POLITECHNIKA KRAKOWSKA
im. T. Kościuszki
Wydział Inżynierii Materiałowej i Fizyki

Katedra Fizyki

Kierunek studiów: Fizyka Techniczna
Specjalność: Modelowanie Komputerowe

STUDIA STACJONARNE

PRACA DYPLOMOWA

INŻYNIERSKA

Gabriela Białoskórska

Realizacja systemu webowego do symulacji wybranego zagadnienia fizycznego w oparciu o konteneryzację i chmurę obliczeniową

Implementation of a web system to simulate a selected physical problem based on containerization and cloud computing

Promotor:
Dr **Radosław Kycia**

Kraków, rok akad. 2021/2022

Niniejszą pracę dedykuję moim rodzicom, którzy nieustannie wspierali mnie w trakcie studiów oraz podczas pisania pracy.
Za nieocenioną pomoc oraz wyrozumiałość przy realizacji niniejszej pracy, pragnę złożyć serdeczne podziękowania mojemu promotorowi dr Radosławowi Kyci.

Spis rysunków

Abstrakt

Dynamiczny rozwój technologii informatycznych, trwający nieprzerwanie od XX wieku, przyczynił się do znacznego wzrostu mocy obliczeniowej komputerów. Fakt ten sprawił, że stały się one podstawowym narzędziem badań naukowych. Z czasem tradycyjne, fizyczne serwery zaczęto zastępować maszynami wirtualnymi, które wpłynęły na rozwój popularnego obecnie Cloud Computingu (chmury obliczeniowej). Coraz częściej w procesie tworzenia infrastruktury IT wykorzystuje się również konteneryzację, będącą kolejną warstwą abstrakcji w procesie tworzenia architektury, umożliwiającą izolację wykorzystywanych zasobów.

Przytoczone powyżej technologie są obecnie powszechnie stosowane w modelowaniu komputerowym. Najpopularniejszym, a zarazem najbardziej spektakularnym przykładem wykorzystania nowoczesnych technologii informatycznych do prowadzenia badań naukowych na płaszczyźnie fizycznej jest *The Worldwide LHC Computing Grid* (WLCG), umożliwiający przechowywanie i analizę danych uzyskiwanych co roku z Wielkiego Zderzacza Hadronów. Niniejsza praca ma na celu ukazanie potencjału wykorzystania chmury obliczeniowej oraz konteneryzacji w badaniach, bazując na symulacji eksperymentu Rutherforda. Ma ona zatem charakter interdyscyplinarny, poruszając zagadnienia zarówno z zakresu fizyki jak i informatyki. Przedstawia możliwości jakie oferują badaczom nowoczesne technologie informatyczne.

Abstract

The dynamic development of information technologies, which lasts continuously since the 20th century, has contributed to a significant increase in the computing power of computers. This fact made them the basic tool of scientific research. Over time, traditional physical servers began to be replaced with virtual machines, which influenced the development of the currently popular cloud computing. More and more often, in the process of creating IT infrastructure, containerization, which is another layer of abstraction in the process of creating architecture, is also used, enabling isolation of resources.

The technologies cited above are now widely used in computer modeling. The Worldwide LHC Computing Grid (WLCG) is the most popular and, at the same time, the most spectacular example of using modern Information Technologies to conduct scientific research in Physics, which enables the storage and analysis of data obtained each year from the Large Hadron Collider. This paper aims to show the potential of using cloud computing and containerization in research, based on the simulation of the Rutherford experiment. It has therefore an interdisciplinary character, covering issues in both physics and computer science. It presents the possibilities which modern information technologies offer researchers.

Spis treści

Wstęp

0.1 Cel pracy

Celem niniejszej pracy było stworzenie systemu webowego, użytecznego w tworzeniu symulacji układów fizycznych oraz ich modelowaniu. Realizację aplikacji działającej z poziomu przeglądarki oparto na wykorzystaniu chmury obliczeniowej oraz konteneryzacji. Rozwiązanie to może zostać wykorzystane na gruncie nauk ścisłych, jako efektywne narzędzie do przeprowadzania symulacji fizycznych. Umożliwia ono wykonywanie ich na dowolnym środowisku oraz bez potrzeby zwiększania mocy obliczeniowej komputera.

0.2 Zakres pracy

Praca została podzielona na dwie części: teoretyczną i praktyczną. Teoretyczna część pracy składa się z dwóch działów. Pierwszy z nich traktuje o podstawach fizycznych stworzonej symulacji, opisując zagadnienia z zakresu fizyki atomowej - jej historię oraz eksperyment Rutherforda. Drugi przybliża podstawowe pojęcia związane z modelowaniem komputerowym; ukazuje historię rozwoju wirtualizacji, prowadzącą do powstania chmur obliczeniowych oraz konteneryzacji, a także opisuje wykorzystane na potrzeby niniejszej pracy serwisy i narzędzia.

Część praktyczna ukazuje sposób realizacji systemu webowego. Przybliża zasadę działania aplikacji oraz uzasadnia wykorzystanie konteneryzacji. Przedstawia także konfigurację serwisów, tworzących infrastrukturę w chmurze obliczeniowej i sposób jej wdrożenia. Ostatnie rozdziały opisują uzyskane wyniki oraz podsumowują całość wykonanej symulacji, bazującej na doświadczeniu z zakresu fizyki. Jest to wirtualne odwzorowanie eksperymentu Rutherforda.

0.3 Metodyka

Trzon systemu webowego (aplikacja webowa) napisany został w języku Python, dzięki wykorzystaniu biblioteki *Matplotlib*. Symuluje on eksperyment Rutherforda. Gotową aplikację umieszczono w wirtualnym kontenerze, korzystając z platformy *Docker*. Zarządzanie zadaniami i serwisami uruchamianymi w kontenerach, a także deklaratywną konfigurację i automatyzację umożliwiło narzędzie orkiestracyjne - *Kubernetes*. Całość opisanej infrastruktury uruchomiona została w chmurze obliczeniowej (*Microsoft Azure*), dzięki wykorzystaniu odpowiednich, opisanych w części teoretycznej serwisów.

Część I

Część teoretyczna

Rozdział 1

Podstawy fizyczne

1.1 Wstęp - historia fizyki atomowej

Powstanie fizyki atomowej, działu opisującego naukowe badania struktury atomu, określa się na lata trzydzieste ubiegłego wieku. Jej podstaw należy jednak szukać znacznie wcześniej, bowiem podobnie jak w przypadku wielu innych dziedzin nauki, podwaliny fizyki atomowej narodziły się już w starożytnej Grecji. Za ojca atomistyki uważa się Demokryta z Abdery (460-370 r. p.n.e.), głoszącego wówczas pogląd, że materia złożona jest z niepodzielnych cząstek zwanych atomami. Stąd samo określenie - *atomos* (niepodzielny) [44]. Według Demokryta atomy posiadały różne wielkości i kształty, a ich nieustanny ruch, prowadzący do grupowania i rozpraszania się, tworzył intrygujący nas wszechświat [59]. Teorie atomistyczne propagował również Platon i Arystoteles. Mimo to, atomizm przerodził się w znaną nam fizykę atomową ponad dwa milenia później.

Atomizm podzielić można na trzy rodzaje, zgodnie z chronologią odkrywania określonych zjawisk. Wobec tego wyróżniamy: atomizm *materii*, atomizm *zjawisk elektrycznych* oraz atomizm *energii*, opisane w literaturze [44]. Pierwszy z nich wynika z uznania faktu, że wszystkie pierwiastki chemiczne zbudowane są z atomów, czego dowiodły badania chemiczne. Pierwszy model atomowy, sformułowany przez Prouta (1815 r.) zakładał, że atomy wszystkich pierwiastków zbudowane są z atomów wodoru, co doprowadziło do uporządkowania pierwiastków w oparciu o ich chemiczne właściwości (*układ okresowy L. Meyera i D.I. Mendelejewa* - 1869 r. [44]). Prout zauważył, że masy atomowe można do pewnego stopnia wyrazić poprzez wielokrotność masy atomowej wodoru. Hipoteza ta upadła, gdy w miarę coraz dokładniejszego określania mas atomowych, wyniki zaczęły znacznie od niej odbiegać [21]. Obecnie wiemy, że Prout w rzeczywistości odkrył istnienie protonów i neutronów, jako podstawowych składników materii. Wodór, posiadając wyłącznie jeden proton (lub neutron w swoich izotopach), stał się dla niego najmniejszym elementem, z jakiego może składać się pierwiastek.

Atomizm zjawisk elektrycznych odkryto w 1833 r. za pośrednictwem badań Michaela Faradaya, angielskiego uczonego. Dotyczyły one pomiarów elektrolizy cieczy. Bazując na wynikach ilościowych wykonanych pomiarów Faraday wnioskował, że istnieją „atomy" elektryczności (elektrony), związane z atomami materii [44].

Atomizmowi energii przypisać można bardzo konkretną datę powstania: 14 grudnia 1900 r. - wówczas Max Planck wygłosił prawa promieniowania *ciała doskonale czarnego*. Pochłania ono w 100% padające na nie promieniowanie elektromagnetyczne oraz emi-

tuje własne, gdy nagrzane zostanie do odpowiedniej temperatury T. Niejednokrotnie modeluje się je jako wnękę z niewielkim otworem, przez który promieniowanie dostaje się do wnętrza ciała. Następnie odbija się ono wielokrotnie od jego wewnętrznych ścianek, nie wydostając ponownie na zewnątrz. Stąd wynika niemalże idealna absorbcja ciała doskonale czarnego. Pomiar widma spektralnego promieniowania, pozostającego w równowadze termicznej ze ściankami ciała, możliwy jest dzięki wspomnianemu otworowi [40]. Planck wyprowadził słynny wzór na gęstość spektralną zakładając, że materia ciała doskonale czarnego stanowi zbiór klasycznych oscylatorów harmonicznych [40]. Twierdził, że energia tychże oscylatorów może przyjmować jedynie dyskretne wartości, stając tym samym w opozycji do panującej ówcześnie, klasycznej teorii, zgodnie z którą wartości energii tworzą kontinuum [44].

Na przestrzeni XIX wieku narodziła się również odrębna gałąź atomizmu, badająca właściwości gazów - atomizm *ciepła*. Składają się na niego między innymi hipoteza Avogadra (1811 r.), kinetyczna teoria gazów Clausiusa i Boltzmanna (1870 r.) oraz badania Gay-Lussaca (1808 r.). Doprowadziły one do wyjaśnienia praw termodynamiki na podstawie nieustannego ruchu i zderzeń atomów [44].

Najsilniejszy wpływ na wiedzę, jaką ludzkość dysponuje obecnie w dziedzinie fizyki atomowej wywarły badania widm optycznych, charakterystycznych dla danych pierwiastków. Z czasem gałąź fizyki atomowej powiązana została z fizyką kwantową, doprowadzając do sformułowania przez Bohra podstawowych zasad kwantowania orbit elektronów w atomach [44].

Fizyka atomowa stanowi podstawę także innych dyscyplin badawczych, między innymi chemii i biologii. Opisuje ona strukturę atomów i ich wzajemne oddziaływania, także z polem elektrycznym i magnetycznym. Niniejszy rozdział pracy przedstawi w sposób szczegółowy jeden z najważniejszych eksperymentów fizyki atomowej - rozpraszanie Rutherforda, który umożliwił odkrycie jądra atomowego.

1.2 Budowa i rozmiary atomu

1.2.1 Definicja atomu

Atom jest najmniejszym, niezmiennym składnikiem pierwiastka chemicznego. W wyniku reakcji chemicznych lub pod wpływem temperatury atomy mogą ulec zmianie, jednak nieznacznej co do swojego stopnia jonizacji. Temperatura ta musi natomiast charakteryzować się określoną wielkością energii równoważnej kT, gdzie k jest stałą Boltzmanna, a T temperaturą w skali Kelvina. Nie może ona przekroczyć wartości kilku elektonowoltów [44].

Wiadomo obecnie, że atom zbudowany jest z cząstek mniejszych niż on sam: protonów (obdarzonych ładunkiem dodatnim), neutronów (elektrycznie obojętnych) oraz elektronów (posiadających ładunek ujemny). Protony wraz z neutronami skupione są w jądrze atomowym, elektrony natomiast znajdują się poza nim. Ponadto zarówno protony jak i neutrony można podzielić na kwarki, uważane obecnie za tzw. cząstki elementarne[1], podlegające oddziaływaniom silnym [58]. Bezmasową cząstką elementarną pośredniczącą w oddziaływaniach silnych jest *gluon*, który przenosi ładunek kolorowy

[1]Cząstka elementarna - najmniejszy, niepodzielny element formujący materię. Podział cząstek elementarnych opiera się obecnie na Modelu Standardowym. Wyróżnia on fermiony, odpowiedzialne za tworzenie materii oraz bozony, przenoszące oddziaływania [59].

[26]. W rozumieniu fizyki współczesnej atom nie jest już zatem obiektem niepodzielnym, wobec czego jego pierwotna nazwa przestaje określać go w sposób dosłowny.

Z uwagi na neutralność elektryczną atomu, liczba protonów zawartych w jądrze jest równa liczbie elektronów i nosi nazwę *liczby atomowej (Z)*. Pojęcie *liczby masowej (A)* oznacza natomiast sumę protonów i neutronów, stanowiących o masie atomu. Atomy tego samego pierwiastka mogą występować w formach różniących się liczbą masową - są one wówczas *izotopami* danego pierwiastka [59].

Idee określające czym jest elementarny składnik materii uległy drastycznej zmianie na przestrzeni rozwoju cywilizacji. Wbrew wczesnoatomistycznym teoriom wiemy już, że atom jest w istocie obiektem złożonym, ponadto nie w pełni trwałym i niezmiennym, czego dowodzą rozpady promieniotwórcze niektórych nuklidów. Koncepcja nieskończoności możliwych rodzajów atomów, postulowana w filozofii antycznej, sprowadzona została do uporządkowanego względem własności układu elementów. Rozwój wiedzy z zakresu fizyki atomowej wynika z wielu lat zarówno rozważań filozoficznych jak i badań naukowych. Sposób pojmowania samej budowy atomu, który także zmieniał się i ewoluował na przestrzeni lat, przedstawiony zostanie w kolejnym rozdziale.

