędzie zbierania danych do modułu Tensorboard

```
def variable_summaries(var):
    """Attach a lot of summaries to a Tensor
    (for TensorBoard visualization)."""
    with tf.name_scope('summaries'):
        mean = tf.reduce_mean(var)
        tf.summary.scalar('mean', mean)
        with tf.name_scope('stddev'):
            stddev = tf.sqrt(tf.reduce_mean(tf.square(var-mean)))
        tf.summary.scalar('stddev', stddev)
        tf.summary.scalar('max', tf.reduce_max(var))
        tf.summary.scalar('min', tf.reduce_min(var))
        tf.summary.histogram('histogram', var)
```

Następną operacją jest rozmieszczanie zmiennych na jednostkach pracujących w trybie *ps*. Podczas inicjalizacji modelu zmienne zostaną samoistnie rozmieszczone w odpowiedni sposób według logiki zawartej w bibliotece Tensorflow. Wywołanie logiki rozmieszczenia zmiennych jest realizowane za po-

mocą funkcji kontekstowej języka Python *with*, gdzie w zdefiniowanym kontekście inicjalizowany jest model

Listing 5.4: Definicja logiki dystrybuującej zmienne na jednostkach typu *ps*

```
with tf.device(
    tf.train.replica_device_setter(worker_device=worker_device,
                                   ps_device="/job:ps/cpu:0",
                                   cluster=cluster)):
    ...
```

W zdefiniowanym kontekście następnie inicjalizowany jest model i jego parametry uczenia. Parametry modelu są ustalane na podstawie argumentów przekazanych w wierszu poleceń opisanych wcześniej. Rozmiar wejścia sieci neuronowej odpowiada rozmiarowi zdjęć przekazywanych z danych uczących biblioteki MNIST. Na wyjściu sieci ustawiona jest warstwa Softmax.

Listing 5.5: Definicja modelu sieci neuronowej

```
# Input placeholders
with tf.name_scope('input'):
    x = tf.placeholder(tf.float32,
                       [None, IMAGE_PIXELS*IMAGE_PIXELS],
                       name='x-input')
    y_ = tf.placeholder(tf.float32, [None, 10], name='y-input')

# Reshape input image for summary show
with tf.name_scope('input_reshape'):
    image_shaped_input = tf.reshape(x, [-1, 28, 28, 1])
    tf.summary.image('input', image_shaped_input, 10)

# hidden layer - layer_linear
with tf.name_scope('layer_linear'):
    with tf.name_scope('weights'):
        # Variables of the hidden layer
        hid_w = tf.Variable(tf.truncated_normal(
            [IMAGE_PIXELS * IMAGE_PIXELS, FLAGS.hidden_units],
            stddev=1.0 / IMAGE_PIXELS),
                            name="hid_w")
        variable_summaries(hid_w)
    with tf.name_scope('bias'):
        hid_b = tf.Variable(tf.zeros([FLAGS.hidden_units]),
                            name="hid_b")
        variable_summaries(hid_b)
    with tf.name_scope('Wx_plus_b'):
        hid_lin = tf.nn.xw_plus_b(x, hid_w, hid_b)
        tf.summary.histogram('pre_activations', hid_lin)
    # activation - Relu
    hid = tf.nn.relu(hid_lin)
    tf.summary.histogram('activations', hid)

# softmax layer
with tf.name_scope('softmax_layer'):
    # Variables of the softmax layer
    sm_w = tf.Variable(tf.truncated_normal(
        [FLAGS.hidden_units, 10],
        stddev=1.0 / math.sqrt(FLAGS.hidden_units)),
                       name="sm_w")
    sm_b = tf.Variable(tf.zeros([10]), name="sm_b")
    y = tf.nn.softmax(tf.nn.xw_plus_b(hid, sm_w, sm_b), name="y")
```

Po definicji modelu, definiowane są parametry optymalizacji oraz funkcji dokładności modelu.

Listing 5.6: Definicja optymalizacji i funkcji dokładności modelu

```
# lose function - cross_entropy
with tf.name_scope('cross_entropy'):
    cross_entropy = -tf.reduce_sum(
        y_ * tf.log(tf.clip_by_value(y, 1e-10, 1.0)))
tf.summary.scalar('cross_entropy', cross_entropy)

# Optimization
with tf.name_scope('train'):
    opt = tf.train.AdamOptimizer(FLAGS.learning_rate)

    if FLAGS.sync_replicas:
        if FLAGS.replicas_to_aggregate is None:
            replicas_to_aggregate = num_workers
        else:
            replicas_to_aggregate = FLAGS.replicas_to_aggregate

        opt = tf.train.SyncReplicasOptimizer(
            opt,
            replicas_to_aggregate=replicas_to_aggregate,
            total_num_replicas=num_workers,
            name="mnist_sync_replicas")

    train_step = opt.minimize(cross_entropy,
                              global_step=global_step)

# Accuracy
with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
        correct_prediction = tf.equal(tf.argmax(y, 1),
                                      tf.argmax(y_, 1))
    with tf.name_scope('accuracy'):
        accuracy = tf.reduce_mean(
            tf.cast(correct_prediction, tf.float32))
tf.summary.scalar('accuracy', accuracy)
```

Na koniec definiowana jest sesja uczenia modelu. Jedna z jednostek pracujących w trybie *worker* wybierana jest jako jednostka wiodąca, która będzie inicjalizować proces uczenia. Pozostałe jednostki będą na nią czekać, żeby zacząć przetwarzać dane.

Listing 5.7: Inicjalizacja sesji uczenia

```
sess_config = tf.ConfigProto(
  allow_soft_placement=True,
  log_device_placement=FLAGS.log_device_placement,
  device_filters=[
      "/job:ps", "/job:worker/task:%d" % FLAGS.task_index
  ])

# The chief worker (task_index==0) session will
# prepare the session,
# while the remaining workers will wait for
# the preparation to complete.
if is_chief:
    logger.info("Worker %d: Initializing session..." %
                FLAGS.task_index)
else:
    logger.info("Worker %d: Waiting for "
                "session to be initialized..." %
                FLAGS.task_index)

if FLAGS.existing_servers:
    server_grpc_url = "grpc://" + worker_spec[FLAGS.task_index]
    logger.info("Using existing server at: %s" % server_grpc_url)

    sess = sv.prepare_or_wait_for_session(server_grpc_url,
                                          config=sess_config)
else:
    sess = sv.prepare_or_wait_for_session(server.target,
                                          config=sess_config)

logger.info("Worker %d: Session initialization complete." %
            FLAGS.task_index)

if FLAGS.sync_replicas and is_chief:
    # Chief worker will start the chief queue
    # runner and call the init op.
    sess.run(sync_init_op)
    sv.start_queue_runners(sess, [chief_queue_runner])
```

Następnie w pętli *while True*, czyli nieskończonej pętli, wykonywany jest proces uczenia. Jeśli liczba kroków uczenia jest zdefiniowana (czyli nie wynosi 0), to po wykonaniu odpowiedniej liczy kroków uczenia pętla zostanie przerwana. Co 10 kroków uczenia sieci testuje obecny stan zmiennych na danych testowych, w pozostałych krokach sieć uczy się na danych uczących.

Listing 5.8: Kod obsługujący proces uczenia

```
while True:
  # Test-set accuracy and record summary
  if local_step % 10 == 0:
      test_feed = {x: mnist.test.images, y_: mnist.test.labels}
      acc, summary = sess.run([accuracy, summary_op],
                              feed_dict=test_feed)
      test_writer.add_summary(summary, local_step)
      local_step += 1
      logger.info("Accuracy at local step %s: %s" %
                  (local_step, acc))
  else:
      # Training feed
      batch_xs,batch_ys=mnist.train.next_batch(FLAGS.batch_size)
      train_feed = {x: batch_xs, y_: batch_ys}

      _, summary, step = sess.run(
          [train_step, summary_op, global_step],
          feed_dict=train_feed)
      local_step += 1
      train_writer.add_summary(summary, step)

      now = time.time()
      logger.info(
          "%f: Worker %d: training step "
          "%d done (global step: %d)" %
          (now, FLAGS.task_index, local_step, step))

  if step >= FLAGS.train_steps and FLAGS.train_steps > 0:
      break
```

Po zakończeniu procesu uczenia możliwe jest wyeksportowanie gotowego modelu w wybrane miejsce na dysku. W praktyce zazwyczaj jest to globalny dysk zdefiniowanych dla wszystkich kontenerów, gdzie składowane są wszystkie dane z procesu uczenia. W ten sposób analiza logowanych danych w każdym kroku uczenia jest wygodna, a utworzony model szeroko dostępny. Jeśli globalny dysk nie jest zdefiniowany, to wtedy dane są rozrzucone po maszynach gospodarzach, które uruchamiały kontenery w ramach grupy infrastruktury uczącej. Jest to wtedy skomplikowane dla operatora szukającego konkretnych danych, ponieważ jest on zmuszony szukać danych w wielu miejscach.

## 5.4 Realizacja infrastruktury uczącej

Mając gotowy kontener, który zawiera uniwersalny skrypt możliwy do uruchomienia w każdym środowisku należy zaimplementować jego wykorzystanie w gronie (ang. cluster) jednostek obliczeniowych. Docelową platformą jest platforma Kubernetes, która jest obecnie standardem przemysłowym uruchamiania aplikacji w rozproszonych systemach. Dodatkowo implementacja rozwiązania działającego z platformą Kubernetes pozwala od razu na możliwość uruchomienia rozwiązania na platformie RedHat Openshift - popularnym produkcie na rynku. Jest to możliwe, ponieważ RedHat architektura Openshift korzysta z systemu kubernetes. Znaczącymi różnicami są jedynie bardziej rygorystyczne zasady bezpieczeństwa, które jednak nie są problemem przy aplikacji napisanej zgodnie z zachowaniem dobrych praktyk programistycznych. Inną znaczącą różnicą jest wykorzystanie do obsługi kontenerów programu Podman zamiast programu Docker. Możliwe jest jednak uruchamianie już istniejących kontenerów zbudowanych w programie Docker w programie Podman.

Platforma Kubernetes skupia się wokół Pod'ów - najmniejszej jednostki możliwej do uruchomienia, która zawiera w sobie jeden lub grupę działających kontenerów. Ponieważ cała aplikacja uczenia maszynowego zawiera się w jednym kontenerze, to docelowy Pod będzie prostą definicją jednego kontenera.
Posługując się jednak samymi Pod'ami wymagane byłoby zaimplementowanie skomplikowanej logiki, która w zależności od liczby zadanych jednostek obliczeniowych w trybie *ps* i trybie *worker* uruchamia kontenery na wybranych węzłach i dodatkowo dba o równe rozłożenie zasobów, żeby uzyskać jak najlepszą sprawność aplikacji. Na szczęście platforma Kubernetes posiada już wbudowane rozwiązania pozwalające dynamicznie dopasować wymogi aplikacji do aktualnej architektury węzłów i jej obciążenia.

Jeśli celem jest posiadanie po jednym kontenerze aplikacji działającym na każdym z węzłów, to doskonałym rozwiązaniem jest zasób DaemonSet. DaemonSet jest zasobem, który zapewnia uruchomienie po jednej kopi Pod'a na każdym z węzłów lub grupie wybranych węzłów dzięki opcji *nodeSelector*. Możliwe jest nadanie etykiet każdemu z węzłów w oparciu o logikę klucz-wartość. Możliwe byłoby np. dodanie etykiety z kluczem *rola* do każdego z węzłów, a następnie w polu wartości klucza wpisać, czy dany węzeł powinien pracować w trybie *ps*, czy też *worker*. Następnie korzystając z opcji *nodeSelector* wykorzystać dwa zasoby DaemonSet. Jeden, który uruchomi kontener w trybie *ps* na każdym węźle z wartością klucza *role* równą *ps* oraz drugi

działający w analogiczny sposób na wartość *worker*.

Zasadniczym problemem takiego rozwiązania jest, że w zależności od ilości węzłów w gronie tylko tyle kontenerów można uruchomić na raz. Często jednak możliwości podzespołów maszyn służących za węzły pozwala na uruchomienie więcej niż jednego kontenera na jednym węźle bez utraty jakości pracy już działającego kontenera. W takiej sytuacji bardziej racjonalne jest wykorzystanie zasobu StatesfulSet. StatesfulSet przyjmuje w pliku konfiguracyjnym ilość Pod'ów jakie mają istnieć, a następnie umieszcza taką liczbę na dostępnych węzłach. Platforma Kubernetes oferuje również zasób, który nazywa się po prostu Set i w założeniu działa dokładnie jak StatesfulSet. StatesfulSet jednak jest zasobem dedykowanym do systemów rozproszonych i zapewnia bardziej stałe warunki działania Pod'ów niż zwykły Set. Przykładowo wykorzystywana jest logika dbająca o niezmienność warunków takich jak np. hostname Pod'a (zazwyczaj przy starcie nowy Pod dostaje hostname z losowo wygenerowanym numerem identyfikacyjnym). Dzięki temu nawet, jeśli warunki zewnętrzne wymuszą ponowne uruchomienie lub przeniesie Pod'a na inny węzeł, to wiele jego parametrów pozostanie niezmiennych, co zapewni poprawne działanie aplikacji. Analizując powyższe, wykorzystano zasób StatesfulSet do ostatecznej realizacji.