1.2.2 Modele budowy atomu

Jak zostało już wspomniane, pierwsze modele atomu zrodziły się w starożytnej Grecji. Leukippos z Miletu (VI-V w. p.n.e.) określał go jako niepodzielną cząstkę, charakteryzującą się określonym miejscem w przestrzeni. Teorię tę rozwinął Demokryt z Abdery (ok. 460-370 p.n.e.), tworząc tym samym podwaliny atomistyki. Jego zdaniem liczba atomów była stała, a one same nie mogły powstawać z próżni ani się w nią przeradzać. Różniły się kształtem, wielkością oraz stopniem „przylegania" do siebie nawzajem, warunkując cechy tworzonych ciał. Ich nieustanne zderzenia wpływały na zmiany otaczającego nas świata [59].

W tym samym okresie Platon (427-347 p.n.e.) rozwinął koncepcję *atomizmu geometrycznego*, będącego próbą matematycznego rozwikłania zmian zachodzących w przyrodzie. Opierał się on na elementach, będących złożeniem trójkątów, określanych przez Platona mianem *elementów matematycznych*. Budulec wszelkiej materii stanowić miały wielościany foremne, powstające ze złożeń elementów matematycznych. Nieco później Epikur (341-270 p.n.e.) wysunął koncepcję bazującą na poglądach Demokryta, twierdząc jednak, że atomy zbudowane są z pewnej ilości „najmniejszych cząstek", funkcjonujących jedynie w ich wnętrzu [59].

Rozwój koncepcji atomistycznych znacznie podupadł w okresie średniowiecza, gdy za jedyne słuszne postulaty tłumaczące zjawiska przyrodnicze uznawano prawdy religijne, a w późniejszym okresie również poglądy Arystotelesa. Dopiero w renesansie nastąpiło realne odrodzenie się nauk Demokryta, między innymi za sprawą Galileusza, postulującego, że materia zbudowana jest z nieskończonej ilości punktowych cząstek (1564-1642 r.) i Roberta Boyle'a (1627-1691 r.), który stworzył własną definicję pierwiastka. Również Isaac Newton (1642-1727 r.) twierdził, że materia zbudowana jest z cząstek elementarnych - korpuskuł [59].

Pierwsze szczegółowe koncepcje dotyczące budowy atomu wysunął John Dalton (1766-1844 r.), bazując na odkryciu chemicznego prawa stosunków wielokrotnych. Wiadomym było już wówczas, że pierwiastki tworzą związki chemiczne. Jednak postulat, że *łączą się one ze sobą w stałych stosunkach wagowych, które określają niewielkie liczby całkowite* [60] był rewolucyjny, bowiem ukazywał, że niektóre atomy są lżejsze od pozostałych. Dalton przypisał atomom cechy jakościowe pierwiastków przez nie tworzonych, wysnuwając pięć postulatów [59]:

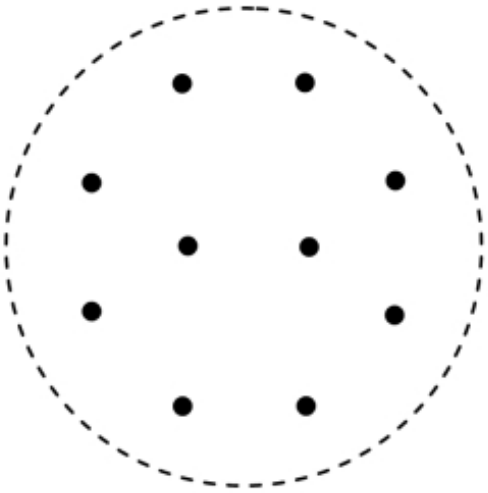
- Wszystkie ciała składają się z atomów, które powiązane są ze sobą wzajemnie siłami przyciągania;
- Wszystkie atomy jednorodnego ciała charakteryzują się takimi samymi własnościami (masą, wielkością itp.);
- Różne pierwiastki zbudowane są z różnych atomów, które różnią się przede wszystkim pod względem ciężaru. Zarówno atomy, jak i pierwiastki z nich zbudowane są niezmienne i nie mogą się wzajemnie w siebie przekształcać;
- Reakcje chemiczne mogą doprowadzić do zmiany połączeń atomów, nie mogą jednak doprowadzić do ich podziału;
- Proces tworzenia się związku chemicznego polega na powstaniu cząsteczek, zawierających określone ilości atomów każdego z budujących je pierwiastków w odpowiednich proporcjach.

W XIX w. znaczący wpływ na rozwój koncepcji atomu stanowiło odkrycie charakterystycznych drgań drobin zawieszonych w cieczy, zwanych inaczej *ruchami Browna*, z uwagi na ich odkrywcę - Roberta Browna (1773-1858 r.) [103]. Są one tym silniejsze, im mniejszych cząstek zawiesiny dotyczą oraz im wyższa jest temperatura. Badania nad tym zjawiskiem prowadzili między innymi Albert Einstein, Marian Smoluchowski oraz Łukasz Bodaszewski. Pozostałe, dokonane w XIX wieku odkrycia są jednymi z najbardziej przełomowych, wobec czego opisane zostaną w odrębnych podrozdziałach.

1.2.2.1 Model Thomsona

W istocie model Kelvina-Thomsona, bazował na wykorzystaniu nowoodkrytej wówczas cząstki - elektronu. Swoją nazwę zawdzięcza Williamowi Thomsonowi (1824--1907 r.), odkrywcy temperatury zera bezwzględnego, twórcy drugiej zasady termodynamiki oraz Josephowi Thomsonowi (1856-1940 r.), laureatowi nagrody Nobla z 1906 roku za badania nad przewodnictwem elektrycznym gazów i odkrycie elektronu. Teoria Lorda Kelvina (Williama Thomsona) budowy atomu polonu stworzyła podwaliny modelu atomu Thomsona jaki znamy po dziś dzień.

Joseph Thomson opisywał atom jako równomiernie naładowaną dodatnio kulę, wewnątrz której w jednorodny sposób rozmieszczone były elektrony. Dodatni ładunek kuli wynikał z musu zrównoważenia całkowitego ładunku ujemnego elektronów, w celu zachowania równowagi elektrycznej atomu [59]. Z uwagi na wygląd, model ten określany jest niejednokrotnie mianem *ciastka z rodzynkami*.



Rysunek 1.1: Model atomu Thomsona [61].

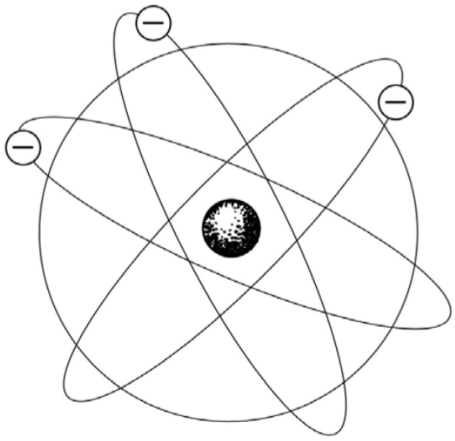
Thomson szczegółowo określił możliwe położenia elektronów w atomie: były one zależne od ich ilości oraz symetryczne względem środka kuli. Jeśli atom posiadał dwa elektrony, były one zdaniem Thomsona ułożone w linii prostej, przechodzącej przez środek kuli. W przypadku trzech układ przyjmował kształt trójkąta równobocznego. Większa ilość powodowała rozmieszczenie ich na wierzchołkach kolejnych wielokątów foremnych. Gdy liczba elektronów przekraczała osiem, układ miał przyjmować formę dwóch lub większej liczby wspomnianych wielokątów, ułożonych na wielu płaszczyznach. Ich punkt przecięcia był jednocześnie środkiem całego atomu [59].

Choć model Kelvina-Thomsona tłumaczył wiele zjawisk chemicznych i fizycznych, np. jonizację, jako wytrącanie poza obręb atomu jednego lub kilku elektronów oraz emisję światła - skutek drgań elektronów, jednak posiadał również wady. Główną z nich była trudność w wyjaśnieniu trwałości atomów, bowiem odpychające siły działające pomiędzy elektronami powinny zaburzać stan układu. Niewiadoma pozostawała także natura wiązań chemicznych oraz okresowość pierwiastków wraz z ich liniami spektralnymi. Z tego powodu model ten został podważony przez wyniki dalszych badań [59].

1.2.2.2 Model Rutherforda

Model Ernesta Rutherforda (1871-1937 r.), inaczej *model planetarny*, wynikał bezpośrednio z badań, jakie Rutherford prowadził wraz z dwoma studentami: Hansem Geigerem (1882-1945 r.) oraz Ernestem Marsdenem (1889-1970 r.). Szczegółowo zostaną one opisane w rozdziale 1.3.1. Ich wyniki wykazały brak zgodności pomiędzy założeniami modelu atomu Kelvina-Thomsona, prowadząc do jego ostatecznego obalenia.

W modelu Rutherforda całkowity ładunek dodatni atomu stanowić miało jądro atomowe, rozmiarach rzędu 10^{-15} m, znajdujące się w jego środku. Wokół jądra, na wzór orbit planetarnych, poruszały się elektrony. Niemal całe wnętrze atomu stanowiła pusta przestrzeń [60].



Rysunek 1.2: Model atomu Rutherforda [59].

Choć model Rutherforda dostarczył odpowiedzi na pytanie jakich rozmiarów jest jądro atomowe, jednak nie wyjaśniał w jaki sposób składniki atomu trzymają się razem oraz w jakiej dopuszczalnej odległości od jądra może znajdować się elektron. Nie odpowiadał również na pytanie dlaczego dwa atomy tego samego rodzaju emitują dokładnie takie same dyskretne linie widmowe. Był on przełomowy na gruncie fizyki klasycznej, prowadzącej na tym etapie badań w ślepy zaułek. Tym samym zwiastował potrzebę oparcia się na zupełnie nowych ideach dotyczących budowy atomu [59].

1.2.2.3 Model Bohra

Model atomu wodoru Bohra bazował na koncepcji Rutherforda, jednak jego przewaga wynikała z oparcia jej o warunki kwantowe, zwane również *postulatami Bohra*. Powstał on w 1913 r., okresie *starszej teorii kwantów*[2], dzięki badaniom duńskiego fizyka - Nielsa Bohra (1885-1962 r.).

W modelu atomu Rutherforda elektron mógł znajdować się w dowolnej odległości od jądra, co wynikało z zastosowania fizyki klasycznej w matematycznym opisie jego ruchu. Opierał się on na drugiej zasadzie dynamiki Newtona ($F = ma$) oraz uwzględnieniu działania siły Coulomba[3] między dodatnio naładowanym jądrem i ujemnie naładowanym elektronem. Fakt ten prowadził do sprzecznych z trwałością atomu konsekwencji; zgodnie z elektrodynamiką klasyczną elektron powinien tracić energię wraz z ruchem przyspieszonym po okręgu i tym samym spadać spiralnie na jądro. Promienowałby wówczas również ciągłe widmo elektromagnetyczne, co przeczy obserwowanym dyskretnym liniom widmowym danych pierwiastków [60].

Wobec powyższych sprzeczności Bohr wysunął koncepcję istnienia pewnych wyróżnionych stanów w atomie - *stanów stacjonarnych*, określających odległości elektronów od jądra, wartości energii i charakter emitowanego promieniowania. Wyraża się ona we wspomnianych postulatach, opisanych poniżej [59].

[2]Starsza teoria kwantów - teoria sformułowana w pierwszych dekadach XX wieku w odpowiedzi na nieudane próby zastosowania mechaniki klasycznej w celu wyjaśnienia zagadnienia promieniowania ciała doskonale czarnego. Z teorii tej rozwinęła się współczesna mechanika kwantowa. [60]

[3]Zgodnie z Prawem Coulomba, oddziaływanie elektrostatyczne pomiędzy dwoma ładunkami ma taki sam charakter, jak oddziaływanie grawitacyjne ciał obdarzonych masą, z tą jednak różnicą, że rozważamy ładunki elektryczne. Siła Coloumba dla oddziaływania elektrostatycznego przyjmuje następującą postać: $F = -k(q_1 q_2/R^2)$. [93]

- Ze wszystkich możliwych klasycznych orbit kołowych dozwolone są jedynie takie, na których wartość momentu pędu elektronu (iloczynu pędu $p = mv$ i promienia orbity R) jest całkowitą wielokrotnością $\hbar$ - stałej Plancka h podzielonej przez 2π ($\hbar = \frac{h}{2\pi}$):

$$mVr = \frac{nh}{2\pi} = n\hbar, \tag{1.1}$$

gdzie: m - masa elektronu, V - prędkość elektronu, r - promień orbity elektronu, n - główna liczba kwantowa [4], $h = 6,63 \cdot 10^{-34} Js$ - stała Plancka.

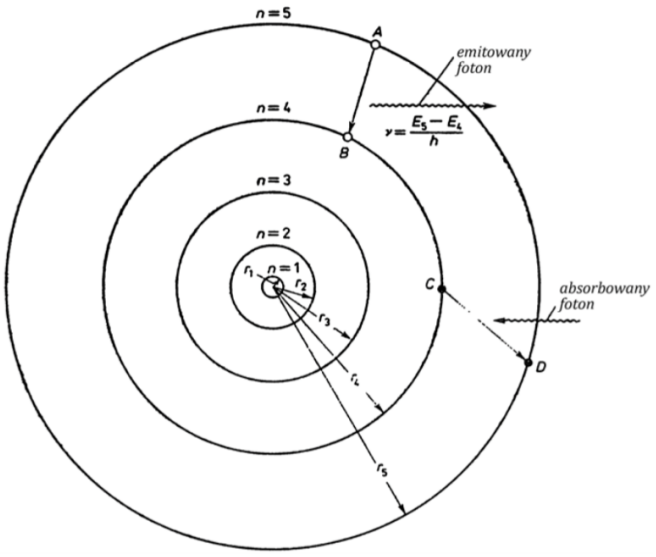
- Znajdujące się na orbitach stacjonarnych (dozwolonych) elektrony nie promieniują energii. Promienie orbit stacjonarnych mogą przybierać jedynie ściśle określone, dyskretne wartości, wobec czego określa się je mianem *skwantowanych.*

- Emisja lub absorpcja energii następuje tylko wówczas, gdy elektron przechodzi z jednej orbity stacjonarnej na drugą, a energia wypromieniowanego bądź pochłoniętego kwantu promieniowania elektromagnetycznego równa jest wartości bezwzględnej różnicy energii stanu końcowego E_k i początkowego E_p:

$$|E_k - E_p| = h\nu, \tag{1.2}$$

gdzie ν – częstotliwość wyemitowanej bądź pochłoniętej fali elektromagnetycznej.

Wszystkie z powyższych założeń były wówczas sprzeczne ze znaną, klasyczną teorią. Na ich podstawie można również wyznaczyć: prędkość elektronu na dowolnej orbicie stacjonarnej, wzór na promień dowolnej orbity elektronu w atomie wodoru, energię całkowitą elektronu w atomie wodoru, wzór na energię całkowitą elektronu znajdującego się na n-tej orbicie atomu wodoru oraz częstotliwość fali elektromagnetycznej wypromieniowywanej lub zaabsorbowanej podczas przeskoku elektronu z n-tej orbity na orbitę m-tą [40].

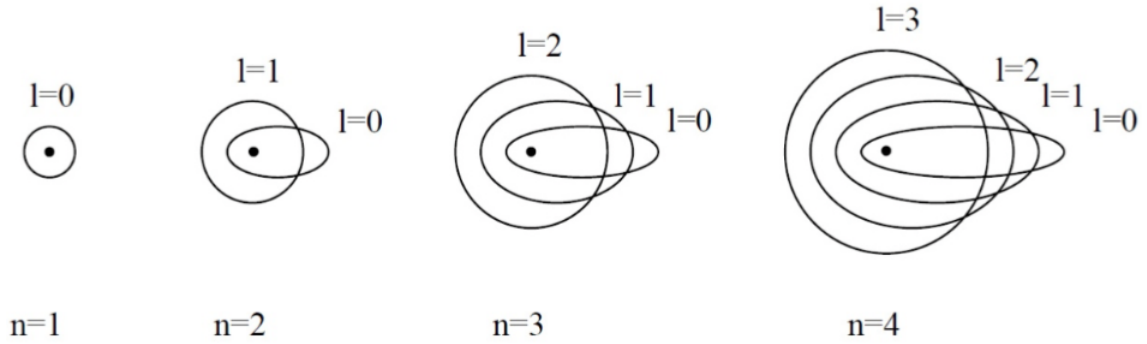


Rysunek 1.3: Model atomu Bohra [59].

[4] Główna liczba kwantowa - określa numer orbity stacjonarnej elektronów w atomie liczonej od orbity o najmniejszym promieniu [59]

Model atomu Bohra stanowił przełom na drodze do lepszego poznania mikroświata, będąc pierwszą teorią wykorzystującą fizykę kwantową. Wyjaśniał on trwałość atomów, dyskretne linie widmowe i budowę układu okresowego pierwiastków. Mimo to najlepsze ilościowo wyniki pozwalał uzyskać dla wodoru, z mniejszym powodzeniem w przypadku cięższych pierwiastków. Ponadto nie wprowadzał reguł zakazujących pewnych przejść pomiędzy różnymi stanami atomu.

Z czasem model Bohra doczekał się modyfikacji ze strony Arnolda Sommerfelda (1868-1951 r.), który sformułował teorię bazującą na ruchu elektronów po orbitach eliptycznych, których liczba była równa wartości głównej liczby kwantowej n. Jądro atomu znajdowało się w jednym z ognisk elipsy. Przyjął on również, że elektrony krążące bliżej jądra atomowego mają większą masę niż elektrony na orbicie położonej daleko od jądra, z powodu większej prędkości [60]. Teoria Sommerfelda wprowadziła także orbitalną liczbę kwantową l, która określała kształt (spłaszczenie) orbity. Przyjmowała ona wartości od 0 do $n-1$ i *„uwzględniała możliwość różnych wartości pędu elektronu przy tej samej energii całkowitej”* [103].



Rysunek 1.4: Teoria Sommerfelda - kształty orbit elektronu odpowiadające różnym liczbom kwantowym n i l [59].

1.2.2.4 Model współczesny

Współczesny model budowy atomu wynika przede wszystkim z koncepcji Louise'a de Broglie'a (1892-1987 r.), który wysunął przypuszczenie, że skoro fale elektromagnetyczne mogą przejawiać naturę korpuskularną, to również cząstki materii (np. elektrony), mogą przejawiać własności falowe [60]. Twierdził on, że każdej cząstce o pędzie p można przypisać falę materii o długości

$$\lambda = \frac{h}{p}, \tag{1.3}$$

gdzie: λ - długość fali, h - stała Plancka, p - pęd cząstki.

W przypadku elektronów długość fali maleje wraz ze wzrostem prędkości, wobec czego dla wielkości bliskich prędkości światła osiąga wartość rzędu 10^{-12} m [59]. Hipoteza de Broglie'a została udowodniona doświadczalnie przez Clintona Josepha Davissona (1881-1958 r.) i Lestera Halberta Germera (1896-1971 r.) w 1927 roku. Łączy się ona z modelem atomu Bohra, tłumacząc występowanie orbit stacjonarnych. Zakładając, że elektron w atomie przyjmuje postać fali stojącej, należy uznać, że długość orbity

stacjonarnej jest całkowitą wielokrotnością długości fali elektronów λ. W przeciwnym razie fale ulegałyby wygaszeniu w wyniku interferencji destruktywnej [60].

Uwzględniając idee de Broglie'a, Erwin Schrödinger przypisał własności falowe dowolnej cząstce, opisując jej ruch funkcją zespoloną współrzędnych przestrzennych i czasu - *funkcją falową* Ψ. Jej postać otrzymuje się rozwiązując równanie Schrödingera z 1926 r.:

$$-\frac{\hbar^2}{2m}\nabla^2\Psi + U\Psi = \frac{\partial\Psi}{\partial t}i\hbar, \tag{1.4}$$

gdzie: Ψ - funkcja falowa, m - masa elektronu, $\hbar$ - stała Plancka, $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$ - operator Laplace'a, t - czas, U - funkcja współrzędnych przestrzennych i czasu. Gradient U, wzięty ze znakiem minus, określa siłę działającą na cząstkę. Gdy funkcja U jest niezależna od czasu, ma ona charakter energii potencjalnej cząstki [59]. Z uwagi na symetrię atomu łatwiej jest rozpatrywać go we współrzędnych sferycznych, w których funkcja falowa Ψ zależna jest od promienia wodzącego r oraz zmiennych kątowych φ i θ. Wówczas przekształcenia równania Schrödingera dla elektronu w atomie wodoru umożliwiają otrzymanie *„niezależnych równań falowych, z których każde będzie zawierało funkcję tylko jednej współrzędnej sferycznego układu współrzędnych"* [44]:

$$\Psi_{nlm} = R_{nl}(r) \cdot \Theta_{lm_l}(\theta) \cdot \Phi_{m_l}(\varphi), \tag{1.5}$$

gdzie R, Θ, Φ oznaczają kolejno część radialną, biegunową i azymutalną, n - główna liczba kwantowa, l - orbitalna liczba kwantowa, m_l - magnetyczna liczba kwantowa, r, φ, θ - współrzędne sferyczne. Równanie Schrödingera można rozwiązać jedynie w nielicznych przypadkach, np. atomu jednoelektronowego lub wodoropodobnego. Bardziej złożone atomy i układy wymagają rozwiązań przybliżonych [59].

Fizyczny sens funkcji falowych wyjaśnił w 1926 r. Max Born (1882-1970 r.) twierdząc, że kwadrat wartości bezwzględnej funkcji falowej opisującej daną cząstkę w pewnym obszarze jest wprost proporcjonalny do prawdopodobieństwa znalezienia tej cząstki w tym obszarze [59]:

$$\Psi^*\Psi = |\Psi|^2 = p, \tag{1.6}$$

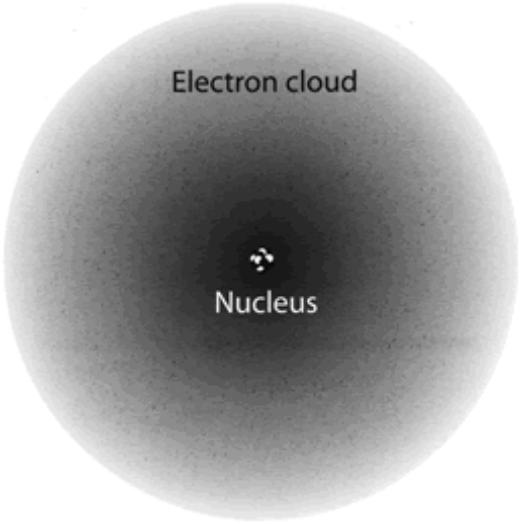
gdzie: p - prawdopodobieństwo znalezienia cząstki (w otoczeniu ustalonego punktu o współrzędnej x). Prawdopodobieństwo znalezienia cząstki w obszarze dx jest równe:

$$P(x, x + dx) = p \cdot dx = |\Psi|^2 \cdot dx. \tag{1.7}$$

Tak określona funkcja P nosi nazwę *funkcji gęstości prawdopodobieństwa* [43]. Postulat Borna wiąże zatem własności falowe i korpuskularne cząstek, umożliwiając określenie z pewnym prawdopodobieństwem miejsce znalezienia cząstki jako korpuskuły na podstawie falowego opisu jej stanu.

W 1927 roku Werner Heisenberg (1901-1976 r.) sformułował *zasadę nieoznaczoności*, zgodnie z którą, jeśli pęd cząstki zostanie dokładnie określony, równie dokładne wyznaczenie jej położenia nie będzie możliwe. Wynika to z własności świata - nie niedokładności pomiaru. Wobec tego przewidzenie dalszego ruchu cząstki traci sens, stąd standardowa wersja mechaniki kwantowej nie przypisuje elektronowi orbit w rozumieniu fizyki klasycznej.

W mechanice kwantowej atom opisuje się jako *„obiekt składający się z dodatnio naładowanego jądra atomowego, które skupia prawie całą masę atomu (...), i otaczającej je chmury elektronowej, którą tworzą różne fale stojące, otaczające jądro”*. [60].



Rysunek 1.5: Kwantowy model atomu - jądro otoczone chmurą elektronową [3].

Do opisu atomu stosuje się liczby kwantowe. Wyróżniamy:

- **Główną liczbę kwantową** n - określa ona numer powłoki, na której znajduje się elektron, a także energię powłoki elektronowej, daną równaniem:

$$E_n = \frac{-me^4}{32\pi^2\varepsilon_0^2\hbar^2 n^2}, \tag{1.8}$$

  gdzie ε_0 - przenikalność elektryczna próżni. Przyjmuje ona wartości kolejnych całkowitych liczb dodatnich. Stosuje się również oznaczenie literowe, w którym każdej kolejnej powłoce przypisana jest następna litera alfabetu, począwszy od litery K: $n = 1$(K), $n = 2$(L), $n = 3$(M) itd. [59];

- **Orbitalną liczbę kwantową** l - oznaczającą podpowłokę, na której znajduje się elektron. Przyjmuje ona wartości zera lub kolejnych całkowitych liczb dodatnich i określa niewielkie różnice energii elektronów danego poziomu energetycznego, związane z różnicami ich orbitalnego momentu pędu $L = \hbar\sqrt{l(l+1)}$, gdzie L - moment pędu elektronu [44]. Orbitalnej liczbie kwantowej także przypisuje się oznaczenia literowe, np.: $l = 0$ (s) itd.;

- **Magnetyczną liczbę kwantową** m_l - oznaczającą niewielkie różnice poziomu energetycznego elektronów przy tych samych liczbach kwantowych n i l w polu magnetycznym [59]. Przyjmuje ona wartości w zakresie $m_l \in [-l; l] \wedge m_l \in C$;

- **Magnetyczną spinową liczbę kwantową** m_s - określającą spin elektronu znajdującego się w spoczynku. Przyjmuje ona wartości $m_s = \pm\frac{1}{2}$. Spin s jest pojęciem czysto kwantowym, definiującym rzut całkowitego momentu magnetycznego na oś wyróżnioną (tzw. oś kwantyzacji). Spinowy moment pędu $L_s = \sqrt{s(s+1)}\hbar$ [59].

Niejednorodne pole magnetyczne powoduje rozszczepienie poziomów energetycznych w atomie. W przypadku jego braku, na energię elektronów wpływ mają wyłącznie dwie pierwsze liczby kwantowe. Ponadto orbitalny moment magnetyczny, względem np. kierunku pola magnetycznego (wyróżnionej osi) nie może ustawić się pod dowolnym kątem w przestrzeni - kąt biegunowy wektora całkowitego momentu pędu jest zdeterminowany przez warunek [59]:

$$cos\theta = \frac{L_z}{L} = \frac{m_l\hbar}{\hbar\sqrt{l(l+1)}} = \frac{m_l}{\sqrt{l(l+1)}}, \tag{1.9}$$

gdzie: L_z - składowa orbitalnego momentu pędu, L – orbitalny moment pędu, θ – kąt pomiędzy L_z a L, m_l – magnetyczna liczba kwantowa, l – orbitalna liczba kwantowa.