Listing 5.9: Konfiguracja zasobu StatesfulSet dla Pod'ów pracujących w trybie *worker*

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: test-worker
  labels:
    app: distributed-tensorflow
    chart: distributed-tensorflow-0.1.3
    release: test
    heritage: Helm
    role: worker
spec:
  selector:
    matchLabels:
        app: distributed-tensorflow
        release: test
        role: worker
  serviceName: test-distributed-tensorflow-worker
  podManagementPolicy: Parallel
  replicas: 2
```

```
template:
  metadata:
    labels:
      app: distributed-tensorflow
      chart: distributed-tensorflow-0.1.3
      release: test
      heritage: Helm
      role: worker
  spec:
    volumes:
      - hostPath:
          path: /tmp/mnist
          name: vol
    containers:
      - name: worker
        image: "dysproz/distributed-tf:1.7.0"
        imagePullPolicy: IfNotPresent
        command:
          - "python"
          - "train_distributed.py"
          - --learning_rate
          - "0.001"
          - --batch_size
          - "20"
          - --train_steps
          - "200000"
        env:
          - name: WORKER_HOSTS
            valueFrom:
              configMapKeyRef:
                name: test-distributed-tensorflow
                key: worker.hostList
          - name: PS_HOSTS
            valueFrom:
              configMapKeyRef:
                name: test-distributed-tensorflow
                key: ps.hostList
          - name: POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: JOB_NAME
            value: worker
        ports:
        - containerPort: 9000
        volumeMounts:
        - mountPath: /tmp/mnist-log
        name: vol
```

Na początku pliku definiowany jest typ zasobu StatesfulSet oraz wersja API wykorzystywanego - w tym przypadku *apps/v1*. W sekcji *metadata* zdefiniowana jest nazwa zasobu oraz etykiety, które pojawią się na każdym z utworzonych Pod'ów.
W sekcji *spec* zdefiniowana jest specyfika zasobu. W sekcji *template* zdefiniowany jest Pod, a w reszcie zmiennych konfiguracja uruchomienia zasobu StatesfulSet. Sekcja *selector* specyfikuje etykiety już zawarte w sekcji *metadata*. Jest to pozostałość po pierwszych wersjach platformy Kubernetes i od wersji 1.8 ta sekcja może być ominięta. Zmienna *replicas* definiuje ile Pod'ów ma być uruchomionych. Zmienna *serviceName* definiuje nazwę zasobu Service, który ma być użyty w stosunku do Pod'ów uruchomionych przez definiowany StatesfulSet. Opis zasobu Service został zawarty w dalszej części tego rozdziału. Definicja Pod'u w sekcji *template* jest bardzo podobna do ogólnej definicji zasobów, gdzie pod sekcją *spec* zdefiniowana jest specyfikacja. W specyfikacji znajdują się dwa obiekty. Pierwszym jest *volumeMounts*, który określa zewnętrzne dyski, które będą dostępne dla kontenerów w ramach Pod'a. W tym przypadku istnieje tylko jeden dysk, który jest w rzeczywistości fizyczną ścieżką na węźle, który będzie uruchamiał kontener. Dysk wskazuje na ścieżkę */tmp/mnist* i jest w ramach poda rozpoznawalny pod nazwą *vol*. Drugim obiektem jest kontener zdefiniowany w sekcji *containers*. Kontener nazywa się *worker*, korzysta z obrazu *dysproz/distributed-tf:1.7.0*, który jest wcześniej przygotowanym obrazem kontenera i dostępnym z Internetu na hub.docker.io. Zmienna *imagePullPolicy* definiuje politykę pobierania obrazu. W tym przypadku jest ona zdefiniowana na pobranie obrazu jeśli taki obraz nie znajduje się już w lokalnych zasobach węzła. Definicja tej polityki może być przydatna, jeśli wymogi aplikacji definiują potrzebę pobierania najnowszego obrazu przy każdym uruchomieniu (np. żeby pobrać najnowszą wersję obrazu *latest*). Pole *command* jest jednym z najważniejszych pól, ponieważ definiuje komendę, która będzie uruchomiona po starcie kontenera. Jest to lista kolejnych argumentów. W tym przypadku uruchamiany jest skrypt w języku Python z argumentami definiującymi prędkość uczenia, rozmiar paczki danych oraz liczbę kroków uczenia. Należy zauważyć, że nie są podawane adresy innych jednostek pracujących w ramach tej aplikacji, co może wydawać się błędne, ale ostatecznie informacja ta jest przekazywana w inny sposób. W skrypcie zawarta jest logika (co było opisane w rozdziale *Realizacja skryptów uczenia maszynowego*), która przy braku zdefiniowanych adresów innych jednostek przez argument do skryptu korzysta ze zmiennych środowiskowych. Sekcja *env* definiuje właśnie takie zmienne środowiskowe. Zmienne środowiskowe *PS_HOSTS* oraz *WORKER_HOSTS* są odczytywane z zasobu ConfigMap, o którym bardziej szczegółowo opisane jest w dalszej części tego rozdziału. Zmienna środowiskowa *POD_NAME* jest

odczytywana z zmiennych środowiskowych Pod'a, który będzie uruchamiał kontener i jest to zmienna zawarta w sekcji *metadata* pod zmienną *name*. Zmienna środowiskowa *JOB_NAME* jest po prostu wpisaną na twardo wartością *worker* jako, że dany przykład zasobu StatesfulSet jest dla jednostek pracujących w tym trybie. Dodatkowo kontener ma zdefiniowany port numer 9000, która ma być otwarty do komunikacji, jako że skrypt w języku Python nawiązuje połączenie właśnie po tym porcie. Kontener oczywiście ma też podłączony dysk, który jest zdefiniowany wcześniej. Dysk ten będzie podłączony pod ścieżkę kontenera */tmp/mnist-log*.

Innym problemem do rozwiązania jest zapewnienie stabilnego połączenia w ramach grona węzłów dla poprawnego działania aplikacji. Zasób StatesfulSet zapewnia już wiele stabilnych parametrów Pod'ów tworzonych na węzłach, ale nie rozwiązuje całkowicie problemów sieciowości. Pod'y wymagają stabilnego adresu domenowego lub IP, ponieważ biblioteka Tensorflow przyjmuje podczas inicjalizacji te dane w celu mapowania infrastruktury uczącej, a następnie odwołuje się do nich podczas procesu uczenia. Pod'y często się restartują lub przenoszą między węzłami i bardzo prawdopodobne jest, że podczas jednej sesji uczenia kilka Pod'ów zostanie zrestartowany. W takiej sytuacji od platformy Kubernetes wymagane jest utrzymanie stałej adresacji każdego z Pod'ów dla poprawnego działania. W tym celu możliwe jest wykorzystanie zasobu Service. Zasób Service platformy Kubernetes w najprostszym przypadku jest wykorzystywany do równoważenia obciążenia sieci. W klasycznym przypadku w gronie węzłów pracuje kilka Pod'ów, które przykładowo są serwerami WWW. Dzięki zasobowi Service strona WWW otrzymuje stały adres IP lub domenę i w przypadku zapytania na podany adres kieruje ruch do jednego z Pod'ów tak, żeby jak najlepiej wykorzystać dostępne zasoby. W przypadku omawianej aplikacji rozproszonego uczenia maszynowego jednak funkcja balansowania sieci nie jest przydatna. Każde Pod powinien posiadać swój odrębny adres do przekierowania zapytań. Jest to możliwe do zrealizowania przez zasób Service w trybie *Headless*. Tryb ten oznacza, że nie istnieje jeden wspólny adres dla grupy Pod'ów, a każdy Pod dostaje własny adres. Z tego powodu potrzebne jest wprowadzenie zasobu Service w trybie *Headless* do platformy Kubernetes. Zasób StatesfulSet posiada gotowe pole, gdzie można podać nazwę zasobu Service z którego powinien korzystać.

Listing 5.10: Konfiguracja zasobu Service dla Pod'ów pracujących w trybie *worker*

```
apiVersion: v1
kind: Service
metadata:
  name: test-distributed-tensorflow-worker
  labels:
    app: distributed-tensorflow
    chart: distributed-tensorflow-0.1.3
    release: test
    heritage: Helm
spec:
  clusterIP: None
  ports:
    - port: 9000
  targetPort: 9000
  protocol: TCP
  name: worker
  selector:
    app: distributed-tensorflow
    release: test
    role: worker
```

Definicja zasobu Service jest bardzo podobna do definicji zasobu StatesfulSet. Definicja ta również zawiera pola w sekcji *metadata*, która zawiera nazwę zasobu oraz etykiety. Bardziej szczegółowy opis konfiguracji znajduje się w sekcji *spec*. Najważniejszym polem jest *ClusterIP*, gdzie w tym przypadku wpisana jest wartość None, która oznacza wymuszenie trybu *Headless*. Oznacza on, że każdy Pod będzie miał niezależny adres IP oraz nazwę domenową rozpoznawalną z poziomu grona węzłów. Jest to specyficzny przypadek wykorzystania zasobu Service, który znajduje zastosowanie w danej sytuacji. Lista *ports* definiuje porty, które mają być otwarte w ramach zasobu. W tym przypadku jest to port 9000, ponieważ po nim odbywa się komunikacja aplikacji. Pole *targetPort* oznacza port do którego będzie przekierowywany ruch w ramach samego Podu. Ponieważ celem jest przesłanie ruchu na porcie 9000 do portu 9000 Pod'a, to pole to również zawiera wartość 9000. Pole *protocol* odpowiada za wybór protokołu komunikacyjnego. W tym przypadku jest to protokół TCP, który jest wykorzystywany przez aplikację. Protokół ten gwarantuje dostarczenie pełnej wiadomości dzięki zaimplementowanym mechanizmom potwierdzenia otrzymania danych w przeciwieństwie do protokołu UDP. Protokół TCP jest jednak nieco wolniejszy w działaniu od UDP, ale na potrzeby aplikacji różnice te są pomijalne i dużo większy wpływ na prędkość przesyłania będzie miała dostępna przepustowość między poszczególnymi węzłami. Poza wymienionymi polami znajdują się także pola określa-

jące nazwę zasobu oraz pole z etykietami, które muszą znaleźć się na Pod'ach aby polityka zasobu została do nich zastosowana. Powyższy plik konfiguracyjny przedstawia zasób Service dla Pod'ów pracujących w trybie *worker*. Analogiczne zasoby deklarowane są dla Pod'ów w trybie *ps*. Jedyną znaczącą różnicą jest zdefiniowanie numeru portu 8000 zamiast portu 9000, ponieważ dla tego trybu właśnie ten port jest wykorzystywany.

Analizując konfigurację zasobu StatesfulSet można również zauważyć, że pewne dane są pobierane z zasobów typu ConfigMap. ConfigMap jest to zasób, który służy do przechowywania standardowych danych konfiguracyjnych w ramach grona węzłów. Nie jest to zasób dedykowany do przetrzymywania dynamicznej konfiguracji, która jest edytowana, a do danych, które na etapie wdrożenia wprowadzane są do grona i wykorzystywane przez Pod'y tworzone na każdym etapie życia aplikacji, Obecnie często jednak zasób ConfigMap zostaje wykorzystywany również jako dynamiczna konfiguracja przez coraz popularniejsze budowanie aplikacji na platformie Kubernetes w oparciu o Operator Framework, który pozwala w języku Go implementować spersonalizowaną logikę działania własnych zasobów wykorzystując istniejące już zasoby. Jest to jednak kontrowersyjne rozwiązanie budzące wiele dyskusji i na potrzeby niniejszej pracy zasób ConfigMap zostanie wykorzystany w klasyczny, rekomendowany sposób.

Listing 5.11: Konfiguracja zasobu ConfigMap dla całej aplikacji

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: test-distributed-tensorflow
  labels:
    heritage: "Helm"
    release: "test"
    chart: distributed-tensorflow-0.1.3
    app: distributed-tensorflow
data:
  ps.hostList: |
    "test-ps-0.test-distributed-tensorflow-ps:8000,"
    "test-ps-1.test-distributed-tensorflow-ps:8000"
  worker.hostList: |
    "test-worker-0.test-distributed-tensorflow-worker:9000,"
    "test-worker-1.test-distributed-tensorflow-worker:9000"
```