Ogólna liczba stanów kwantowych na powłoce n równa jest $2n^2$ [60]. Dwa elektrony nie mogą zajmować jednego stanu (nie mogą być opisane tym samym zestawem liczb kwantowych), co wynika z *Zakazu Pauliego*[5] (1900-1958 r.). Dzięki niemu elektrony nie mogą wspólnie zajmować stanu o najniższej energii, zatem nie 'spadają' na orbitę położoną najbliżej jądra [60]. Z uwagi na Zasadę Nieoznaczoności Heisenberga, pojęcie orbity elektronu nie może funkcjonować w obrębie fizyki kwantowej. Wobec tego do określania położenia elektronu we współczesnym modelu atomu stosuje się pojęcie *orbitalu*. Jest to przestrzenna prezentacja gęstości elektronowej ($\Psi^*\Psi$), obejmująca obszary przestrzeni, w których prawdopodobieństwo znalezienia elektronu wynosi powyżej 90% [59]. Gęstość elektronowa określona jest przez iloczyn prawdopodobieństwa radialnego $p_r = R^*_{nl}R_{nl}$ (gdzie: $R(r)$ - radialna część funkcji falowej) oraz prawdopodobieństwa biegunowego $p_\Theta = \Theta^*_{lm_m}\Theta_{lm_l}$ i wynosi:

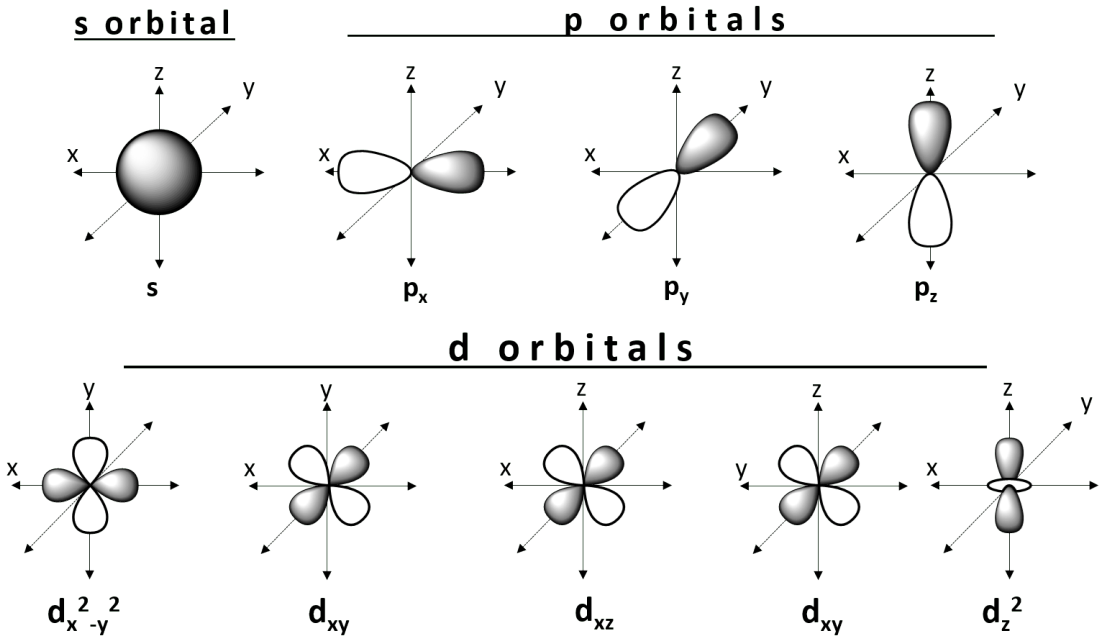
$$\Phi^*_{ml}\Phi_{ml} = (A^*e^{-im_l\varphi})(Ae^{im_l\varphi}) = A^*A = const. \tag{1.10}$$

Orbitale atomowe opisują elektrony przypisane do jądra atomowego, które nie uczestniczą w danym momencie w tworzeniu wiązań chemicznych. Ich kształt zależy jedynie od części biegunowej Θ_{im_l} funkcji falowej. Wobec tego wyróżniamy:

- Orbitale s o kształcie sferycznym;
- Orbitale p o kształcie hantli;
- Orbitale będące kombinacją powyższych.

Opisanej powyżej kształty orbitali zostały zwizualizowane na rysunku 1.6.

[5]Zakaz Pauliego - zasada, która dopuszcza obsadzenie stanu opisanego liczbami kwantowymi (n, l, m_l) nie więcej niż dwoma elektronami, którym można przypisać odpowiednio liczby $m_s = +\frac{1}{2}$ i $m_s = -\frac{1}{2}$. Zakaz Pauliego obejmuje swym działaniem wszystkie fermiony (cząstki o połówkowym spinie) [59]. Opsisuje również inne, bardziej ogólne reguły, np. fakt, że fermiony nie mogą posiadać tego samego położenia gdy mają taką samą wartość „orientacji spinu" jako cząstki swobodne.



Rysunek 1.6: Podstawowe kształty orbitali - s, p, d [33].

1.2.3 Wyznaczanie rozmiarów atomu

Współcześnie fizyka atomowa nie ogranicza się jedynie do teoretycznych spekulacji na temat budowy atomu. Rozwój nanotechnologii [6] doprowadził do znacznego postępu w badaniach doświadczalnych, umożliwiając manipulowanie pojedynczymi cząsteczkami i atomami. Dzięki nim pozyskać możemy coraz więcej rzeczywistych danych i informacji dotyczących składników materii [60].

Historia fizyki doświadczalnej jest równie długa, co opisana w niniejszym rozdziale historia rozwoju teorii i spekulacji na temat budowy atomu. Obecnie dysponujemy możliwością obrazowania postaci atomów przy pomocy skaningowej i wysokonapięciowej mikroskopii elektronowej. Skaningowy mikroskop tunelowy to jedno z głównych narzędzi obserwacji i manipulacji pojedynczych atomów[7]. Rozmiary atomu można natomiast wyznaczyć bazując na *przekroju czynnym na oddziaływanie* (1.2.3.2), z jakim atom uczestniczy w zderzeniach z innymi atomami [44].

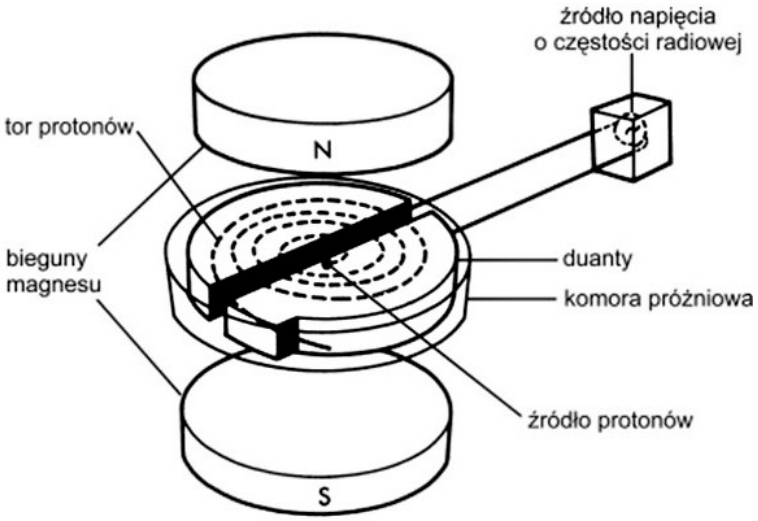
1.2.3.1 Zderzenia

Zderzenie definiuje się jako oddziaływanie ciał (dwóch lub więcej) siłami krótkozasięgowymi, w wyniku którego następuje zmiana czteropędów (czterowymiarych uogólnień wektora pędu) tychże ciał i/lub pojawiają się inne ciała w stanie końcowym [53].

[6] Nanotechnologia - dziedzina nauki i inżynierii materiałowej zajmująca się kontrolowanym wytwarzaniem nanostruktur i nanomateriałów oraz metodami służącymi do ich badania i modelowania [51]. Nano- — przedrostek w metrycznym systemie miar, oznaczający jedną miliardową jednostki podstawowej (10^{-9}), np. nanometr, nanosekunda.[60]

[7] Skaningowy mikroskop tunelowy (mikroskop elektronowy skaningowy) - rodzaj mikroskopu elektronowego, w którym wiązka elektronów, skupiona na powierzchni badanej próbki w plamkę o bardzo małej średnicy (do 0,1 nm), omiata wybrany prostokątny obszar powierzchni ruchem skanującym, linia po linii [50].

Kwadrat sumy czteropędów zderzanych cząstek to niezmiennik s, a zarazem masa niezmiennicza tego układu [78]. Zderzenia stanowią podstawowy sposób pozyskiwania informacji o materii w małych odległościach. Z tego względu fizyka doświadczalna niejednokrotnie bazuje na celowym zderzaniu przyspieszanych w akceleratorach cząstek.



Rysunek 1.7: Budowa cyklotronu - formy akceleratora cyklicznego cząstek obdarzonych ładunkiem elektrycznym [101].

Akcelerator umożliwia obserwację rezultatów zderzeń, dzięki zastosowaniu odpowiednich detektorów promieniowania jonizującego. Jego zasada działania jest prosta; naładowane cząstki poruszają się w próżniowym tunelu w polu magnetycznym. Siła magnetyczna powoduje zakrzywienie toru ruchu cząstki, działając prostopadle do jej prędkości, dzięki czemu porusza się ona po okręgu. Wówczas siła dośrodkowa równa jest sile magnetycznej:

$$\frac{mv^2}{r} = qvB, \tag{1.11}$$

gdzie m - masa cząstki, v - prędkość, q - ładunek, r - promień okręgu, B - wartość indukcji magnetycznej.

W pewnych odstępach czasu, z generatora wysokiej częstości przykładane jest zmienne pole elektryczne, powodujące przyspieszenie cząstek - aż do osiągnięcia pożądanych wielkości energii. Wówczas zderzane są one z tarczą lub inną wiązką cząstek [60]. W akceleratorach niejednokrtonie dochodzi do spadku częstości krążenia cząstek, a w konsekwencji utraty synchronizacji. Potocznie twierdzi się, że jest to spowodowane wzrostem masy cząstek. W rzeczywistości wspomniany „wzrost masy" wynika z kontrakcji Lorentza, efektywnie opisywanej poprzez zmianę masy. We wzorze na prędkość odcinek czasu ulega zmiane w związku z efektami Lorentzowskimi.

Problem synchronizacji rozwiązany został w synchrotronie - szczególnym typie akceleratora cyklicznego. Umożliwia on ciągłe utrzymywanie synchronizacji z krążącymi cząstkami dzięki zmianie pola magnetycznego B oraz częstotliwości oscylacji pola elektrycznego [48]. Synchrotron pozwala uzyskać energię elektronów do ok. 47 GeV i protonów do ok. 1 TeV [27]. Największym, funkcjonującym na świecie synchrotronem jest

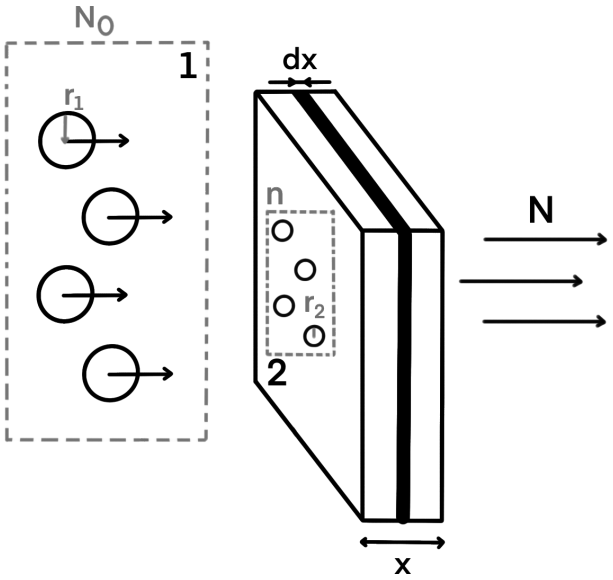
obecnie Wielki Zderzacz Hadronów (ang. *Large Hadron Collider* (LHC)) znajdujący się w Europejskim Ośrodku Badań Jądrowych CERN w pobliżu Genewy. Długość jego torusa mierzy 27 km [14].

Rodzaje zderzeń [53]:

- **Zderzenia elastyczne (sprężyste)** - występują, gdy po zderzeniu zachowana zostaje energia kinetyczka E_k oraz osobno energia potencjalna E_p cząstek;
- **Zderzenia nieelastyczne (niesprężyste) I rodzaju (endoenergetyczne)** - występują, gdy energia kinetyczna E_k cząstek po zderzeniu jest mniejsza od energii kinetycznej przed zderzeniem, przy jednoczesnym wzroście energii potencjalnej (w tym spoczynkowej). Mogą one przebiegać wyłącznie powyżej pewnej energii progowej;
- **Zderzenia nieelastyczne (niesprężyste) II rodzaju (egzoenergetyczne)** - występują, gdy energia kinetyczna E_k cząstek po zderzeniu jest większa od energii kinetycznej przed zderzeniem, kosztem energii potencjalnej (w tym spoczynkowej). Mogą one zachodzić dla wszystkich energii.

1.2.3.2 Przekrój czynny na oddziaływanie

Przekrój czynny na oddziaływanie (zwany również *przekrojem czynnym na rozpraszanie*) określa prawdopodobieństwo zajścia zderzenia i opisuje efektywne pole obszaru oddziaływania. Zwykle oznaczany jest on symbolem σ. Biorąc pod uwagę wiązkę atomów 1 (o powierzchni przekroju poprzecznego równej A, promieniu atomów r_1 oraz gęstości atomów N_0), uderzającą w warstwę 2 składającą się z atomów (o grubości $\triangle x$, promieniu r_2 i gęstości atomów n), określamy *jak wiele atomów* 1 *dozna zderzenia z atomami* 2 *i ulegnie odchyleniu od pierwotnego kierunku, nie przechodząc niezaburzenie przez warstwę* [44].



Rysunek 1.8: Przekrój czynny na oddziaływanie cząstek o promieniu r_1 z cząstkami o promieniu r_2 (źródło: opracowanie własne).