Definicja zasobu jest standardową definicją podobną bardzo do struktury StatesfulSet oraz Service. Sekcją na którą warto zwrócić uwagę jest sekcja *data*. Wszystko, co znajdzie się w tej sekcji może zostać wykorzystane jako dane przy definicji innych zasobów. ConfigMap'a bazuje na bardzo prostej

strukturze danych, które są definiowane jako klucz-wartość. Oznacza to, że niemożliwe jest zagnieżdżanie danych w postaci np. słowników wielopoziomowych. Możliwe jest obejście tego ograniczenia przez wpisanie w pole wartości do klucza łańcuch tekstowy w powszechnie znanym formacie, jak YAML czy JSON, a następnie manualne odczytanie tak zapisanych danych w aplikacji wewnątrz kontenera. Nie jest to jednak zalecana forma prezentacji danych. W przypadku tej definicji zasobu wykorzystano zagnieżdżenie w nazwie klucza wykorzystując kropkę jako podział kolejnych poziomów zagnieżdżeń. Zasób ConfigMap zawiera dwa klucze - *worker* oraz *ps*, które wskazują na dane odpowiednie dla jednego z dwóch trybów aplikacji. Następnie zawarty jest podklucz *hostList*, który jest listą wszystkich nazw domenowych Pod'ów o danej roli. W momencie wdrożenia aplikacji wiadomo już ile będzie jednostek, a dzięki definicji StatesfulSet wiadomo, że będą one zawierać po prostu kolejne cyfry po nazwie zasobu StatesfulSet. Dlatego możliwe jest od razu wpisanie wszystkich nazw domenowych Pod'ów. Podobnie jest z numerem portu, który jest zdefiniowany arbitralnie i może być wpisany już na etapie wdrożenia. Warto zauważyć, że lista jest przedstawiona jako łańcuch tekstowy, a nie typ listy, który jest dostępny w standardzie YAML. Zasób ConfigMap nie wspiera nawet standardowych typów formatu, w jakim jest zapisany. Lista również musi być obsłużona jako łańcuch tekstowy w samej aplikacji działającej w kontenerze. Analizując kod aplikacji w języku Python widać od razu, że flaga obsługująca listę jednostek *worker* oraz *ps* jest zaimplementowana tak, żeby obsługiwać łańcuch tekstowy, a nie listę. Implementacja ta jest wykonana właśnie z myślą o typie danych zamieszczonych w zasobie ConfigMap.

## 5.5 Dystrybucja pakietu aplikacji

Pełne uruchomienie projektu wymaga utworzenia wielu zasobów na platformie Kubernetes, co dla osoby wdrażającej projekt wiąże się z uruchomieniem wielu komend w odpowiedniej kolejności. Dodatkowo jeśli potrzebne są jakieś zmiany w konfiguracji projektu (np. zmiana portu wykorzystywanego przez aplikację w kontenerach), to wymagana jest zmiana wartości w kilku plikach przez osobę zaznajomioną ze specyfiką projektu. Jest to rozwiązanie niewygodne dla projektów wdrażanych produkcyjnie, ponieważ osoba wdrażająca musi być dobrze zaznajomiona z projektem oraz wykonać wiele operacji w systemie produkcyjnym, co zwiększa ryzyko popełnienia błędu.

W celu zminimalizowania tego problemu powstało narzędzie Helm, które udostępnia mechanizm do dynamicznego tworzenia wielu zasobów skompliko-

wanego systemu. Osoba wdrażająca aplikację może ją uruchomić za pomocą jednej komendy i ewentualnie zmienić kluczowe wartości dodając w opcjonalnej fladze plik z własnymi wartościami. Dzięki temu możliwe jest zmiana np. wartości portu wykorzystywanego przez aplikację w kontenerze w jednej zmiennej w pliku *values.yaml* (bo tak jest określany plik konfiguracyjny wartości standardowo) i przy uruchomieniu projektu wartość ta zostanie odczytana i nadpisana we wszystkich miejscach, gdzie powinna być zastąpiona. Platforma Helm pozwala na dystrybucję paczek pod nazwą Chart. Platforma posiada przyjazną stronę internetową z listą publicznych paczek aplikacji. Możliwe jest również zainstalowanie Chart'u z zewnętrznego repozytorium (np. udostępnionego na platformie GitHub), a także korzystając z lokalnych plików.

Omawiany projekt został również zamieniony w Chart platformy Helm, aby ułatwić jego instalację. Struktura plików takiej paczki zawiera się w definicji Chart'u oraz szablonów zawartych w folderze *templates*. Przy instalacji Chart'u platforma Helm wyszukuje plik *Chart.yaml*, a następnie próbuje wypełnić zmiennymi wszystkie szablony w folderze *templates* i uruchomić w platformie Kubernetes. Jeśli nie została podana flaga z plikiem *values.yaml* posiadająca własne wartości do wypełnienia szablonów, to zostanie wykorzystany plik *values.yaml* w folderze głównym Chart'u. Oprócz tego opcjonalnie mogą pojawić się pliki pomocnicze. Dobrą praktyką jest dokumentacja wszelkich aplikacji, dlatego możliwe jest dodanie pliku *README.md* z opisem Chart'u, który wyświetlany jest np. na stronie internetowej platformy Helm. Oprócz tego jeśli paczka danych jest oficjalnie dystrybuowana, to często dodaje się plik *OWNERS* zawierający listę osób opiekujących się danym Chart'em. Platforma Helm udostępnia repozytorium na platformie GitHub, gdzie zamieszczane są publiczne Chart'y. Zaletą takich Chart'ów jest, że wystarczy przy instalacji podać tylko ich nazwę i platforma Helm automatycznie pobierze odpowiedni Chart. W przeciwnym wypadku należy podać dokładny adres dostępny w Internecie. Niestety przez ilość projektów znajdujących się aktualnie repozytorium *helm/charts* na platformie GitHub trudno jest prowadzić kontrybucje w ramach OpenSource, ponieważ nawet właściciele repozytorium nie są w stanie stwierdzić kto jest odpowiedzialny za dany projekt Chart'u. Dlatego wprowadzono plik z listą opiekunów projektu. Pozwala on na automatyczne dodanie w sekcji recenzentów właścicieli projektu, aby mogli ocenić Pull Request na platformie GitHub i ewentualnie zatwierdzić. Innymi możliwymi plikami jest plik *templates/NOTES.txt*, który zawiera notatkę dla użytkownika wyświetlaną po instalacji Chart'u (np. w jaki sposób może on zobaczyć aktualny stan aplikacji). Istnieje również możliwość utworzenia własnych narzędzi do obróbki szablonów (standardowo są

to szablony oparte o język jinja2) - takie definicje zawarte są w pliku *templates/_helpers.tpl*.

Listing 5.12: Struktura plików Chart'u w platformie Helm

```
.
|-- Chart.yaml
|-- distributed-tf.jpg
|-- index.yaml
|-- OWNERS
|-- README.md
|-- templates
|   |-- config.yaml
|   |-- _helpers.tpl
|   |-- NOTES.txt
|   |-- service-ps.yaml
|   |-- service-worker.yaml
|   |-- statefulset-ps.yaml
|   |-- statefulset-worker.yaml
|-- values.yaml
```

Plik *Chart.yaml* składa się z definicji wykorzystywanej wersji interfejsu platformy Kubernetes, krótkiego opisu Chart'u oraz nazwy. Ponadto w pliku znajdują się dwie definicje wersji. Jedną z nich jest pole *version*, które oznacza wersję Chart'u, a drugą *appVersion*, które oznacza wersję aplikacji zawartą w strukturze uruchamianej na platformie Kubernetes. W tym przypadku *appVersion* odnosi się do wersji kontenera wykorzystywanego w projekcie Możliwe jest również zawarcie informacji o źródłach wykorzystanych w projekcie, stronie domowej projektu i opiekunach projektu.

Listing 5.13: Konfiguracja danych Chart'u zawarta w pliku *Chart.yaml*

```
apiVersion: v1
description: |
  A Helm chart for running distributed TensorFlow on Kubernetes
name: distributed-tensorflow
version: 0.1.4
appVersion: 1.7.0
sources:
  - https://github.com/tensorflow/tensorflow
  - https://www.tensorflow.org/deploy/distributed
home: https://www.tensorflow.org
maintainers:
  - name: Dysproz
    email: krasuski.szymon.piotr@gmail.com
```

W pliku z domyślnymi wartościami (który może być wykorzystany również jako wzór poprawnego pliku z niestandardowymi wartościami) zawarte są możliwe opcje do zdefiniowania w projekcie. Zawarte są dwie definicje zmien-

nych dla pobieranych kontenerów (dla jednostek w trybie *worker* oraz *ps*). Definicja taka zawiera informację, ile jednostek danego typu ma się znaleźć w aplikacji (tutaj domyślnie 2). Zawarta jest również informacja o polityce zarządzania Pod'ami, która w zasobie StatesfulSet decyduje czy każdy Pod powinien być uruchamiany pojedynczo, czy równolegle z innymi. W tym przypadku polityka pozwala na równoległe uruchamianie Pod'ów. Sekcja *image* zawiera informację o repozytorium, z którego obraz kontenera zostanie pobrany, o wersji kontenera do pobrania oraz polityce pobierania obrazu. W tym przypadku obraz zostanie pobrany tylko, jeśli nie jest obecny na maszynie uruchamiającej Pod. Dodatkowo istnieje również zmienna z numerem portu wykorzystywanym przez Pod'y w danym trybie. Interesująca jest sekcja *hyperparams*, która zawiera zmienne przekazywane bezpośrednio do skryptu uczącego. W sekcji tej zdefiniowane są zmienne odpowiedzialne za rozmiar paczki danych uczenia, prędkości uczenia oraz ilości kroków uczenia. Domyślnie, jak widać, uczenie będzie działało w nieskończoność, ponieważ zmienna *trainsteps* jest ustawiona na wartość 0.

Listing 5.14: Definicja domyślnych wartości zmiennych paczki w pliku *values.yaml*

```
# Default values for distributed-tensorflow.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.
worker:
  number: 2
  podManagementPolicy: Parallel
  image:
    repository: dysproz/distributed-tf
    tag: 1.6.0
    pullPolicy: IfNotPresent
  port: 9000
ps:
  number: 2
  podManagementPolicy: Parallel
  image:
    repository: dysproz/distributed-tf
    tag: 1.6.0
    pullPolicy: IfNotPresent
  port: 8000
# optimize for training
hyperparams:
  batchsize: 20
  learningrate: 0.001
  trainsteps: 0
```

Posiadając zdefiniowany Chart, można przejść do tworzenia szablonów za-

sobów platformy Kubernetes. Zaczynając od zasobu ConfigMap widać od razu, że szablony bazują na klamrach *{{}}* oznaczających kod w języku jinja2. Język jinja2 daje duże możliwości na formatowanie pliku wyjściowego. Możliwe jest wykorzystywanie podstawowych funkcjonalności języków programowania, jak funkcje warunkowe, czy pętle. Analizując szablon zasobu ConfigMap, na początku szablonu można zauważyć definicje zmiennych. Jak widać, zmienne przekazane przez plik *values.yaml* są osiągalne przez odwołanie się do *.Values* i następnie przez kropki do konkretnych zmiennych. W tym przypadku przypisanie do lokalnych zmiennych wartości z pliku *values.yaml* zostało przeprowadzone dla czytelności szablonu, który w dalszej części wykorzystuje te zmienne, ale za każdym razem pisanie całego odwołania zaciemniałoby kod. W sekcji *metadata* widoczne jest wykorzystanie polecenia *template*, które odwołuje się funkcji pomocniczych zawartych w pliku *_helpers.tpl*. Dzięki definicji funkcji pomocniczej zmienna jest wypełniania wartością wyliczoną w innym pliku, dzięki czemu kod pozostaje przejrzysty. W tej samej sekcji widać również odwołanie do *.Release*, które pobiera dane. *.Release* odwołuje się do zmiennych instalacyjnych Chart'u i jest to gama zmiennych tworzonych na podstawie danych instalacyjnych. Patrząc dalej w sekcję *data* widać pierwsze wykorzystanie funkcji programistycznych języka jinja2. Są one identyczne dla listy jednostek w trybie *worker*, jak i *ps*. Wykorzystana tutaj jest pętla, która iteruje w zakresie liczby zdefiniowanych jednostek (w zmiennej *workerNum*). Jeśli liczba jednostek wynosi 0, to nic nie zostanie wpisane. W przeciwnym wypadku zostanie dołączony łańcuch znaków o zdefiniowanej nazwie, który będzie na końcu zawierał kolejne liczby. Dzięki wykorzystaniu zasobu statesfulSet wiadomo jest, że nazwa będzie ściśle ustalona i kolejne numery Pod'ów będą wypełnianie od 0 do maksymalnej liczby zwiększając co 1. W ten sposób nie odwołując się do platformy Kubernetes, możliwe było utworzenie mapy zmiennych konfiguracyjnych aplikacji.

Listing 5.15: Szablon wypełniany przez mechanizm platformy Helm dla zasobu ConfigMap

```
{{- $workerNum := .Values.worker.number -}}
{{- $workerPort := .Values.worker.port -}}
{{- $psNum := .Values.ps.number -}}
{{- $psPort := .Values.ps.port -}}
{{- $tfService := include "distributed-tensorflow.fullname" . }}
{{- $releaseName := .Release.Name -}}
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ template "distributed-tensorflow.fullname" . }}
  labels:
    heritage: {{ .Release.Service | quote }}
    release: {{ .Release.Name | quote }}
    chart: {{ template "distributed-tensorflow.chart" . }}
    app: {{ template "distributed-tensorflow.name" . }}
data:
  ps.hostList: "
    {{- range $i, $none := until (int $psNum) }}
    {{- if gt $i 0}},
    {{- end }}
    {{$releaseName}}-ps-{{$i}}.{{$tfService}}-ps:{{$psPort}}
    {{- end }}"
  worker.hostList: "
    {{- range $i, $none := until (int $workerNum) }}
    {{- if gt $i 0}},
    {{- end }}
    {{$releaseName}}-worker-{{$i}}.
    {{$tfService}}-worker:{{$workerPort}}
    {{- end }}"
```

W kolejnym szablonie dla zasobu Service również wykorzystane są te same funkcjonalności języka jinja2, co w zasobie ConfigMap. W tym zasobie nie ma potrzeby dynamicznego wypełniania szablonu w pętli lub warunkowe wypełnianie, więc wystarczy wpisanie gotowych wartości zmiennych.

Listing 5.16: Szablon wypełniany przez mechanizm platformy Helm dla zasobu Service dla jednostek pracujących w trybie *worker*

```
apiVersion: v1
kind: Service
metadata:
  name: {{ template "distributed-tensorflow.fullname" . }}-worker
  labels:
    app: {{ template "distributed-tensorflow.name" . }}
    chart: {{ template "distributed-tensorflow.chart" . }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
spec:
  clusterIP: None
  ports:
  - port: {{ .Values.worker.port }}
    targetPort: {{ .Values.worker.port }}
    protocol: TCP
    name: worker
  selector:
    app: {{ template "distributed-tensorflow.name" . }}
    release: {{ .Release.Name }}
    role: worker
```

Najbardziej rozbudowanym szablonem jest ten dla zasobu statesfulSet. Tutaj pojawiają się warunkowe wypełnienia zasobami, ponieważ dodane zostały opcje zaawansowane dla Pod'ów znajdujących się w tym zasobie. Standardowe wartości są wypełnianie podobnie do reszty szablonów, czyli z pliku *values.yaml*, zdefiniowanych funkcji pomocniczych lub wartości *.Release*. Dodatkowo możliwe jest zdefiniowanie w pliku *values.yaml* dodatkowych definicji. Jedną z nich jest dodanie dodatkowych dysków zamontowanych do Pod'a - może to być przykładowo dysk zbierający wszystkie logowane informacje podczas działania kontenera. Jeśli dysk nie zostanie dołączony do Pod'a, to wtedy wszystkie logowania zostaną zapisane jedynie w systemie plików kontenera i będą trudno dostępne. Dyski są definiowane w pliku *values.yaml* według standardowej definicji platformy Kubernetes. Następnie tak zapisana struktura, jest sprawdzana w szablonie czy istnieje, a jeśli istnieje, to jest wypełniania wpisując wartości według standardowego formatowania YAML. Inną opcjonalną definicją jest dodanie zmiennych środowiskowych do Pod'ów. Mogą one się okazać przydatne w przypadku, jeśli nastąpiłaby potrzeba dodać opcjonalne flagi do skryptów w kontenerze (które najczęściej są dostępne również jako zmienne środowiskowe). Ostatnią opcjonalną definicją jest wykorzystanie zasobów pracujących na karcie graficznej. W ten sposób uczenie maszynowe może odbyć się bardziej wydajnie. Jest to też opcja bardziej przystosowana do realnego wykorzystania aplikacji, gdzie bar-

dziej prawdopodobne jest wykorzystanie wielu połączonych ze sobą węzłów posiadających karty graficzne zaprojektowane pod cele obliczeniowe (obecnie na rynku pojawia się coraz więcej kart dedykowanych uczeniu maszynowym).

Listing 5.17: Szablon wypełniany przez mechanizm platformy Helm dla zasobu StatesfulSet dla jednostek pracujących w trybie *worker*

```
{{- $lr := .Values.hyperparams.learningrate -}}
{{- $batchsize := .Values.hyperparams.batchsize -}}
{{- $trainsteps := .Values.hyperparams.trainsteps -}}

apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: {{ .Release.Name }}-worker
  labels:
    app: {{ template "distributed-tensorflow.name" . }}
    chart: {{ template "distributed-tensorflow.chart" . }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
    role: worker
spec:
  selector:
    matchLabels:
      app: {{ template "distributed-tensorflow.name" . }}
      release: {{ .Release.Name }}
      role: worker
  serviceName: |
    {{ template "distributed-tensorflow.fullname" . }}-worker
  podManagementPolicy: {{ .Values.worker.podManagementPolicy }}
  replicas: {{.Values.worker.number}}
  template:
    metadata:
      labels:
        app: {{ template "distributed-tensorflow.name" . }}
        chart: {{ template "distributed-tensorflow.chart" . }}
        release: {{ .Release.Name }}
        heritage: {{ .Release.Service }}
        role: worker
    spec:
{{- if .Values.volumes }}
      volumes:
{{ toYaml .Values.volumes | indent  6 }}
{{- end }}
      containers:
      - name: worker
        image: |
          "{{ .Values.worker.image.repository }}:"
```

```
          "{{ .Values.worker.image.tag }}"
        imagePullPolicy: {{ .Values.worker.image.pullPolicy }}
        command:
        - "python"
        - "train_distributed.py"
{{- if gt (int .Values.worker.gpuCount) 0 }}
        - --num_gpus
        - "{{ .Values.worker.gpuCount }}"
{{- end }}
        - --learning_rate
        - {{ $lr | quote }}
        - --batch_size
        - {{ $batchsize | quote }}
        - --train_steps
        - {{ $trainsteps | quote }}
        env:
        - name: WORKER_HOSTS
          valueFrom:
          configMapKeyRef:
            name: {{template "distributed-tensorflow.fullname".}}
            key: worker.hostList
        - name: PS_HOSTS
          valueFrom:
          configMapKeyRef:
            name: {{template "distributed-tensorflow.fullname".}}
            key: ps.hostList
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: JOB_NAME
          value: worker
        {{- if .Values.worker.env }}
        {{- range $key, $value := .Values.worker.env }}
        - name: "{{ $key }}"
          value: "{{ $value }}"
        {{- end }}
        {{- end }}
        ports:
        - containerPort: {{ .Values.worker.port }}
{{- if .Values.volumeMounts }}
        volumeMounts:
{{ toYaml .Values.volumeMounts | indent 8 }}
{{- end }}
{{- if gt (int .Values.worker.gpuCount) 0 }}
        resources:
          limits:
            nvidia.com/gpu: {{- .Values.worker.gpuCount }}
          request:
```

```
            nvidia.com/gpu: {{- .Values.worker.gpuCount }}
{{- end }}
```

Posiadając tak zdefiniowaną aplikację, można już w prosty sposób uruchamiać ją w wielu środowiskach testowych lub produkcyjnych. Dodatkowo Chart dla platformy Helm został opublikowany w oficjalnym repozytorium na platformie GitHub. Możliwe jest więc zainstalowanie aplikacji z Internetu przez wywołanie prostej komendy *helm install -name distributedApp stable/distributed-tensorflow*.

## 5.6 Środowiska Badawcze

Platforma Kubernetes może być uruchamiana w wielu środowiskach. Możliwe jest uruchomienie jej na fizycznych maszynach, maszynach wirtualnych, a nawet kontenerach. Na potrzeby tej pracy przeprowadzono badania na kilku środowiskach.

### 5.6.1 Środowisko Kind

Pierwszym środowiskiem jest lokalne grono zbudowane w oparciu o kontenery uruchamiane na komputerze domowym. Grupa k8s-sig udostępnia program o nazwie Kind (Kubernetes in Docker) [43], który pozwala tworzyć grona węzłów oparte o platformę Kubernetes właśnie na podstawie kontenerów. Każdy węzeł składa się z obrazu kontenera przygotowanego przez grupę k8s-sig, który wykorzystując narzędzia do łączenia w sieć platformy Docker łączy z innymi kontenerami w grono oparte o platformę Kubernetes. Kind pozwala na dowolną konfigurację grona (to znaczy, że możliwe jest dowolne dobieranie ilości węzłów pracujących oraz węzłów pracujących [worker] jako nadzorca [master]) za pomocą prostego pliku w formacie YAML:

Listing 5.18: Plik konfiguracyjny Kind z jednym 4 węzłami - 3 w trybie *worker* i jeden w trybie *master*