Odchylenie wskutek zderzenia atomów o promieniach r_1 i r_2 następuje wówczas, gdy zachodzi ono w obszarze o powierzchni $\sigma = (r_1 + r_2)^2\pi$. Łącząc odchylenia obu cząstek otrzymać można przekrój czynny, wyrażający się poniższym wzorem [44]:

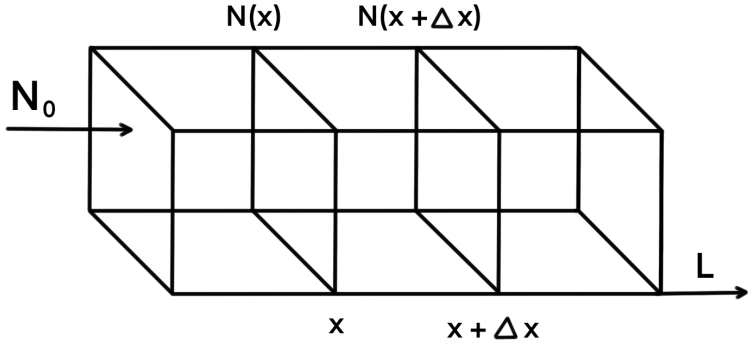
$$W = \frac{\text{powierzchnia wszystkich przekrojów czynnych na oddziaływanie w wiązce}}{\text{powierzchnia całkowita A}}. \tag{1.12}$$

Jednak założenie to jest prawdziwe jedynie wtedy, gdy powierzchnie πr^2 cząstek nie przekrywają się. Aby obliczyć liczbę atomów odchylanych w warstwie o grubości skończonej L należy ją uprzednio podzielić na cieńsze warstwy o grubości $\triangle x$. Wówczas:

$$\triangle N = -WN = -\frac{\text{całkowita liczba atomów w objętości} \cdot \sigma}{\text{całkowita powierzchnia}} \cdot N, \tag{1.13}$$

gdzie całkowita liczba atomów w danej jednostce objętości równa jest iloczynowi gęstości cząstek w warstwie n, powierzchni A i grubości $\triangle x$, wobec tego:

$$\triangle N = -\frac{nA \triangle x\sigma}{A} N. \tag{1.14}$$



Rysunek 1.9: Podział warstwy o grubości L, przez którą przechodzą cząstki N_0, na cieńsze warstwy o grubości $\triangle x$ (źródło: opracowanie własne na podstawie [44]).

Dla nieskończenie małych przyrostów:

$$\frac{dN}{N} = -n\sigma dx. \tag{1.15}$$

Po scałkowaniu otrzymujemy liczbę odchylonych atomów po przebyciu odcinka x:

$$lnN = -n\sigma x + lnN_0, \tag{1.16}$$

gdzie lnN_0 jest stałą całkowania, N_0 natomiast liczbą cząstek padających w punkcie $x = 0$. Liczba cząstek nie ulegających odchyleniu po przejściu odległości x wynosi:

$$N = N_0 e^{-n\sigma x}. \tag{1.17}$$

Natomiast po przejściu całkowitej odległości L:

$$N = N_0 e^{-n\sigma L}. \tag{1.18}$$

Liczba atomów rozproszonych jest wówczas równa:

$$\boxed{N_{roz} = N_0(1 - e^{-n\sigma L}),} \tag{1.19}$$

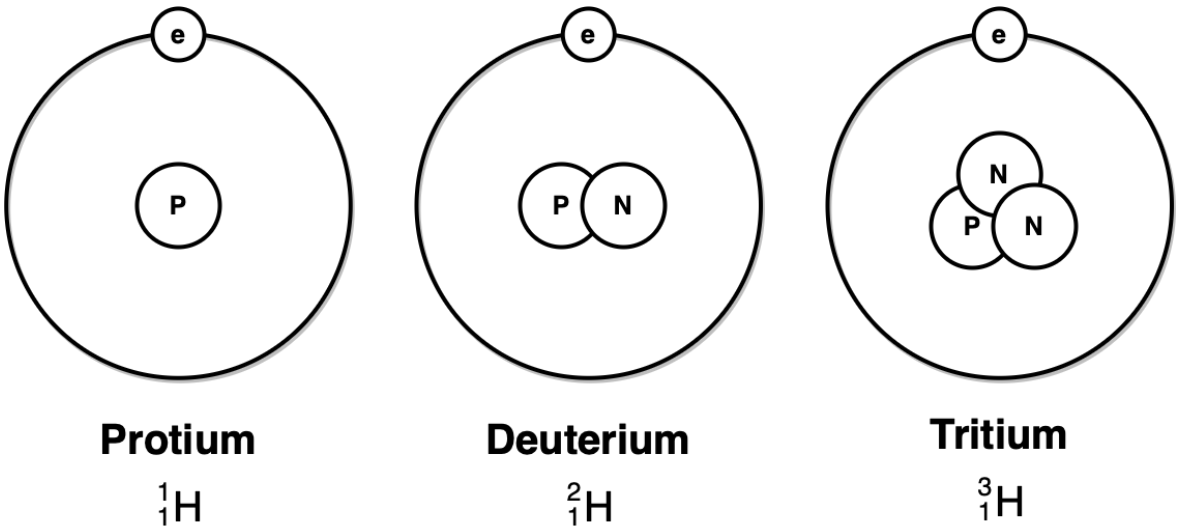
gdzie: $n\sigma = a$ - makroskopowy współczynnik rozpraszania, σ - mikroskopowy, całkowity przekrój czynny na oddziaływanie.

σ równa jest $(r_1 + r_2)^2\pi$, wobec czego z jej pomiaru wynika wartość sumy $r_1 + r_2$. Gdy atomy są jednakowe ($r = r_1 = r_2$), wówczas w ten sposób określić można wielkość r - rozmiar atomu [44].

1.3 Jądro atomowe

Odkrycie jądra atomowego zrewolucjonizowało znany wówczas obraz fizyki atomowej, obalając tym samym model atomu Thomsona (1.2.2.1), oparty na odpowiednim rozmieszczeniu ładunku elektrycznego. Odzwierciedlał on jakościowo obserwacje doświadczalne, ale nie był ilościowo zgodny z wynikami pomiarów [82].

Wiemy obecnie, że ładunek jąder, przypisywany protonom, jest wielokrotnością ładunku elementarnego (ze znakiem plus). Najlżejszym jądrem jest jądro atomu wodoru, zawierające jedynie jeden proton. Jednak z porównania masy i ładunku jąder ciężkich wynika, że ich masa jest około dwukrotnie większa niż ta, o której świadczy liczba protonów w jądrze, co spowodowane jest istnieniem neutronów, potwierdzonym doświadczalnie w 1932 roku przez J. Chadwick'a (1891-1974 r.). Protony i neutrony nazywane są wspólnie *nukleonami*. Jądro atomowe składa się z $Z + N = A$ nukleonów, gdzie Z - liczba atomowa, A - liczba masowa (1.2.1). Układy nukleonów o różnych liczbach Z i A noszą nazwę nuklidów [81].



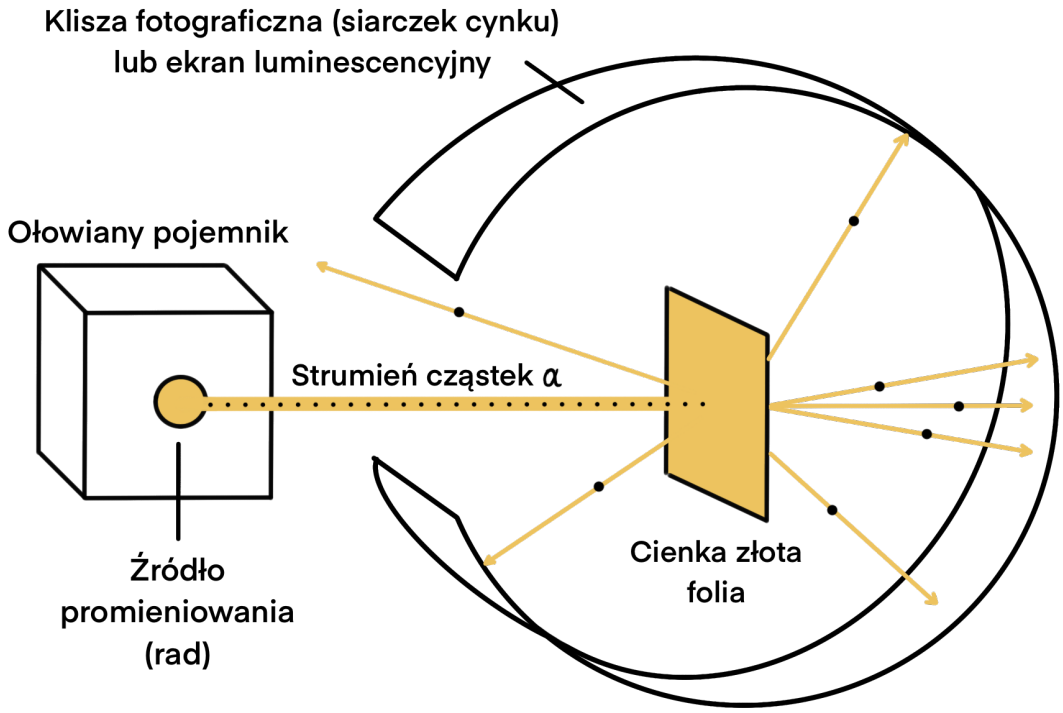
Rysunek 1.10: Izotopy wodoru - nuklidy o tej samej liczbie atomowej, a różnej masowej (źródło: opracowanie własne).

1.3.1 Rozpraszanie Rutherforda

Słynne doświadczenie Rutherforda, które umożliwiło odkrycie jądra atomowego, jest obecnie uważane za jeden z najpiękniejszych eksperymentów fizyki [18]. Pierwotnie jednak nie miało ono tego na celu. Rutherford wraz ze współpracownikami (Ernestem Marsdenem oraz Hansem Geigerem) wykonywali wówczas wiele podobnych eksperymentów, chcąc zbadać właściwości cząstek alfa na podstawie ich interakcji z materią. Przepuszczali je zatem przez różnego rodzaju substancje, w tym również folie [23].

Cząstki α składają się z podwójnie zjonizowanych jąder helu ${}^4_2He^{2+}$. Posiadają one zdolność jonizowania powietrza, dzięki czemu można je wykryć np. w komorze Wilsona. Wskutek procesów jonizacji oraz wzbudzeń cząsteczek powietrza, tracą one swoją początkową energię kinetyczną. Cząstki α posiadają zdolność przenikania przez tysiące atomów bez zauważalnego odchylenia ich prostoliniowego toru ruchu. Jedynie po utracie większości energii kinetycznej, pod koniec toru, ulegają one odchyleniu [44].

Rutherford badał właściwości cząstek alfa, również z wykorzystaniem złotej folii, jeszcze przed dokonaniem przełomowego odkrycia. Mierzył on niewielkie kąty rozpraszania cząstek po zderzeniu, z których większość uderzała w ekran na przeciwko folii. W poźniejszym czasie, współpracując z Marsdenem w projekcie badawczym, zlecił mu sprawdzenie, czy którekolwiek z cząstek alfa ulegają rozproszeniu wstecz, co wydawało się wówczas zupełnie niedorzeczne. Sam Marsden uznał to za sprawdzian jego umiejętności eksperymentalnych i wykonał pomiary zgodnie z poleceniem Rutherforda [15]. Układ składał się ze źródła promieniowania (naturalnego materiału promieniotwórczego), kolimatora oraz cienkiej, złotej folii. Cząstki α, emitowane ze źródła, przechodziły przez kolimator, uderzając następnie w złotą folię. Natężenie wiązki można było określić przy pomocy ekranu scyntylacyjnego, obserwowanego przez soczewkę [44].



Rysunek 1.11: Eksperyment Rutherforda
(źródło: opracowanie własne na podstawie [79]).

Ku zaskoczeniu Marsdena, błyski będące oznaką zderzenia się cząstek α z ekranem scyntylacyjnym, zaczęły się pojawiać pod bardzo dużymi kątami. Średnio jedna z kilku tysięcy cząstek ulegała rozproszeniu pod kątem większym niż 90^0, co przeczyło obowiązującemu wówczas modelowi Thomsona. Bazując na nim, cząstki powinny rozpraszać się jedynie pod bardzo małymi kątami, w sposób w jaki obserwował je Rutherford. Badania Marsdena przeczyły tej teorii [15]. Ponadto natężenie wiązki silnie malało wraz ze wzrostem kąta rozpraszania [44].

W 1911 r. Rutherford ogłosił teorię, mająca wytłumaczyć zaskakujące wyniki doświadczenia. Twierdził on, że cząstki α ulegały rozproszeniu z powodu dużej ilości dodatniego ładunku, skoncentrowanego w niewielkiej przestrzeni w środku atomu złota. Tak powstał *model planetarny* Rutherforda (rys. 1.2), wedle którego elektrony krążyć miały wokół jądra atomowego na wzór planet wokół Słońca [15].
Ilościowe wyjaśnienie wyników doświadczenia w oparciu o model planetarny [44]:

- Wielkość promienia R jądra atomowego jest rzędu 10^{-12} cm. Jądro skupia w sobie niemalże całą masę atomową i ma dodatni ładunek Ze, gdzie Z jest numerem miejsca pierwiastka w tablicy układu okresowego.

- Zderzenie cząstki α z dużo lżejszym elektronem atomu nie prowadzi do mierzalnego odchylenia toru cząstki alfa.

- Wokół dodatnio naładowanego jądra panuje pole kolumbowskie o natężeniu (w odległości r) danym wzorem:

$$E = \frac{1}{4\pi\varepsilon_0}\frac{Ze}{r^2}\frac{\mathbf{r}}{r}. \tag{1.20}$$

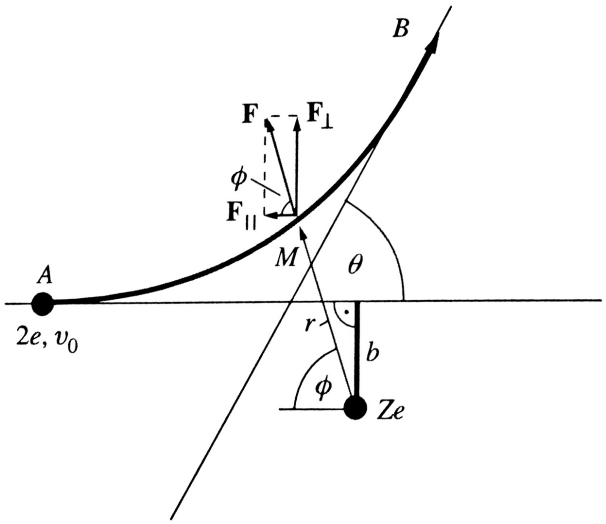
  Siła kolumbowska pomiędzy cząstką a jądrem wynosi:

$$\vec{F} = \frac{2Ze^2}{4\pi\varepsilon_0 r^2}\frac{\vec{r}}{r}, \tag{1.21}$$

  gdzie: Ze - ładunek jądra, e - ładunek elementarny, ε_0 - przenikalność elektryczna próżni, r - odległość pomiędzy jądrem a cząstką α.