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
nodes:
    - role: control-plane
    - role: worker
    - role: worker
    - role: worker
```

Tak utworzone grono węzłów nadaje się w pełni do uruchomienia projektu przy pomocy platformy Helm. Minusem takiej implementacji środowiska jest zagnieżdżenie projektu w wielu warstwach abstrakcji kontenerów. Oznacza

to, że nadal do dyspozycji mamy jedynie zasoby obliczeniowe maszyny gospodarza, na której uruchomione są kontenery służące za węzły. Następnie na węzłach są uruchamiane kolejne kontenery, które wykonują obliczenia. W ten sposób zasoby obliczeniowe są ograniczone i problematyczne jest skalowanie takiej aplikacji. Z drugiej strony zaletą takiego środowiska jest zniwelowanie ograniczenia prędkości uczenia ze względu na obciążenie sieci. Wszystko dzieje się w ramach tej samej maszyny, a obsługa protokołu warstwy transportowej jest na tyle znikoma, że czas przesyłu danych między Pod'ami jest pomijalnie mały.

Ważnym tematem jest również bufor potrzebny na uruchomienie Pod'ów na każdym z węzłów. Zazwyczaj na świeżo postawionym gronie węzłów nie ma pobranych obrazów kontenerów potrzebnych do uruchomienia Pod'a. Zgodnie z polityką zdefiniowaną w statesfulSet obraz jest wtedy pobierany. Oznacza to, że na każdym węźle przed rozpoczęciem obliczeń musi być pobrany obraz. Tworzy to bufor, który jest niedetermistyczny i może się zdarzyć, że część Pod'ów zacznie działać, podczas gdy inne będą jeszcze pobierać obrazy. Może to zafałszować otrzymane wyniki. Na szczęście środowisko Kind udostępnia opcję, która pozwala pobrać obraz lokalnie do systemu gospodarza. Następnie przy pomocy polecenia *kind load* możliwe jest przekopiowanie obrazu do każdego z kontenerów węzłowych. Dzięki temu każdy Pod niezwłocznie zaczyna pracę bez buforu potrzebnego na pobranie obrazu.

Środowisko uruchomiono na komputerze posiadającym parametry: 8 CPU, 16 GB RAM, Arch Linux OS, procesor Intel(R) Core(TM) i7-8550U @ 1.80GHz, dysk optyczny. Uruchamiając kolejne Pod'y w opisanym środowisku zastosowano ograniczenie dla każdego z nich 256MB RAM oraz 1 CPU, aby uzyskać zróżnicowane wyniki dla testów ze zwiększoną liczbą węzłów.

### 5.6.2 Środowisko chmurowe

Środowiska chmurowe stały się w przeciągu kilku ostatnich lat standardem i większość aplikacji uruchamiana jest właśnie w takim środowisku. Dostawcy usług chmurowych oferują różne rozwiązania wynajmu mocy obliczeniowej pod postacią maszyn wirtualnych, które można organizować w sieci komputerowe dzięki systemowi tzw. VPC (Virtual Private Cloud), który posiada wszystkie znane funkcjonalności dobrze znanych standardowych rozwiązań na maszynach fizycznych (firewall'e, usługi NAT, routing z publicznym IP itp.). Dodatkowo każdy z dostawców stara się dostarczyć różne specjalizowane usługi, aby zapewnić klientowi jak najprostsze rozwiązania gotowe do wdrożenia aplikacji. Tak samo praktycznie każdy dostawca chmurowy oferuje

utworzenie grona węzłów opartych na platformie Kubernetes bez potrzeby manualnego konfigurowania szczegółów. Najpopularniejsze obecnie rozwiązania opierają się o ukrycia warstwy infrastruktury takiego grona węzłów i włączenie systemu dynamicznego skalowania węzłów. Oznacza to, że użytkownik nie definiuje nawet liczby węzłów, ponieważ w razie potrzeby (np. większej ilości Pod'ów) dodatkowe węzły zostaną automatycznie utworzone. Jednak na potrzeby badań niniejszego projektu wymagane jest kontrolowanie ilości węzłów w gronie. Poddano więc analizie trzech najpopularniejszych dostawców usług chmurowych - GCP (Google Cloud Platform), AWS (Amazon Web Services) oraz Azure (powiązany z firmą Microsoft). Zarówno Azure, jak i GCP oferują możliwość utworzenia grona węzłów ze ściśle określoną liczbą węzłów. Jednak w przypadku AWS gotowe rozwiązanie jest wyłącznie dynamicznie skalowane. Oznacza to, że w przypadku AWS wymagane byłoby utworzenie ręcznie zasobu VPC i umieścić w nim pożądaną liczbę instancji, a następnie za pomocą np. narzędzia Ansible utworzyć grono węzłów platformy Kubernetes.

Ostatecznie wybrano dostawcę chmurowego GCP, korzystając z okazji, że podczas badań posiadano bezpłatny dostęp do tej platformy. Dodatkowo firma Google jest właścicielem platformy GCP, a także twórcą platformy Kubernetes. Obecnie firma Google sprawuje tylko patronat nad projektem platformy Kubernetes, ponieważ platforma jest produktem OpenSource (otwartoźródłowym) i jest zarządzana przez społeczność. Jednak wpływy firmy Google są nadal duże. Ponieważ jedna firma zajmuje się obiema platformami, to rozwiązanie chmurowe jest doskonale związane z platformą Kubernetes i wszelkie detale doskonale ze sobą współgrają.

Wykorzystane instancje w środowisko były instancjami generalnego zastosowania według oferty Google Cloud Platform. Każda z instancji posiadała 4 vCPU, 8GB RAM, dysk SSD. Przy uruchomieniu kolejnych Pod'ów nie stosowano ograniczeń na wykorzystane zasoby symulując rzeczywiste zastosowanie komercyjne.

# Rozdział 6

# Badania wydajności i efektywność środowiska dla zmiennej liczby węzłów

## 6.1 Strategia przeprowadzania badań

Jak opisano w poprzednich rozdziałach, architektura projektu składa się w dwóch typów jednostek pracujących - *ps* oraz *worker*. W celu zbadania wpływu zmiany liczby poszczególnych jednostek na wydajność pracy rozproszonego uczenia maszynowego przeprowadzono serię badań dla każdego ze środowisk badawczych.

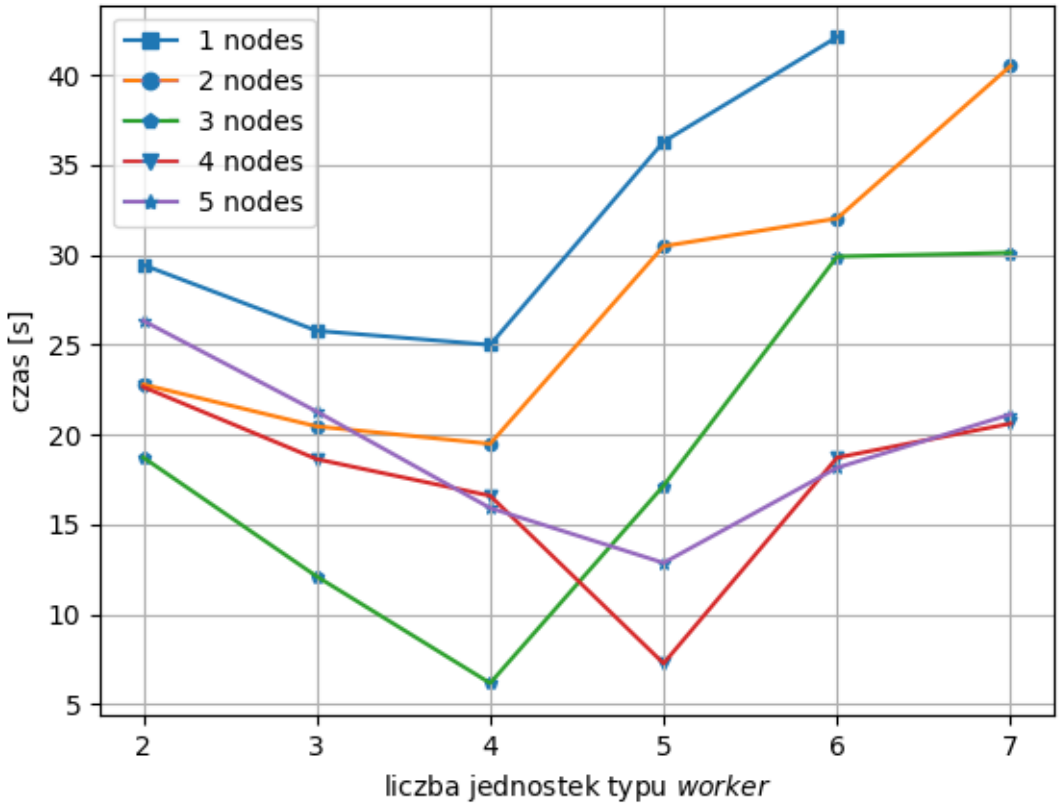
Pierwsza seria badań zakładała niezmienność ilości jednostek pracujących w trybie *ps* przy zmiennej ilości jednostek w trybie *worker*. Przy zwiększaniu ilości jednostek wykonujących obliczenia powinna rosnąć wydajność procesu uczenia. W idealnym przypadku byłaby to zależność liniowa, ale biorąc pod uwagę ograniczenia wynikające z możliwości środowiska ze względu na przepustowość łącza, obsługę zmiennych w jednostkach *ps*, możliwości rozdzielenia zadań obliczeniowych i innych czynników, przewidywana była gorsza wydajność przy większej ilości jednostek *worker* w stosunku do charakterystyki liniowej.

Kolejna seria badań zakładała niezmienność ilości jednostek pracujących w trybie *worker* przy zmiennej ilości jednostek w trybie *ps*. Przy zwiększaniu ilości jednostek zarządzających parametrami modelu uczonego powinna rosnąć wydajność procesu uczenia dla większej ilości jednostek w trybie *worker*. W przypadku małej ilości jednostek w trybie *worker* wpływ powinien

być mało widoczny, ponieważ parametry nie muszą być zebrane, przeliczone i wysłane do wielu węzłów - inaczej niż to jest przy dużej ilości jednostek pracujących.

## 6.2 Wyniki badań dla środowiska Kind

Badania przeprowadzone dla środowiska Kind były wykonane w dwóch etapach. Pierwszy etap zakładał niezmienną liczbę jednostek typu *ps* przy zmiennej liczbie jednostek typu *worker*. Następnie przeprowadzono proces uczenia maszynowego na przykładowym zestawie danych MNIST, 2000 krokach uczenia, modelu z 100 neuronami w warstwie ukrytej oraz 2 jednostkach w trybie *ps* jako reprezentatywnym przykładzie. Proces uczenia powtórzono, zwiększając stopniowo ilość węzłów w gronie, aby zaobserwować również wpływ rozmiaru grona węzłów na proces uczenia. Otrzymane wyniki zaprezentowano na wykresie:

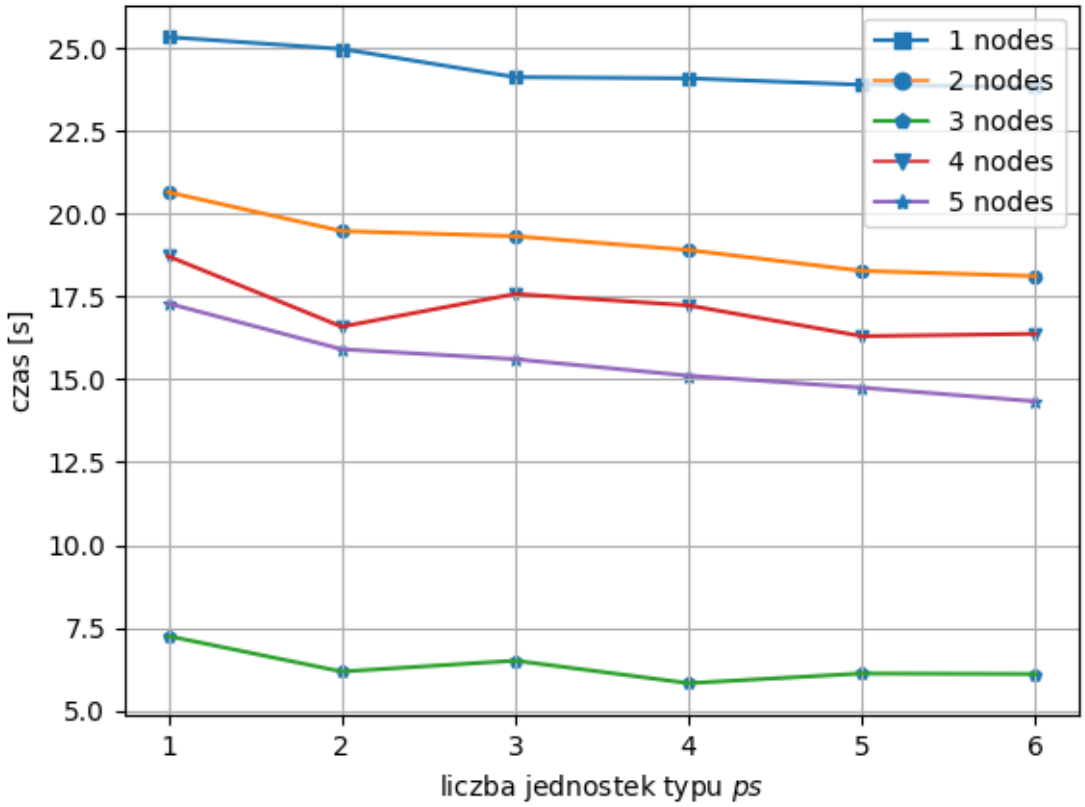



Rysunek 6.1: Porównanie zależności czasu uczenia maszynowego od ilości jednostek typu *worker* dla gron węzłowych o różnej ilości węzłów

Analizując zmianę czasu uczenia dla pojedynczego węzła widać, że czas maleje do danej ilości jednostek typu *worker*, a następnie gwałtownie zaczyna rosnąć. Jest to spowodowane ograniczeniami maszyny zapewniającej zasoby dla środowiska Kind, które są ograniczone. Pod'y uruchamiane w gronie mają ograniczenia zasobów, które mogą maksymalnie zająć, więc w punkcie, gdzie czas uczenia jest najmniejszy, osiągane jest wykorzystanie blisko 100% dostępnych zasobów. Następnie dokładanie kolejnych jednostek rozdziela dostępne zasoby równomiernie na utworzone Pod'y, co powoduje przyznanie coraz mniejszych zasobów per Pod. Jako efekt każdy z Pod'ów wykonuje obliczenia coraz wolniej powodując opóźnienia w przesyłaniu kolejnych wag do jednostek typu *ps*.
Innym elementem wpływającym na czas uczenia jest rozmieszczenie Pod'ów

na węzłach środowiska Kind. Środowisko Kind jest skonfigurowane domyślnie tak, żeby starać się nie wykorzystywać całkowicie dostępnych zasobów maszyny. Jednak jeśli uruchomiona aplikacja na środowisku znacząco wymaga wykorzystania większe ilości zasobów, to wtedy automatycznie środowisko Kind zajmie więcej zasobów. Analizując charakterystyki dla różnej ilości węzłów w gronie, można zauważyć, że minimalny czas uczenia osiągany jest przy 4 jednostkach typu *worker* dla 2 oraz 3 węzłów w gronie oraz 5 jednostek dla 4 i 5 węzłów. Widoczny jest więc nieznaczny postęp przy większej ilości węzłów w gronie. Aczkolwiek bardziej widoczny jest wpływ ilości węzłów na osiągany minimalny czas uczenia. Widać wyraźnie, że między dwoma węzłami a trzema jest znaczący postęp, gdzie grono z trzema węzłami osiągnęło najlepszy wynik. Przy dalszym zwiększaniu ilości węzłów jednak samo utrzymanie środowiska zajmuje więcej zasobów maszyny i przez to minimalny czas uczenia jest gorszy, niż trzech węzłów.

W kolejnym etapie powtórzono badania, ale tym razem utrzymano stałą ilość jednostek typu *worker*, a zmieniano ilość jednostek typu *ps*. Wybrano środowisko z powtórnie zestawem danych MNIST, 2000 krokami uczenia, 100 neuronami w warstwie ukrytej oraz 4 jednostkami typu *worker* jako reprezentatywny przykład. Otrzymane wyniki porównano dla gron węzłów z różną ilością węzłów:




Rysunek 6.2: Porównanie zależności czasu uczenia maszynowego od ilości jednostek typu *ps* dla gron węzłowych o różnej ilości węzłów

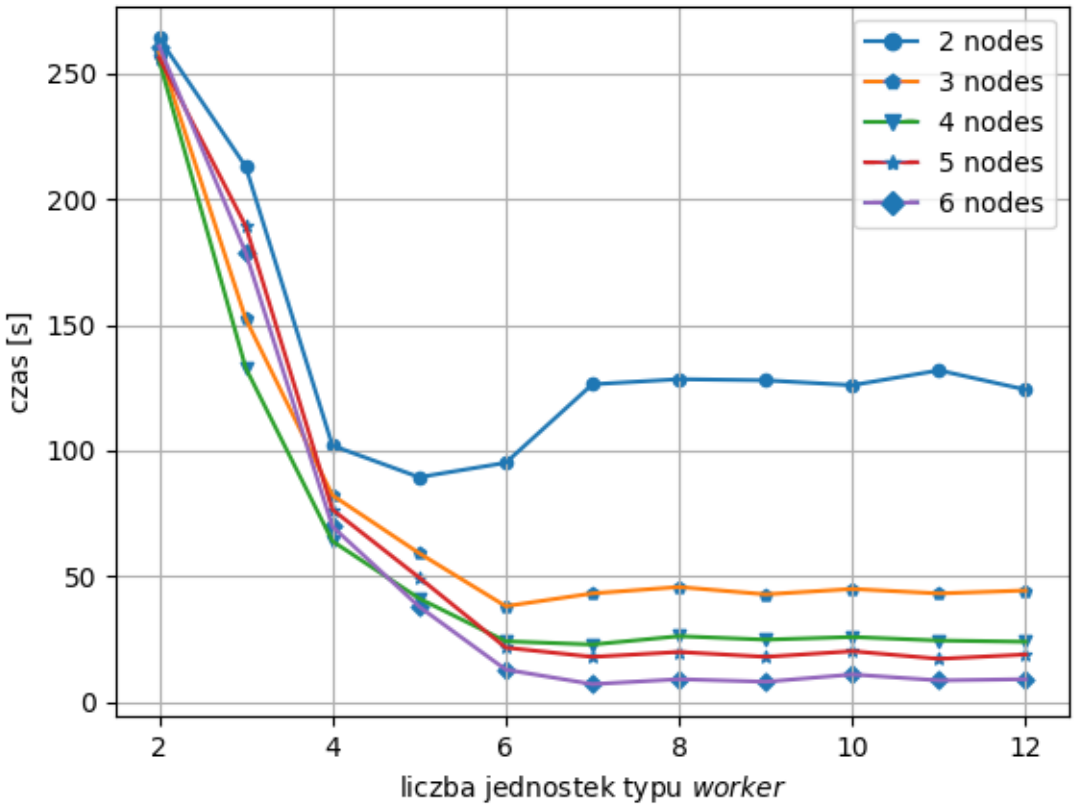
Analizując zmianę jednostek w trybie *ps* widać wyraźnie, że zmiana czasu uczenia nie jest znacząca. Widać jednak lekki trend spadkowy czasu uczenia przy zwiększaniu jednostek *ps* dla każdego badanego grona węzłów. Podział obsługi parametrów w modelu na więcej jednostek pozwala przeliczyć dane pozyskane z kilku jednostek typu *worker* w krótszym czasie, co oznacza również szybsze przesłanie do jednostek *worker* kolejnych danych do uczenia. Jednostki typu *ps* wykorzystują znacząco mniej danych do pracy, niż jednostki *worker*, dlatego ograniczenia zasobów maszyny utrzymujących środowisko Kind nie są widoczne w tym przypadku. Przy modelach bardzo dużych (posiadających wiele warstw ukrytych i dużo więcej neuronów) mała liczba jednostek typu *ps* wprowadza duże opóźnienia, pozostawiając jednostki typu *worker* w stanie bezczynności. Liczba jednostek typu *ps* powinna być dosto-

sowana do rozmiaru uczonego modelu - tym większa, im większy jest model.

## 6.3 Wyniki badań dla środowiska chmurowego

Powtarzając badania w środowisku chmurowym, zmieniono liczbę kroków uczenia maszynowego na 200000 oraz liczbę neuronów w warstwie ukrytej na 1000. Pozostałe parametry pozostają bez zmian.

W pierwszym etapie przeprowadzono badania na zmiennej liczbie jednostek typu *worker* przy niezmiennej liczbie jednostek *ps* równej 2. Uzyskane wyniki przedstawia następujący wykres:

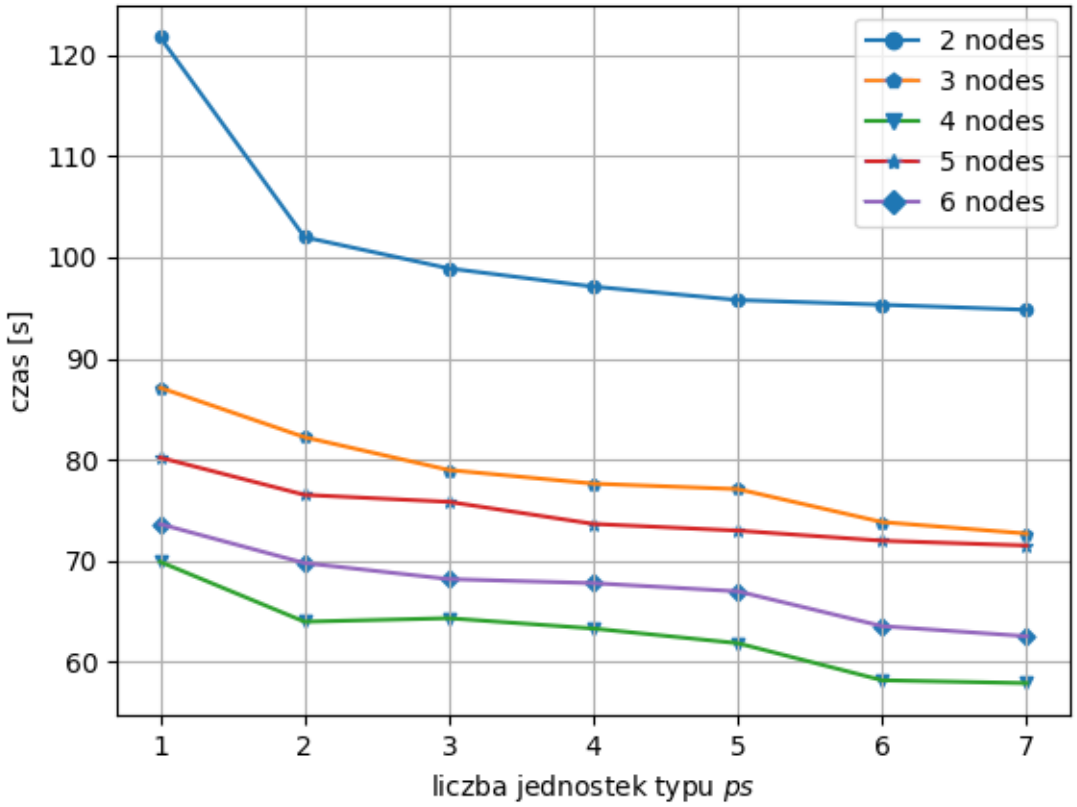



Rysunek 6.3: Porównanie zależności czasu uczenia maszynowego od ilości jednostek typu *worker* dla gron węzłowych o różnej ilości węzłów

Analizując uzyskane charakterystyki, można zauważyć, że nie ma wyraź-

nego minimum, jak to było widoczne w badaniach dla środowiska Kind. W przypadku środowiska chmurowego widać, że czas uczenia maleje aż osiąga poziom, który utrzymuje się przy dodawaniu kolejnych jednostek typu *worker*. Widać wyraźnie, że przy kilku pierwszych jednostkach typu *worker* aplikacja działa z ograniczeniami zasobów, gdzie jednostki *ps* czekają na dane od jednostek *worker* bez operacji do wykonania, więc każda kolejna jednostka *worker* może być obsłużona bez wpływu na resztę jednostek. Natomiast przy 6 jednostkach *worker* czas uczenia ustala się na pewnym poziomie i przy dokładaniu do grona kolejnych jednostek *worker* oscyluje w przy jednym poziomie. Wyjątkiem jest grono węzłów z jedynie dwoma węzłami, gdzie czas uczenia wzrasta przy dodaniu kolejnych jednostek *worker*. Jest to spowodowane ograniczeniem zasobów, który był widoczny w podobny sposób w środowisku Kind. Grona powyżej 2 węzłów pozwalają ominąć ograniczenia zasobów w badanych modelu. Analizując ustalenie się czasu uczenia w każdym z badanych gron węzłowych, można stwierdzić, że jednostki typu *ps* osiągają maksymalną możliwą przepustowość obróbki danych i dokładanie kolejnych jednostek *worker* nie zwiększa efektywności uczenia. Również podział zadań uczenia na wiele jednostek *worker* może być ograniczone przez rozmiar modelu. Im większy model, tym lepiej można go podzielić na wiele zadań, ale nadal istnieją ograniczenia związane z dzieleniem modelu tak, żeby proces uczenia był realistyczny w kontekście całego modelu. Zauważalne jest jednak, że wzrost liczby węzłów w gronie pozwala na osiągnięcie mniejszych czasów uczenia maszynowego.

W kolejnym etapie powtórzono badania, ale przy zmiennej liczbie jednostek typu *ps* i liczbie jednostek *worker* równej 4. Wyniki przedstawia następujący wykres:




Rysunek 6.4: Porównanie zależności czasu uczenia maszynowego od ilości jednostek typu *ps* dla gron węzłowych o różnej ilości węzłów

Analizując wyniki można zauważyć podobną tendencję, jak dla wyników w środowisku Kind. W tym przypadku również zwiększenie liczby jednostek *ps* powoduje zmniejszenie czasu uczenia. Zmiany czasu są bardziej znaczące, niż dla mniejszego modelu badanego w środowisku Kind. Ewidentnie jednostki typu *ps* miały możliwość lepszego podziału modelu oraz rozmiar parametrów przeliczanych był na tyle duży, że każda kolejna jednostka *ps* mogła przejąć część parametrów do obsługi.

# Rozdział 7

# Podsumowanie i wnioski

Otrzymane wyniki potwierdzają spełnienie kryteriów postawionych we wstępie niniejszej pracy.

Wyniki otrzymane z badania zmiany liczby jednostek typu *ps* pozwoliły stwierdzić, że procesy wykorzystane do zarządzania infrastrukturą uczenia rozproszonego nie mają znaczącego wpływu na efektywność uczenia. Udowodniono wręcz, że zwiększenie procesów związanych z zarządzaniem platformą powoduje nieznaczną poprawę czasu uczenia referencyjnego modelu. Dodatkowo nie zaobserwowano zjawiska, gdzie rosnąca liczba jednostek typu *ps* zaczyna negatywnie wpływać na wydajność infrastruktury, co pozwala swobodnie na dowolność w doborze liczby jednostek tego typu. Jest to z pewnością przydatna cecha przy zapewnieniu wysokiej dostępności infrastruktury. Najczęściej w takich zastosowaniach komponenty zarządzające infrastrukturą są rozmieszczone w wielu niezależnych od siebie miejscach. W infrastrukturze fizycznej będą to maszyny podłączone do niezależnych systemów zasilania i sieci komputerowych. W rozwiązaniach chmurowych będzie to umieszczenie infrastruktury w różnych strefach w ramach jednego regionu (najczęściej są to serwerownie oddalone od siebie o ok. 20-30 kilometrów) lub na kilku regionach platformy chmurowej (serwerowniach rozmieszczonych w różnych krajach i regionach kontynentów). Wartości modelu rozłożone na wiele jednostek typu *ps* pozwalają na ich rozłożenie w wielu niezależnych regionach oraz zapewnienie, że nawet gdy jedna z jednostek ulegnie awarii, to utracone zostanie jedynie niewiele parametrów modelu.

Wyniki otrzymane z badania zmiany liczby jednostek typu *worker*, czyli jednostek reprezentujących dostępne zasoby do procesu uczenia, również spełniają założone kryterium. Skalowanie dostępnych zasobów w gronie węzłów skraca czas uczenia maszynowego hiperbolicznie aż do osiągnięcia ograni-

czeń wynikających z właściwości samego uczonego modelu. Zakładając nieograniczona dostępność do zasobów, co jest normą w większości subskrypcji platform chmurowych możliwe jest wykorzystanie tej charakterystyki do implementacji algorytmu autoskalowania procesu uczenia. Może to być zrealizowane np. poprzez coraz bardziej popularne narzędzie Operator Framework. Algorytm tworzyłby kolejne jednostki typu *worker* aż do wykrycia gradientu czasu uczenia poniżej ustalonego progu - wtedy algorytm ustala optymalną liczbę zasobów, aby osiągnąć jak najkrótszy czas uczenia. Większość platform chmurowych rozlicza użytkowników od zużytej mocy obliczeniowej, a nie chwilowego obciążenia samej platformy, czy też sumarycznego czasu pracy na platformie. Oznacza to, że większa chwilowa zajętość zasobów na platformie chmurowej będzie miała zbliżony koszt do mniejszej liczby zasobów pracujących proporcjonalnie dłużej na platformie. Zatem z punktu widzenia biznesowego jest, to zysk czasu potrzebnego do utworzenia gotowego modelu przy porównywalnych kosztach.

Analizując powyższe wyniki, wyraźnie widać, że rozproszona infrastruktura do uczenia maszynowego posiada dużo więcej zalet nad pojedynczymi jednostkami z zaawansowanymi wyspecjalizowanymi komponentami. Analiza techniczna wskazuje, że takie rozwiązanie nie obniża skuteczności procesu uczenia. Analiza biznesowa pokazuje także, że możliwe jest osiągniecie korzyści w skróceniu czasu uczenia oraz prostoty wdrożenia przez wykorzystanie ogólnodostępnych rozwiązań komercyjnych jakimi są platformy chmurowe.

## 7.1 Perspektywy dalszego rozwoju na przyszłość

Projekt wykorzystany na potrzeby badań nadal ma możliwości rozwoju i ulepszeń. Należy pamiętać, że platforma Kubernetes stale się rozwija. Platforma stale uaktualnia swoje API, przez co niektóre definicje mogą się zmieniać. Przykładowo w wersji 1.15 zasób StatesfulSet został przeniesiony z wersji API *apps/v1beta* do wersji *apps/v1*.
Takie zmiany są oczywiste i wymagają śledzenia notatek do wydań, aby na bieżąco utrzymać kod zgodny z aktualną wersją platformy. Jednak poza takim podstawowym utrzymaniem projektu są też inne pola, na których można usprawnić projekt.

Pierwszym z nich jest wykorzystanie zasobu Job w platformie Kubernetes do przeprowadzenia uczenia maszynowego. Aktualnie jest to realizowane za pomocą StatesfulSet, ponieważ zasób ten ma właściwości wymagane do utrzymania odpowiednich adresów Pod'ów, które zostały opisane w rozdziale

dotyczącym architektury projektu. Minusem wykorzystania zasobu StatesfulSet jest, że Pod'y uruchomione z niego są przeznaczone do ciągłego działania. Jest to pożądany mechanizm w przypadku, gdy implementowana jest aplikacja, która ma być wysoko dostępna. W takiej sytuacji Pod zazwyczaj kończy pracę w przypadku błędu i oczekiwany jest restart. Jednak w przypadku procesu uczenia maszynowego Pod kończy pracę również, gdy dojdzie do końca procesu uczenia. Obecnie po zakończeniu procesu zostają jedynie zalogowane informacje dotyczące procesu uczenia oraz sam model, a zasób StatesfulSet restartuje Pod'y i rozpoczyna proces powtórnie, jeśli nie został ręcznie zatrzymany przez użytkownika. Zasób Job jest zaprojektowany do właśnie takich operacji, które mają możliwość zakończyć się sukcesem. Pod jest tylko restartowany, jeśli kod zwrotny procesu działającego w kontenerze jest różny od 0. W przeciwnym wypadku Pod zostaje zatrzymany w stanie 'Completed', żeby użytkownik mógł ewentualnie przejrzeć zasoby wygenerowane podczas operacji. Niestety zasób Job nie oferuje takich funkcjonalności, jak StatesfulSet i wykorzystanie go wymagałoby modyfikacji architektury komunikacji sieciowej Pod'ów.