1.3.2 Wyprowadzenie wzoru Rutherforda

W poniższym rozdziale wyprowadzony zostanie wzór Rutherforda na rozpraszanie, określający różniczkowy przekrój czynny na elastyczne rozpraszanie cząstek α na spoczywających jądrach atomowych [28].



Rysunek 1.12: Odchylenie cząstki α od A do B w wyniku rozpraszania przez jądro [44].

Związek pomiędzy kątem rozpraszania θ a parametrem zderzenia b wyznaczyć można wiedząc, że cząstka o prędkości v zbliża się do punktu A (wciąż odległego od jądra), gdzie ulega odchyleniu wkutek odpychającej siły kolumbowskiej F (1.21). Gdyby nie ulegała odchyleniu, przeszłaby w odległości b od jądra. W punkcie M na cząstkę działa siła, wyrażającą się dwiema składowymi - prostopadłą (1.22) oraz równoległą (1.23) do pierwotnego kierunku ruchu [44]:

$$F_\perp = Fsin\phi, \qquad (1.22) \qquad\qquad F_\parallel = Fcos\phi, \qquad (1.23)$$

gdzie ϕ - kąt pomiędzy kierunkiem wiązki padającej a promieniem wodzącym r chwilowego położenia cząstki.

Umieszczając początek układu współrzędnych w środku jądra atomowego, zastosować można prawo zachowania momentu pędu. Dla siły radialnej (1.21) moment pędu jest stały, a momenty pędu w punktach A i M są takie same: $(mv_0b)_A = (mr^2\phi)_M$. Korzystamy ze współrzędnych biegunowych (r, ϕ) otrzymując:

$$\frac{1}{r^2} = \frac{\phi}{v_0 b}. \qquad (1.24)$$

Dla ruchu w kierunku prostopadłym do pierwotnego kierunku wiązki, równanie ruchu Newtona wyraża się następującym wzorem:

$$m\frac{dv_\perp}{dt} = F_\perp = \frac{2Ze^2}{4\pi\varepsilon_0}\frac{1}{r^2}\sin\phi. \qquad (1.25)$$

Korzystając z równania 1.24, wprowadzając $k = 2Ze^2/4\pi\varepsilon_0$ i całkując po czasie otrzymujemy:

$$\int_{t_A}^{t_B} \frac{dv_\perp}{dt} dt = \frac{k}{mv_0 b} \int_B^A \sin\phi \frac{d\phi}{dt} dt. \tag{1.26}$$

$v_\perp = 0$ oraz $\phi = 0$, ponieważ w nieskończonej odległości punktu A od jądra, przyjętej dla ustalenia granic całkowania, siła kolumbowska jest zaniedbywalna [44]. Punktowi B pozwalamy oddalić się w nieskończoność, dzięki czemu widać, że $\phi = 180^0 - \theta$, gdzie θ - kąt rozpraszania. Wobec tego $v_\perp = v_0 sin\theta$. Prędkość końcowa w punkcie B równa jest prędkości początkowej w punkcie A, co wynika z prawa zachowania energii oraz faktu, że energia potencjalna zanika w dostatecznie dużej odległości od jądra. Korzystając z zależności [44]:

$$\frac{dv_\perp}{dt} dt = dv_\perp, \tag{1.27}$$

$$\frac{d\phi}{dt} dt = d\phi, \tag{1.28}$$

sprowadzamy równanie całkowe do postaci:

$$\int_0^{v_0 sin\theta} dv_\perp = \frac{k}{mv_0 b} \int_0^{\pi - 0} \sin\phi d\phi, \tag{1.29}$$

otrzymując:

$$v_0 \sin\theta = \frac{k}{mv_0 b}(1 + \cos\theta). \tag{1.30}$$

Ponieważ $1 + \cos\theta / \sin\theta = \cot(\theta/2)$, zatem związek między parametrem zderzenia a kątem odchylenia wynosi:

$$b = \frac{k}{mv_0^2} = \mathrm{ctg}(\theta/2). \tag{1.31}$$

Różniczkując, otrzymać można związek pomiędzy db oraz $d\theta$:

$$db = -\frac{k}{2mv_0^2} \frac{1}{\sin^2(\theta/2)} d\theta. \tag{1.32}$$

Z uwagi na symetrię obrotową wokół osi przechodzącej przez jądro tarczy i równoległej do kierunku wiązki padającej, należy rozważyć pierścień o promieniach $r_1 = b$ i $r_2 = b + db$. Przechodząca przez niego wiązka ulega rozproszeniu w obszar kątowy od $\theta - |d\theta|$ do θ, odpowiadający różniczkowemu przekrojowi czynnemu (1.2.3.2) $da = 2\pi b db$ [44]. Efektywna powierzchnia wszystkich atomów wynosi $dA = 2\pi b N D A db$ (zakładając, że powierzchnie atomów nie przekrywają się), gdzie: D - grubość folii, A - pole powierzchni folii, N - liczba atomów w folii na cm^3. Wówczas prawdopodobieństwo, że padająca cząstka α trafi w efektywną powierzchnię atomu w folii równe jest:

$$W = \frac{\text{efektywna powierzchnia}}{\text{powierzchnia całkowita}} = \frac{dA}{A} = 2\pi N D b db. \tag{1.33}$$

Liczba cząstek trafiających w efektywną powierzchnię wynosi: $dn' = n \cdot 2\pi N D b db$. Przechodzą one przez jednostkową sferę wokół folii tarczy na pierścieniu o powierzchni $d\Omega^{(1)} = 4\pi \sin(\theta/2)\cos(\theta/2)|d\theta|$. Z uwagi na fakt, że detektor stosowany w pomiarach

wycina jedynie fragment powierzchni $d\Omega^{(1)}$, zwany *kątem bryłowym*, liczba rzeczywiście mierzonych cząstek jest mniejsza od dn' o czynnik $d\Omega/d\Omega^{(1)}$. Zatem liczba cząstek obserwowanych pod kątem θ wynosi:

$$dn = dn' \frac{d\Omega}{d\Omega^{(1)}}. \tag{1.34}$$

Podstawiając wzory 1.31 i 1.32 otrzymać można pełny wzór Rutherforda [44]:

$$\boxed{\frac{dn(\theta, d\theta)}{n} = \frac{Z^2 e^2 DN}{(4\pi\varepsilon_0)^2 m^2 v_0^4 \sin^4(\theta/2)} d\Omega^{(1)},} \tag{1.35}$$

gdzie: n - liczba padających cząstek, dn - liczba cząstek rozproszonych pod kątem θ w kącie bryłowym $d\Omega$, Z - liczba atomowa, e - ładunek elementarny, D - grubość folii, N - liczba atomów w folii, ε_0 - przenikalność elektryczna próżni, m - masa rozpraszanych cząstek α, v_0 - prędkość padających cząstek. Odpowiadający powyższemu wzorowi różniczkowy przekrój czynny wynosi:

$$d\sigma = \frac{Z^2 e^4}{(4\pi\varepsilon_0)^2 m^2 v_0^4 \sin^4(\theta/2)}. \tag{1.36}$$

1.3.3 Promień jądra atomowego

Jak wynika z powyższych rozważań, cząstka α, podczas zbliżania się w stronę jądra atomowego, początkowo poddana jest działaniu odpychającego potencjału kolumbowskiego, a następnie przyciągającej siły jądrowej (w dostatecznie bliskiej odległości od jądra) [44]. *Promień jądra* określa odległość, w której powyższe potencjały (kolumbowski i jądrowy) osiągają porównywalne wartości. Dla jąder o liczbie masowej A promień wynosi: $R = (1.3 \pm 0.1)A^{1/3} \cdot 10^{-15}$ [m]. Jest to rozmiar rzędu femtometrów ([fm]). W celu dotarcia dostatecznie blisko do jądra, cząstka α musi charakteryzować się wysoką energią kinetyczną.

1.3.4 Wyniki doświadczalne

Wzór Rutherforda w bardzo dokładny sposób odzwierciedla rzeczywiste obserwacje, co sprawdzono doświadczalnie. Z testów wynika, że [44]:

- Prawo Coulomba jest dobrze spełniane - nawet dla niewielkich parametrów zderzenia, dla których wzór Rutherforda wciąż jest słuszny. Promień jądra jest $R < 6 \cdot 10^{-15}$ [m].
- Na podstawie eksperymentów z foliami różnych materiałów wyznaczyć można ładunek jądra Ze.
- Dla bardzo szybkich cząstek α (E>6 MeV) pod dużymi kątami θ następuje *anomalne rozpraszanie Rutherforda*, w którym obserwuje się wyraźne odstępstwa od wzoru Rutherforda, tzn. odstępstwa od prawa Coluomba. Cząstki docierają na tyle blisko jądra, że doznają krótkozasięgowych oddziaływań. Ze wzoru Rutherforda wynika wobec tego, że gęstość jądra jest większa niż gęstość atomu w ogólności. Podobnie dla bardzo dużych parametrów zderzenia wzór nie jest już spełniany - potencjał kolumbowski jest zaburzany przez elektrony atomu.

- Minimalna odległość, którą można uznać za miarę promienia jądra, związana jest z energią kinetyczną cząstki wyrażeniem:

$$R = \frac{Ze^2}{2\pi\varepsilon_0 E_k}. \tag{1.37}$$

- Ujemnie naładowane elektrony krążą wokół dodatnio naładowanego jądra, wobec czego panuje równowaga dynamiczna i stabilność układu zostaje zachowana.

- Jądro atomowe skupia niemal całą masę atomu, ponieważ masa elektronów jest znacznie mniejsza w porównaniu z masą protonów lub neutronów.

- Odchylenie cząstek α nie może być większe niż 28", co wynika ze stosunku masy elektronu do masy cząstek α. Gdy ulegają one wzajemnemu zderzeniu, wówczas jedynie niewielka część pędu może zostać przeniesiona z uwagi na małą masę elektronu.

Rozdział 2

Podstawy informatyczne

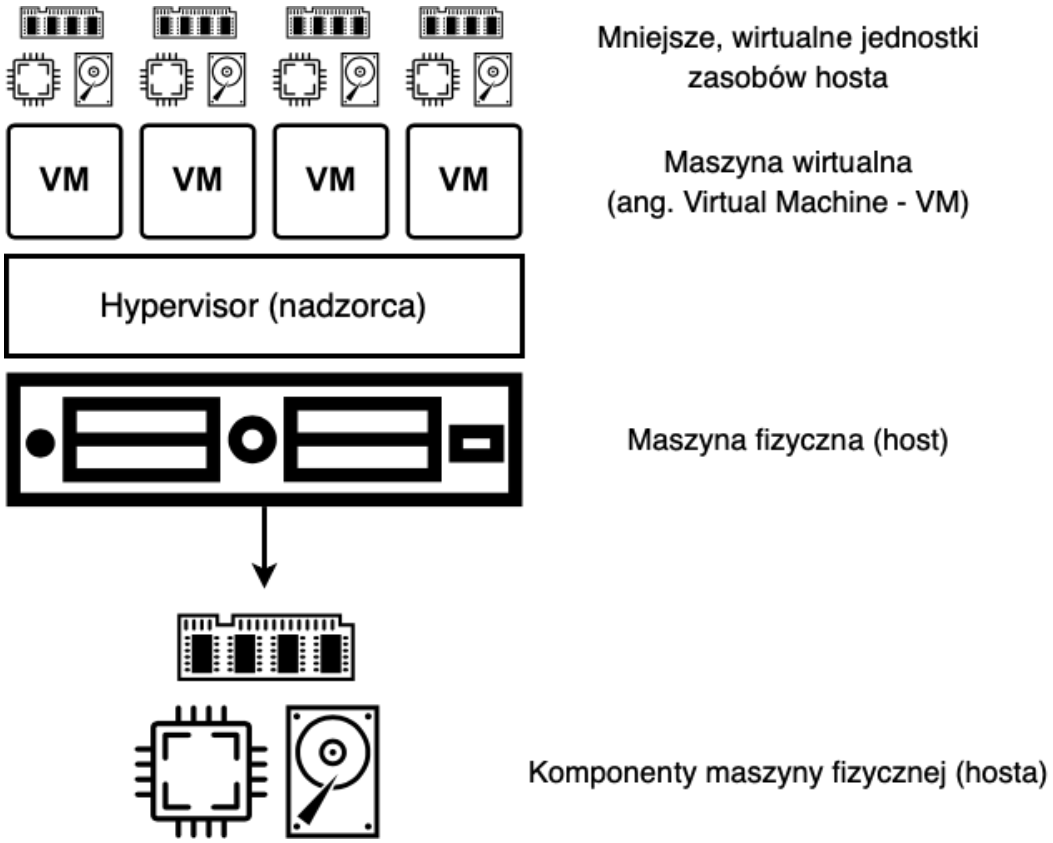
2.1 Wstęp - rozwój wirtualizacji

Wirtualizację definiuje się jako programowe odwzorowanie pewnych zasobów fizycznych komputerów, umożliwiające realizację co najmniej tych samych funkcjonalności. Wirtualizować można komponenty takie jak procesor, pamięć, zasoby dyskowe, jak również serwery, elementy sieci, aplikacje i systemy operacyjne [42]. Wyróżnia się: *parawirtualizację*, w której system wirtualizowany współpracuje ze środowiskiem maszyny - gospodarza (ang. *host*) oraz *wirualizację pełną*, w której wirtualizowany system operacyjny „odnosi wrażenie", że działa na maszynie fizycznej [92].