Komunikacja sieciowa mogłaby być przeniesiona przykładowo na system KubeFlow, który umożliwia szersze zarządzanie sieciowością w gronie węzłów platformy Kubernetes. Aktualnie wykorzystany jest wbudowany system sieciowy biblioteki Tensorflow, która operuje na już istniejącej sieci. Kubeflow jednak pozwoliłby zastąpić ustawienia sieciowe realizowane za pomocą zasobów StatesfulSet oraz Service i zrealizować architekturę w oparciu o zasób Job. Dodatkowo KubeFlow daje również szerokie możliwości monitoringu oraz raportowania działania projektu.

Kolejną możliwością rozwinięcia możliwości systemu jest przeniesienie wdrożenia projektu z platformy Helm na natywny operator platformy Kubernetes. Operatory w platformie Kubernetes są stosunkowo nowym rozwiązaniem, które pozwalają na tworzenie własnych zasobów, korzystając z API platformy Kubernetes. Najczęściej są one realizowane w języku programowania Go (ponieważ sama platforma Kubernetes jest zrealizowana w tym języku) i polegają na utworzeniu kontrolera, które periodycznie kontaktuje się z gronem węzłów i analizuje obecny stan. Jeśli stan jest inny, niż ten zadeklarowany (np. brakuje Pod'ów), to kontroler wysyła odpowiednie zapytania, aby doprowadzić do pożądanego stanu. Przepisanie takiego wdrożenia z platformy Helm na natywny operator umożliwiłoby szersze możliwości monitoringu oraz reagowania na błędy przy procesie uczenia. Również operator działa jako dodatkowy zasób, który może realizować częściowo funkcjonalności innych zasobów. Dlatego możliwe byłoby wykorzystanie potrzebnych

funkcjonalności z zasobu StatesfulSet oraz Service bez konieczności tworzenia takich zasobów. Możliwe byłoby utworzenie zasobu, który łączy w sobie statyczną sieciowość, jak StatesfulSet oraz wykorzystuje możliwości zasobu Job bez restartowania Pod'ów.

Inną perspektywą rozwojową projektu jest implementacja wykorzystania zasobu PersistentVolume. Zasób ten odnosi się do dysków widocznych w gronie węzłów (może to być np. dysk utworzony na platformie chmurowej AWS), który pozostaje niezmienny i niezależny od Pod'ów tworzonych w gronie. Również dysk taki nie istnieje fizycznie nigdzie na węzłach, więc jego dane są bezpieczne od ewentualnych awarii lub przeniesienia Pod'ów wykonujących operacje. Dzięki wdrożeniu dysku widocznego pomiędzy Pod'ami na wielu węzłach możliwe byłoby ujednolicenie logowanych informacji do jednego miejsca oraz sam model byłby zapisywany w konkretnym miejscy, a nie na losowym węźle w zależności tego, gdzie platforma Kubernetes uruchomi odpowiednie Pod'y.

Jeszcze inną możliwością rozwoju jest zaimplementowanie dynamicznego dobierania Pod'ów wykorzystanych do procesu uczenia maszynowego. Już teraz istnieje możliwość autoskalowania Pod'ów tak, żeby zasoby zabierane przez Pod'y na węźle lub ogólnie w gronie węzłów nie przekraczały określonych limitów CPU, pamięci RAM i innych. Na potrzeby projektu uczenia maszynowego użyteczne byłoby skalowanie Pod'ów w zależności od aktualnego stanu grona węzłów. Przykładowo w sytuacji produkcyjnej to samo grono węzłów mogłoby być wykorzystywane przez kilka wdrożeń procesu uczenia maszynowego. Posiadając skalowanie, możliwe byłoby wykorzystanie większej ilości zasobów, gdy aktualnie na gronie węzłów nie są uruchomione inne wdrożenia uczenia maszynowego lub ograniczyć zużycie zasobów, gdy istnieją konkurencyjne procesy uczenia (np. bardziej priorytetowe).

Wymienione perspektywy rozwoju projektu z pewnością nie wyczerpują tematu i na pewno istnieje też wiele innych usprawnień, które mogą wynikać z konkretnych potrzeb produkcyjnych lub polityk organizacji wykorzystujących taki model uczenia maszynowego. W najbliższym czasie możemy spodziewać się dalszego rozwoju narzędzi obsługujących uczenie maszynowe dających zupełnie nowe możliwości rozwoju i ewaluacji przedstawionego rozwiązania. Niewykluczone jest także pojawienie się zupełnie nowych koncepcji i modeli samouczących się systemów. Niewątpliwie jednak, niezależnie od kierunku rozwoju, zagadnienie uczenia maszynowego staje się obecnie jednym z ważnych elementów tworzenia oprogramowania, systemów sterowania czy obsługi współczesnych urządzeń technicznych. Zapotrzebowanie na skuteczne

zarządzanie coraz większymi ilościami pozyskiwanych danych, analizowanie ich i ciągłe automatyczne optymalizowanie procesów online będzie rosło bardzo dynamicznie. Ten trend będzie wymuszał rozwój procesów samouczenia maszyn i w efekcie coraz powszechniejsze ich wykorzystanie.

# Spis skrótów i pojęć

| | |
|---|---|
| węzeł | pojedyncza maszyna (zarówno fizyczna, jak i wirtualna) w gronie większej liczby maszyn tworząca spójny system |
| cluster | grono węzłów, czyli wiele węzłów połączonych w jeden spójny system |
| kontener | Technologia pozwalajaca na izolację alternatywnego systemu operacyjnego na maszynie zarządzana przez i korzystająca z jądra systemu systemu gospodarza |
| VM | (ang. Virtual Machine) Maszyna wirtualna utrzymująca system operacyjne zainstalowany na wirtualnych podzespołach |
| Kubernetes | platforma zarządzająca kontenerami na wielu węzłach |
| MNIST | (ang. Modified National Institute of Standards and Technology database) Baza odręcznie zapisanych cyfr często wykorzystywana do zagadnień uczenia maszynowego |
| bias | waga zerowa stanowiąca wartość stałą dodawaną do wejścia neuronu |
| Google | amerykańska firma oferująca szereg usług technologicznych |
| GCP | (ang. Google Cloud Platform) Platforma chmurowa firmy Google |
| chart | określenie gotowego zestawu manifestóœ platformy Helm służacego do wdrożenia pojedynczej aplikacji |
| systemd | menadżer systemu i usług dla systemu Linux |
| mikroserwis | określenie pojedynczej usługi uruchomionej w systemie wielu usług |
| orchestrator | system automatycznego zarządzania wieloma kontenerami jednocześnie |
| taint | sztucznie nadane oznaczenie elementu systemu blokujące wykorzystanie elementu do pracy systemu lub konkretnych usług systemu |
| Pod | najmniejszy element platformy Kubernetes składający się z jednego lub kilku kontenerów pracujących na jednym węźle w |

| | |
|---|---|
| | ramach jednej usługi |
| Node | pojedynczy węzeł w gronie węzłów powiązanych ze sobą za pomocą platformy Kubernetes |
| DaemonSet | zasób platformy Kubernetes zapewniający uruchomienie kopi Pod'u na każdym z węzłów w gronie |
| Tiller | serwer pośredni platformy Helm służacy do zarządzania wdrożeniem aplikacji |
| Gluon | biblioteka do budowy modeli uczenia maszynowego |
| PyTorch | biblioteka do budowy modeli uczenia maszynowego |
| Keras | biblioteka do budowy modeli uczenia maszynowego |
| Amazon | amerykańska firma oferująca szereg usług technologicznych |
| AWS | (ang. Amazon Web Services) platforma chmurowa firmy Amazon |
| Microsoft | amerykańska firma oferująca szereg usług i produktów technologicznych |
| Azure | platforma chmurowa firmy Microsoft |
| kernel | jądro systemu operacyjnego Linux |
| Docker | narzędzie do uruchamiania kontenerów w ramach pojedynczego węzła |
| Ansible | narzędzie do wykonywania równoległego serii zadań na wielu węzłach |
| Dockerfile | nazwa pliku zawierającego konfigurację kontenera |
| Python | skryptowy język programowania |
| jednostka *ps* | (ang. Parameter Server) typ jednostki wykorzystanej w architekturze rozwiązania do procesowania parametrów modelu uczenia maszynowego |
| jednostka *worker* | typ jednostki wykorzystanej w architekturze rozwiązania do faktycznego procesu uczenia maszynowego |
| RAM | (ang. Random Access Memory) typ pamięci dostępnej w systemie operacyjnym |
| API | (ang. application programming interface) interfejs programowania aplikacji |
| Tensorflow | biblioteka uczenia maszynowego |
| Tensorboard | narzędzie biblioteki Tensorflow do wizualizacji danych |
| RedHat | Firma oferująca komercyjne rozwiązania inżynierskie między innymi produkty chmurowe |
| Openshift | platforma firmy RedHat bazująca na platformie Kubernetes zawierająca gotowe środowisko do komercyjnego użytku |

| | |
|---|---|
| Openstack | platforma firmy RedHat do zarządzania rozproszonym systemem maszyn wirtualnych |
| Service | zasób platformy Kubernetes zapewniający bardziej zaawansowane opcje zarządzania siecią kontenerów |
| ConfigMap | zasób platformy Kubernetes zawierający dane w postacji klucz-wartość |
| Kind | (ang. Kubernetes in Docker) narzędzie tworzące platformę Kubernetes na kilku kontenerach uruchomionych w ramach jednego systemu gospodarza |
| jinja2 | silnik szablonów dla jezyka programowania Python |
| GitHub | platforma oferująca publikację plików źródłowych kodu aplikacji |
| k8s-sig | grupa sympatyków platformy Kubernetes |
| CPU | (ang. Central Processing Unit) sekwencyjne urządzenie cyfrowe pobierające dane, interpretujące je i wykonujące jako rozkazy |
| OS | (ang. Operating System) system operacyjny |
| vCPU | (ang. Virtual Compute Power System) wirtualne CPU |
| StatefulSet | zasób platformy Kubernetes pozwalający uruchomić zdefiniowaną liczbę Pod'ów przy szczególnych warunkach konfiguracji sieciowej |
| DKMS | (ang. Dynamic Kernel Module Support) system zapewniający dynamiczne wsparcie modułów jądra systemu operacyjnego |
| CNI | (ang. Container Network Interface) rozszerzenie platformy Kubernetes o specyficzne zarządzanie komunikacją między Pod'ami w gronie węzłów |
| HA | (ang. High Availability) wysoka dostępność danego systemu lub usługi |
| CNCF | (ang. Cloud Native Computing Foundation) organizacja skupiająca otwartoźródłowe produkty chmurowe |
| framework | struktura sposobu pisania aplikacji, która jest tłumaczona na operacje innego języka programowania |
| playbook | zestaw zadań zdefiniowanych dla narzędzia Ansible |
| TCP | (ang. Transmission Control Protocol) protokół sterowania transmisją zapewniający bezstratną transmisję danych |
| UDP | (ang. User Datagram Protocol) protokół internetowy niegwarantujący dostarczenia przesyłanych danych |
| model TCP/IP | teoretyczny model warstwowej struktury protokołów komunikacyjnych |

| | |
|---|---|
| model ISO OSI | standard zdefiniowany przez ISO (ang. International Organization for Standardization) opisujący strukturę komunikacji sieciowej |
| docker-compose | narzędzie pozwalajace na zautomatyzowane wdrożenie wielu kontenerów równocześnie |
| Go | język programowania popularny w rozwiązaniach chmurowych |
| Kubernetes operator | sposób tworzenia aplikacji na platformę Kubernetes poprzez zastosowanie własnych definicji zasobów oraz kodu kontrolującego zachowanie tych zasobów |
| Operator Framework | narzędzie służące do tworzenia operatorów platformy Kubernetes w języku programowania Go |
| Operator-SDK | narzędzie linii poleceń ułatwiające pracę z operatorami platformy Kubernetes |
| Terraform | narzędzie służące do tworzenia architektury chmurowej bazując na plikach konfiguracyjnych w formacie JSON |
| PyPI | (ang. Python Package Index) Baza bibliotek do języka programowania Python |
| PEP | (ang. Python Enhancement Proposals) Zbiór dokumentów opisujących propozycje usprawnień języka programowania Python sporządzane przez społeczność zebraną wokół tego języka |
| PEP8 | Dokument opisujący propozycję norm stylu pisania kodu w języku programowania Python szeroko uznawany za standard przemysłowy |
| OpenSource | określenie na otwarte źródła, czyli publicznie dostępne i zazwyczaj licencjonowane do powielania na ustalonych warunkach |
| VPC | (ang. Virtual Private Cloud) Wirtualna wyszczególniona przestrzeń robocza w wybranych regionie fizycznym platformy chmurowej |
| Security Group | funkcjonalność platform chmurowych definiująca zasady filtrowania ruchu sieciowego wchodzącego i wychodzącego z VPC |
| instance | zasób platformy chmurowej reprezentujący wirtualną maszynę uruchomioną w VPC |
| batch | paczka danych wykorzystana w jednej epoce uczenia maszynowego |
| epoka | pojedyncza iteracja procesu uczenia maszynowego |
| obraz kontenera | przygotowana wcześniej konfiguracja systemu i udostępniona w postacji kontenera gotowego do uruchomienia |

| | |
|---|---|
| YAML | uniwersalny jezyk formalny do reprezentacji struktur danych |
| Markdown | język znaczników przeznaczony do formatowania tekstu |
| repozytorium | katalog na serwerze zawierający zsynchronizowane pliki źródłowe kodu aplikacji |
| git | system kontroli wersji popularnie używany przy tworzeniu projektów zespołowych |
| WWW | (ang. World Wide Web) internetowy system informacyjny oparty o otwarte standardy |
| IP | (ang, Internet Protocol) główny protokół komunikacyjny w sieci internetowej |
| IPv4 | Protokół IP w wersji 4 |
| log | pojedyncza informacja z aplikacji na temat zdarzenia związanego z działaniem aplikacji |
| Docker Hub | portal zaiwerający publicznie dostępne obrazy kontenerów |
| LTS | (ang. Long Time Support) określenie oznaczające, że dany system lub produkt posiada długi okres wsparcia technicznegi producenta |
| RHEL | (ang. RedHat Enterprise Linux) system operacyjny firmy RedHat pochodny od systemu Linux |
| Ubuntu | system operacyjny firmy Canonical pochodny od systemu Linux |
| HTTPS | (ang. Hypertext Transfer Protocol Secure) protokół przesyłania dokumentów hipertekstowych z szyfrowaniem danych |
| etcd | baza danych wykorzystywana w platformie Kubernetes o sturkturze klucz-wartość |
| Borg | platforma będąca pierwowzorem platformy Kubernetes wykorzystywana na potrzeby pracowników firmy Google |
| Spotify | firma oferująca usługi strumieniowego słuchania muzyki oraz słynąca z wysokiego wykorzystania uczenia maszynowego w swoich produktach |
| skalowanie węzłów | zwiększenie lub zmniejszenie liczby węzłów w gronie |

# Bibliografia


[1] *Data never sleeps 5.0.* Utah: Domo, 2017.

[2] Kratzke, Nane, *About Microservices, Containers and their Underestimated Impact on Network Performance.* Nowy Jork: Cornell University, 2017.

[3] Anderson, Charles, *Docker [Software engineering].* Waszyngton: IEEE Computer Society, 2015.

[4] Vohra, Nardone, *Kubernetes Microservices with Docker.* Nowy Jork: Apress, 1 ed., 2016.

[5] Medel, Víctor, Tolosana-Calasanz, Rafael, Bañares, Ángel, Arronategui, Unai, Rana, Omer, *Characterising resource management performance in Kubernetes.* Saragosa: Elsevier, 2018.

[6] Krill, *Kubeflow brings Kubernetes to machine learning workloads.* Boston: InfoWorld, 2018.

[7] Spillner, Josef, *Quality Assessment and Improvement of Helm Charts for Kubernetes-Based Cloud Applications.* Nowy Jork: Cornell University, 2019.

[8] Schrimpf, Martin, *Should I use TensorFlow.* Augsburg: Augsburg University, 2016.

[9] Olston, Christopher, Fiedel, Noah, Gorovoy, Kiril, Harmsen, Jeremiah, Lao, Li, Fangwei, Rajashekhar, Vinu, *TensorFlow-Serving: Flexible, High-Performance ML Serving.* Nowy Jork: Cornell University, 2017.

[10] Tan, Wai-Tian, Liston, Rob, Apostolopoulos, John, Zhu, Xiaoqing, *Distributed Machine Learning.* Waszyngton: Cisco Systems Inc., 2018.

[11] Tensorflow, "Distributed training with tensorflow." `https://www.tensorflow.org/guide/distributed_training`, 2020.

[12] Google Cloud Platform, "Running distributed tensorflow on compute engine." https://cloud.google.com/solutions/running-distributed-tensorflow-on-compute-engine, 2020.

[13] Kubeflow, "Tensorflow training (tfjob)." https://www.kubeflow.org/docs/components/training/tftraining/, 2020.

[14] Gorauskas, *Managing services in Linux: past, present and future.* Houston: Linux Journal, 2015.

[15] Corradi, Fanelli, Foschini, *VM consolidation: A real case based on OpenStack Cloud.* Bolonia: Elseiver, 2012.

[16] Obasuyi, Sar, *Security Challenges of Virtualization Hypervisors in Virtualized Hardware Environment.* Lefke: International Journal of Communications, Network and System Sciences, 2015.

[17] Cherkasova, Gupta1, Vahdat1, *When Virtual is Harder than Real: Resource Allocation Challenges in Virtual Machine Based IT Environments.* San Francisco: XenQoS, 2017.

[18] Merkel, *Docker: lightweight linux containers for consistent development and deployment.* Houston: Linux Journal, 2014.

[19] Mozilla Firefox, "Facebook container add-on." https://addons.mozilla.org/en-US/firefox/addon/facebook-container.

[20] Preeth, Mulerickal, *Evaluation of Docker containers based on hardware utilization IEEE Explore.* Waszyngton: IEEE Xplore, 2015.

[21] Docker, "Docker hub." https://hub.docker.com/.

[22] Docker, "Dockerfile reference." https://docs.docker.com/engine/reference/builder/.

[23] Linux Foundation, "Docker containers: What are the open source licensing considerations?." https://www.linuxfoundation.org/blog/2020/04/docker-containers-what-are-the-open-source-licensing-considerations/, 2020.

[24] Tarasov, Rupprecht, *In Search of the Ideal Storage Configuration for Docker Containers.* Waszyngton: IEEE, 2017.

[25] Dua, Kohli, Konduri, *Learning Docker networking*. Birmingham: Packt Publishing, 1 ed., 2016.

[26] Combe, Martin, Di Pietro, *To Docker or Not to Docker: A Security Perspective*. Waszyngton: IEEE, 2016.

[27] Ansible, "docker_container – manage docker containers." https://docs.ansible.com/ansible/latest/modules/docker_container_module.html.

[28] Docker, "Overview of docker compose." https://docs.docker.com/compose/.

[29] Google, "Kubernetes: Production-grade container orchestration." https://kubernetes.io/.

[30] Google, "Kubernetes components." https://kubernetes.io/docs/concepts/overview/components/.

[31] Google, "The kubernetes api." https://kubernetes.io/docs/concepts/overview/kubernetes-api/.

[32] RedHat, "Operator framework." https://github.com/operator-framework.

[33] Google, "Pod overview." https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/.

[34] Google, "Replicaset." https://kubernetes.io/docs/concepts/workloads/controllers/replicaset.

[35] Google, "Daemonset." https://kubernetes.io/docs/concepts/workloads/controllers/daemonset.

[36] Helm, "Helm: The package manager for kubernetes." https://helm.sh/.

[37] Helm, "Using helm." https://helm.sh/docs/intro/using_helm/.

[38] AWS, "Introducing gluon: a new library for machine learning from aws and microsoft." https://aws.amazon.com/blogs/aws/introducing-gluon-a-new-library-for-machine-learning-from-aws-and-microsoft/, 2017.

[39] PyTorch, "Pytorch." https://pytorch.org.

[40] Google, "Tensorflow." https://www.tensorflow.org/.

[41] Google, "Keras." https://keras.io/.

[42] Google, "Google cloud platform." https://cloud.google.com/.

[43] Kubernetes Sigs, "Kubernetes in docker." https://github.com/kubernetes-sigs/kind.