Wirtualizacja możliwa jest dzięki wykorzystaniu hipernadzorcy (ang. *hypervisor*). Jest to specjalnie przygotowany system operacyjny, który umożliwia tworzenie, konfigurację i zarządzanie maszynami wirtualnymi (ang. *Virtual Machines* (*VM*)). Pośredniczy on w komunikacji i przydzielaniu zasobów fizycznych (takich jak procesor, pamięć RAM, zasoby dyskowe) hosta, jednocześnie umożliwiając ich podział na mniejsze, wirtualne jednostki. Wyróżnia się dwa typy hypervisora [42]:

- **hypervisor typu I** - instalowany jest bezpośrednio na maszynie, zazwyczaj serwerze. Każda maszyna wirtualna stanowi wówczas osobny byt (partycję), której hypervisor przydziela zasoby sprzętowe bez pośrednictwa systemu operacyjnego. Cechuje się dużą wydajnością i niewielkimi opóźnieniami pracy maszyn wirtualnych względem hosta;
- **hypervisor typu II** - działa w systemie operacyjnym hosta i za jego pośrednictwem przydziela zasoby maszynom wirtualnym. Zazwyczaj uruchamia jeden dedykowany system operacyjny lub środowiska testowe.

Maszyny wirtualne mogą cechować się różną konfiguracją i obsługiwać różne systemy operacyjne. Ponadto wirtualne środowisko daje możliwość dowolnego przenoszenia maszyn pomiędzy hostami, zapewnia wysoką dostępność, replikację oraz tworzenie kopii zapasowych [42].



Rysunek 2.1: Graficzne przedstawienie wirtualizacji (źródło: opracowanie własne z wykorzystaniem programu draw.io).

Choć prawdziwy potencjał wirtualizacji rozwinął się w latach dziewięćdziesiątych ubiegłego stulecia, a obecnie jest ona szeroko wykorzystywanym narzędziem, jednak powstała znacznie wcześniej. Pojęcie *pamięci wirtualnej* zaczęło funkcjonować już w latach pięćdziesiątych dwudziestego wieku. Wówczas na Uniwersytecie w Manchesterze stworzono system Atlas, który umożliwiał wykonywanie 200 tys. operacji zmiennoprzecinkowych na sekundę. Jego działanie oparto na prototypie tejże pamięci. W późniejszym czasie do rozwoju wirtualizacji przyczynił się także Massachusetts Institute of Technology, tworząc *Compatible Time-Sharing System* (CTSS). Umożliwiał on korzystanie z komputera przez kilku użytkowników jednocześnie dzięki współdzieleniu czasu procesora. Popularny po dziś dzień program - nadzorca odpowiadał za przydział i kontrolę zasobów fizycznych [98]. Do dynamicznego rozwoju wirtualizacji oraz hypervisorów przyczynił się przede wszystkim IBM (International Business Machines Corporation) oraz Intel Corporation. W 1985 r. IBM wydał hypervisora typu I, Intel natomiast wprowadził w swoich procesorach tryb wirtualny. Już przed rokiem 2000 rozpoczęła się sprzedaż wirtualnych oprogramowań, a z czasem zaczęły one coraz bardziej zyskiwać na popularności [92].

Wirtualizacja stała się szczególnie użyteczna, gdy możliwości pojedynczych serwerów i komputerów zaczęły znacznie odbiegać od wymagań pojedynczych systemów operacyjnych lub aplikacji. Umożliwiła ona pracę na wielu systemach operacyjnych przy wykorzystaniu jednej fizycznej maszyny. Wymagało to mniej miejsca i funduszy przeznaczonych na utrzymywanie maszyn oraz było bardziej ekonomiczne [42]

2.2 Chmura obliczeniowa

2.2.1 Definicja chmury obliczeniowej

Chmura obliczeniowa (ang. *cloud computing*) jest obecnie jednym z najbardziej popularnych rozwiązań wykorzystywanych w świecie IT. W dużej mierze opiera się ona na wirtualizacji, jednak błędnie jest z nią w pełni utożsamiana. Przede wszystkim wirtualizacja to jedynie narzędzie, podczas gdy chmura definiuje rodzaj udostępnianych na żądanie usług. Mogą to być zarówno maszyny wirtualne, jak również kontenery oraz pojedyncze komponenty, np. magazyny obiektów, bazy danych lub elementy sieci. Wirtualizacja natomiast abstrahuje zasoby fizycznej maszyny, tworząc maszyny wirtualne, które następnie mogą stanowić moc obliczeniową udostępnianą w chmurze. Chmura łączy zasoby wirtualne i automatyzuje ich wykorzystanie, udostępniając je odpłatnie na żądanie (ang. *on-demand*) [100].

Poniżej przytoczona została oficjalna definicja chmury obliczeniowej, pochodząca z Narodowego Instytutu Standaryzacji i Technologii:

„Chmura obliczeniowa to model umożliwiający powszechny i wygodny dostęp na żądanie przez sieć do współdzielonej puli konfigurowalnych zasobów obliczeniowych (sieci, maszyn wirtualnych, przestrzeni dyskowej, aplikacji i usług), które mogą być szybko przydzielane i zwalniane przy minimalnym poziomie zarządzania czy też interakcji ze strony dostawcy usług." [94]

Ideą rozwiązań chmurowych jest zatem zakup *mocy obliczeniowej* zamiast fizycznej maszyny. Koszty obejmują czas pracy wykonanej na zakupionej na żądanie maszynie wirtualnej, której fizyczny host należy do innej osoby (zazwyczaj firmy, tzw. *dostawcy chmury* (ang. *cloud provider*)). Aktualizacja tejże maszyny, jej konserwacja oraz skalowanie nie należą wówczas do obowiązków osoby korzystającej z niej na żądanie. Wynajęta moc obliczeniowa może zostać wykorzystana do uruchomienia własnego oprogramowania lub tak zwanych *usług chmurowych* (ang. *cloud services*) - oprogramowań oferowanych przed dostawców chmur [4]. Dzięki temu możliwe jest stworzenie infrastruktury (definicja infrastruktury została opisana w rozdziale 2.4.1) w pełni działającej w chmurze.

Książka „*Chmura obliczeniowa. Rozwiązania dla biznesu*" autorstwa Jothy Rosenberg i Arthura Mateos [87] wyróżnia pięć podstawowych zasad definiujących przetwarzanie w chmurze, przytoczonych poniżej:

- *Pula zasobów obliczeniowych dostępnych dla każdego zarejestrowanego użytkownika.*
- *Wirtualizacja zasobów obliczeniowych w celu maksymalizacji wykorzystania sprzętu.*
- *Elastyczne skalowanie w górę lub w dół, w zależności od potrzeb (bez wydatków inwestycyjnych).*
- *Automatyczne tworzenie nowych wirtualnych maszyn oraz usuwanie maszyn istniejących (bez konieczności ingerencji ze strony użytkownika).*
- *Naliczanie opłat jedynie za wykorzystane zasoby.*

2.2.2 Rodzaje chmury obliczeniowej

Chmury klasyfikować można na dwa sposoby: względem oferowanych na żądanie usług (2.2.3) oraz rodzajów dostępności, przedstawionych poniżej.

2.2.2.1 Chmura prywatna

Chmura prywatna (zwana także chmurą „wewnętrzną" lub „korporacyjną") obejmuje infrastrukturę techniczną jednej organizacji [102]. Wewnętrzne, zwirtualizowane zasoby firmy dostępne są wyłącznie dla jej pracowników [87]. Zapewnia ona firmom korzyści chmury publicznej (skalowalność, elastyczność, samoobsługę), jednocześnie oferując dodatkową możliwość kontroli zasobów za pośrednictwem lokalnej infrastruktury obliczeniowej. Jest także wyposażona w dodatkowe systemy zabezpieczeń przepływu danych, np. zapory.

Wadą chmury prywatnej są koszty utrzymywania infrastruktury oraz personelu odpowiedzialnego za jej konserwację. Pod tym względem nie różni się ona od posiadania tradycyjnego centrum danych [66]. Znane obecnie, publiczne chmury dostawców (ang. *cloud providers*) takich jak Amazon, Microsoft lub Google używane były niegdyś jako chmury prywatne tychże firm [87].

2.2.2.2 Chmura publiczna

Chmura publiczna jest usługą obliczeniową oferowaną przez dostawców, udostępnioną za pośrednictwem Internetu każdej zainteresowanej osobie [67]. Są one bezpłatne lub sprzedawane na żądanie (klienci płacą wyłącznie za wykorzystane zasoby, np. zużycie cykli procesora CPU). Dzięki temu chmura publiczna jest znacznie tańsza w utrzymaniu niż chmura prywatna. Zakup i utrzymanie lokalnych maszyn oraz zarządzanie nimi należą do obowiązków dostawcy.

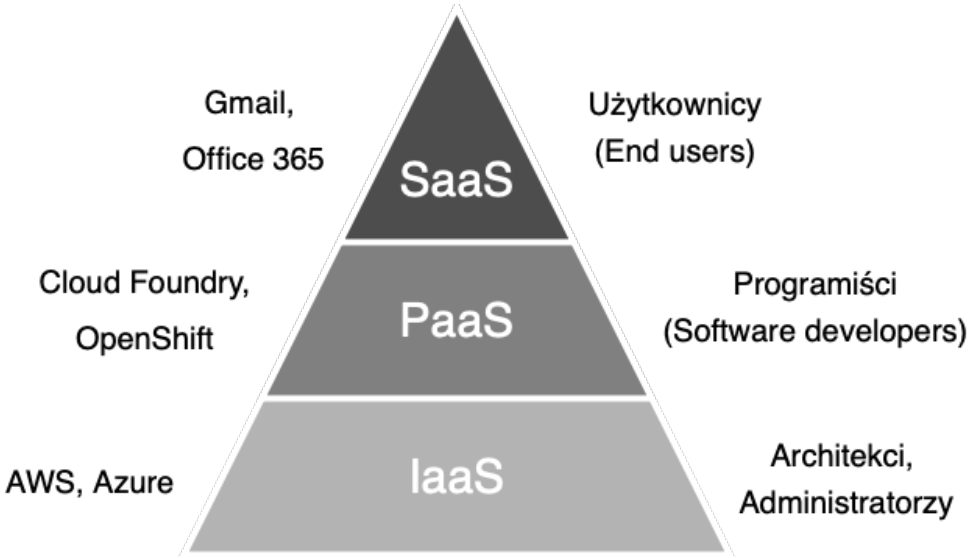
Obecnie chmury publiczne, dzięki oferowanym modelom, mogą zostać znacznie szybciej wdrożone i skalowane. Poziom zabezpieczeń zależy od dostawcy chmury - mogą oni oferować w tym zakresie wiele rozwiązań, np. systemy wykrywania włamań i zapobiegania im (ang. *IDPS - Intrusion Detection and Prevention System*). [67].

2.2.2.3 Chmura hybrydowa

Zgodnie z nazwą, chmury hybrydowe stanowią połączenie prywatnych i publicznych chmur obliczeniowych. Są one stosowane w przypadku przekroczenia możliwości chmury prywatnej [87]. Umożliwiają skalowanie usług do chmury publicznej w celu zwiększenia mocy obliczeniowej [66].

2.2.3 Modele chmury obliczeniowej

Chmurę podzielić można także na określone modele wdrażania usług. Oferują one różny poziom kontroli i rozszerzają wzajemnie swoje możliwości, stąd często określane są mianem *stosu* [70]. Poniższy rozdział opisuje trzy główne modele chmury obliczeniowej: *IaaS*, *PaaS* oraz *SaaS*.



Rysunek 2.2: Modele chmury obliczeniowej
(źródło: opracowanie własne z wykorzystaniem programu draw.io).

2.2.3.1 IaaS - Infrastruktura jako usługa

Infrastruktura jako usługa (ang. *Infrastructure as a Service* (IaaS)) to najbardziej podstawowy model chmury obliczeniowej. Umożliwia on wynajęcie infrastruktury IT od dostawcy oraz płatność zgodnie z rzeczywistym wykorzystaniem zasobów [70]. Są to podstawowe zasoby (instancje obliczeniowe, funkcje sieciowe, przestrzeń dyskowa), zapewniane dzięki wykorzystaniu maszyn wirtualnych. Najczęściej korzystają z nich architekci (projektujący infrastrukturę IT) oraz administratorzy (zarządzający nią). Przykładami takich usług są: Microsoft Azure Virtual Machines, Google Compute Engine oraz Amazon EC2 [102].

2.2.3.2 PaaS - Platforma jako usługa

Platforma jako usługa (ang. *Platform as a Service* (PaaS)) zapewnia narzędzia niezbędne do tworzenia i hostowania aplikacji w chmurze. Jest ona szczególnie użyteczna dla deweloperów (programistów), piszących różnego rodzaju aplikacje w danym języku programowania. Oferuje im najważniejsze komponenty, znacznie usprawniające obsługę aplikacji mobilnych lub sieci Web (dostępnych przez Internet). Jednocześnie nie są oni odpowiedzialni za konfigurację i utrzymywanie całości infrastruktury bazowej - serwerów, baz danych, przestrzeni dyskowej oraz sieci [70]. Przykładami takich usług są: AWS Elastic Beanstalk oraz Google App Engine [102].

2.2.3.3 SaaS - Oprogramowanie jako usługa

Oprogramowanie jako usługa (ang. *Software as a Service* (SaaS)) łączy oprogramowanie działające w chmurze wraz z konieczną infrastrukturą. Oprogramowanie to dostarczane jest za pośrednictwem Internetu, hostowane oraz zarządzane przez dostawców chmury. SaaS umożliwia korzystanie z jednej aplikacji na wielu różnych urządzeniach [70]. Kontakt z nim mają przede wszystkim końcowi użytkownicy danej aplikacji. Przykładami takich oprogramowań są: Amazon WorkSpace, Google Apps for Work oraz Microsoft Office 365 [102].

2.2.4 Podstawy przetwarzania w chmurze

Każda chmura obliczeniowa składa się z określonych elementów technologicznych, umożliwiających jej prawidłowe funkcjonowanie. Zostały one przedstawione w poniższej liście.

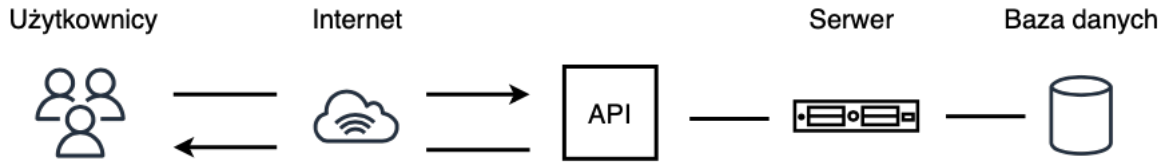
- **Centrum danych** (ang. *Data center*) - miejsce, w którym zgromadzone są fizyczne maszyny: serwery oraz niezbędny sprzęt sieciowo-komunikacyjny. Są to maszyny - hosty, wirtualizowane i udostępniane użytkownikom na żądanie w formie mocy obliczeniowej. Działają one bez przerwy i wymagają specjalistycznej infrastruktury, zarówno odpowiedniego chłodzenia oraz stabilizatorów napięcia, jak i zabezpieczeń. Największe centra danych należą do firm takich jak Amazon, Microsoft oraz Google - dostawców chmury publicznej. Są one ulokowane na całym świecie, głównie w pobliżu rejonów, w których użytkownicy intensywnie korzystają z ich zasobów. Zmniejsza to opóźnienia i ułatwia przełączanie się pomiędzy maszynami w przypadku awarii. Osoby decydujące się na korzystanie z chmury prywatnej samodzielnie gromadzą i konserwują zasoby fizyczne - choć na znacznie mniejszą skalę [87].

Rysunek 2.3: Centrum danych Amazon Web Services (AWS) [90].

- **Wirtualizacja serwerów** - pełna wirtualizacja zasobów zgromadzonych w centrum danych umożliwiła radykalny wzrost wykorzystania mocy procesora, co wpłynęło na ogromne oszczędności finansowe. Na jednym, zwirtualizowanym serwerze fizycznym może działać dowolnie wiele systemów operacyjnych. Ponadto

czas konfiguracji oraz wdrożenia został znacznie zredukowany, a opłaty pobierane są za każdą godzinę wykorzystania procesora. Zanim rozpoczął się proces wirtualizacji fizycznych zasobów, centra danych naliczały opłaty za wykorzystywane serwery, na których działała aplikacja. Obecnie płaci się wyłącznie za faktyczne zużycie mocy obliczeniowej [87].

- **API chmury** (ang. *Application Programming Interface*) - Interfejs Programowania Aplikacji umożliwia dostęp do zasobów udostępnianych w chmurze. W przypadku modelu SaaS użytkownik otrzymuje go zazwyczaj w formie interfejsu w przeglądarce internetowej. Korzystając z modelu niższego poziomu (IaaS) konieczne jest wykorzystanie API. Dzięki niemu możliwa jest interakcja z chmurą - dodawanie zasobów, zarządzanie nimi, ich konfiguracja oraz usuwanie, gdy nie są już potrzebne [87].



Rysunek 2.4: Przykład komunikacji użytkowników z bazą danych za pośrednictwem API (źródło: opracowanie własne z wykorzystaniem programu draw.io).

- **Przechowywanie danych** - chmura zapewnia miejsce, w którym możliwe jest przechowywanie np. obrazów maszyn (definicja obrazu została opisana w rozdziale 2.3.1) oraz danych wykorzystywanych przez aplikacje. Są to tzw. *magazyny obiektów*. Opłaty pobierane są zazwyczaj za ilość przesyłanych danych oraz za zajmowane przez nie miejsce. Dzięki temu możliwe jest tworzenie i zapisywanie kopii zapasowych danych lokalnych, synchronizacja dysku wirtualnego z chmurą (dostęp do danych z poziomu wielu różnych maszyn), udostępnianie przechowywanych danych (np. klientom, z poziomu przeglądarki) itp. Chmura umożliwa również korzystanie z określonych struktur przechowywania obiektów - baz danych [87].

- **Skalowalność i elastyczność** - są to jedne z podstawowych pojęć określających możliwości chmury. Skalowalność to *zdolność platformy do obsłużenia zwiększonej liczby użytkowników korzystających z aplikacji* [87]. Elastyczność to *zdolność skalowania w górę lub w dół bez przerywania normalnego działania aplikacji* [87]. Te dwie cechy wpływają na opłacalność przenoszenia aplikacji i/lub infrastruktury do chmury.

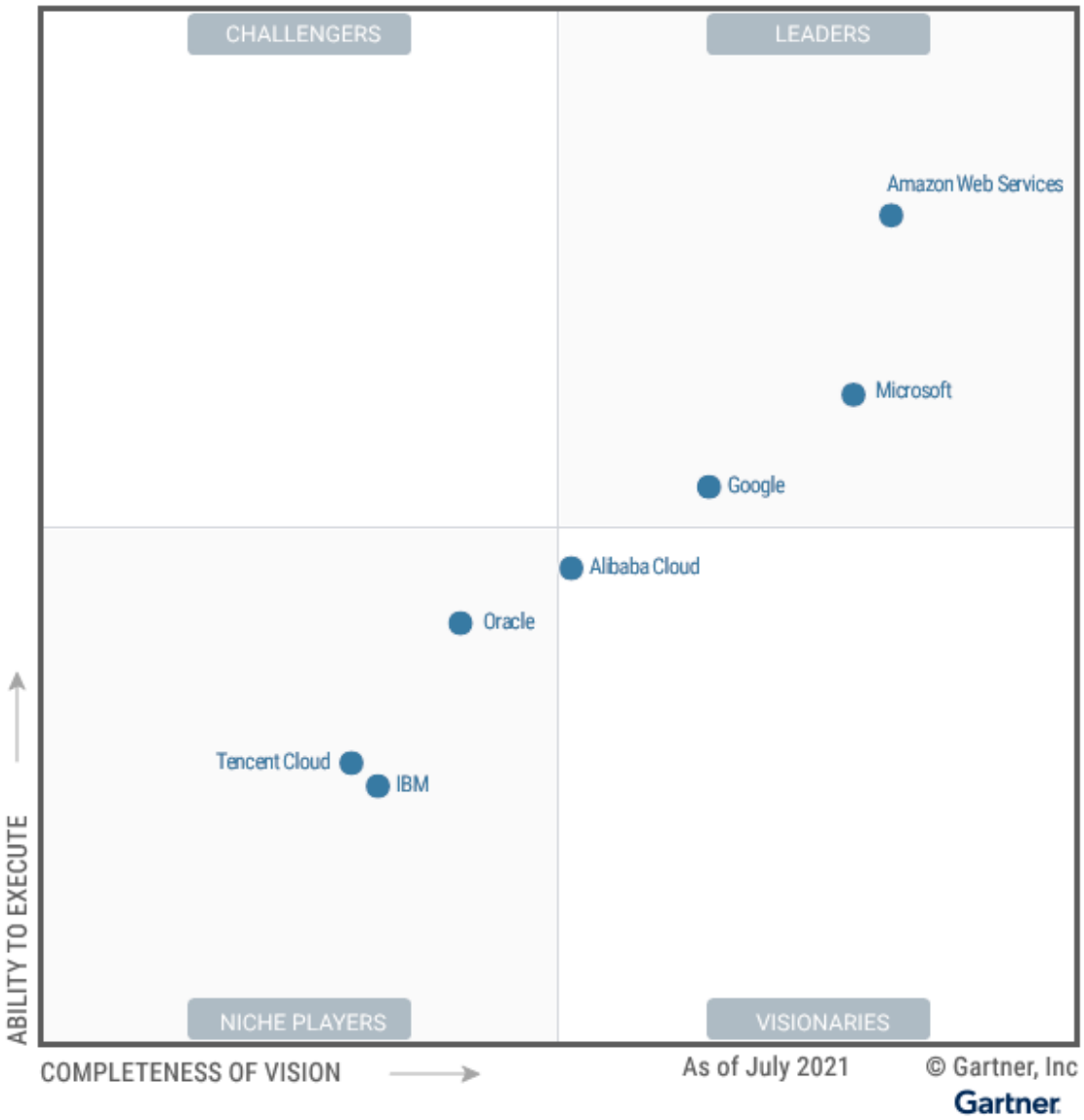
2.2.5 Dostawcy technologii chmurowych

Dostawcy technologii chmurowych (ang. *cloud providers*) to zewnętrzne firmy oferujące usługi oparte na przetwarzaniu w chmurze, takie jak: platforma (PaaS), infrastruktura (IaaS), aplikacje (SaaS) oraz magazyny obiektów [68]. Są oni odpowiedzialni za centra danych którymi dysponują i których zasoby udostępniają użytkownikom na żądanie (aspekt ten opisany został w rozdziale 2.2.2.2).

Klasyfikacji dostawców technologii chmurowych dokonuje firma badawcza Gartner, co roku udostępniając raport *Magic Quadrant for Cloud Infrastructure and Platform Services* [32].

„Badanie to dzieli dostawców na cztery ćwiartki: niszowych graczy (ang. niche players), kandydatów (ang. challengers), wizjonerów (ang. visionaries) i liderów (ang. leaders) i zawiera krótki przegląd rynku przetwarzania w chmurze." [102]

W 2021 roku liderami wśród dostawców usług chmurowych byli: Amazon (Amazon Web Services (AWS)), Microsoft (Azure) oraz Google (Google Cloud Platform (GCP)). Ich pozycja na rynku pozostaje niezmiennie silna od 2018 roku - wówczas do ówczesnych liderów (Amazon oraz Microsoft) dołączyło Google.



Rysunek 2.5: *„Magic Quadrant for Cloud Infrastructure and Platform Services"* - wyniki badania z 2021 roku [32].

Zarówno Amazon Web Services, jak i Microsoft Azure oraz Google Cloud Platform oferują podobne zestawy usług związanych z chmurą. Ich oferty charakteryzują się samoobsługą, szybkim wdrażaniem nowych systemów, skalowalnością, elastycznością oraz bezpieczeństwem. Przy wyborze odpowiedniego dostawcy chmury klienci kierują się przede wszystkim ich podejściem do konkretnych usług oraz ogólną filozofią firmy - tym, na co kładzie ona największy nacisk. Kolejnym aspektem jest różnorodność oferowanych rozwiązań w obrębie jednej dziedziny (np. uczenia maszynowego, przetwarzania w chmurze, przechowywania danych lub konteneryzacji) oraz ich cena.

Poniżej przedstawiona została charakterystyka trzech największych dostawców usług chmurowych:

- **Amazon Web Services (AWS)** - *„jest platformą usług sieciowych oferujących na różnych poziomach abstrakcji rozwiązania w zakresie przetwarzania i przechowywania danych oraz pracy w sieci (...) Usługi AWS dobrze ze sobą współdziałają: można ich użyć do odwzorowania istniejących ustawień sieci lokalnej lub skonfigurować je od podstaw. Model cenowy usług to płatność za wykorzystanie (ang. pay-per-use)"* [102]. W 2021 roku, jedenasty raz z rzędu, Gartner umieścił Amazon Web Services w części „Leaders" swojego raportu, wskazując AWS jako *„usługodawcę mającego najpełniejszą kompletność wizji i największą zdolność do jej wykonania"* [16]. Największymi atutami AWS są: ogromny zestaw oferowanych usług (obecnie (2021 r.) ponad 200 [91]), doświadczenie oraz oferta dostosowana do klientów korporacyjnych. Amazon buduje prywatną chmurę dla amerykańskiej Centralnej Agencji Wywiadowczej (wewnątrz której funkcjonuje Netflix - najpopularniejsza usługa udostępniająca filmy i seriale na żądanie). Wśród klientów AWS znajdują się także m.in.: AstraZeneca, Pfizer, AirBnB, Nike, Financial Times i wiele innych [16].

Rysunek 2.6: Amazon Web Services [24].

- **Microsoft Azure** - platforma usług oferowana przez Microsoft. Jej kluczowym elementem są maszyny wirtualne oraz narzędzia służące do wdrażania aplikacji w chmurze (np. *Cloud Services* i *Azure Autoscaling*). Przez długi czas Azure wyróżniał się na tle pozostałych liderów prostotą tworzenia środowisk hybrydowych i wielochmurowych (składających się z usług dostarczanych przez różnych dostawców), udostępniając *Azure Stack*. Szerokie grono klientów Azure uzyskał w oparciu o dotychczasowe produkty firmy Microsoft, które w naturalny sposób się z nim łączą (np. *Windows Server* lub *Active Directory*). Microsoft promuje platformę Azure poprzez oferty dostosowane do potrzeb konkretnych, pojedynczych klientów. Są to m.in.: Pearson, Ford, NBC News i Easyjet [16].

Rysunek 2.7: Microsoft Azure [49].

- **Google Cloud Platform (GCP)** - platforma usług oferowana przez Google. Ogromny nacisk kładzie ona na rozwój narzędzi z dziedziny sztucznej inteligencji, uważanych za jeden z jej kluczowych atutów i cenionych przez klientów. Google oferuje m.in. gotowe do użytku interfejsy API, obsługujące np. rozpoznawanie języka naturalnego i analizę obrazów. Ponadto Google znane jest z aktywnego tworzenia i wspierania projektów Open Source, np. Kubernetesa (2.3.3). Niemniej GCP to również typowe dla dostawców chmur usługi, np. *Compute Engine* dostarczający maszyny wirtualne, powiązane z przestrzenią dyskową i gwarantujący wydajność oraz szybkie uruchamianie. Obecnie Google wciąż zdobywa nowych klientów, starając się dorównać pozostałym liderom. Do tej pory firma ta nie poszukiwała aktywnie dużych firm, z którymi mogłaby współpracować jako strategiczny partner chmurowy. Mimo to z oferty Google na ten moment korzysta np. Spotify, HSBC, Snap oraz Disney [16].

Rysunek 2.8: Google Cloud Platform [76].

2.3 Konteneryzacja

2.3.1 Definicja konteneryzacji

Konteneryzacja jest narzędziem umożliwiającym umieszczenie aplikacji (wraz z towarzyszącymi jej procesami oraz konfiguracją) w wirtualnej jednostce zwanej *kontenerem*. Zawiera on wszystko, co jest konieczne do prawidłowego funkcjonowania umieszczonego w nim oprogramowania. Jest to szczególnie istotne w przypadku ryzyka wystąpienia tzw. *piekła zależności* - określającego trudne do spełnienia zależności, uniemożliwiające instalację programów. W tym przypadku konteneryzacja pozwala na zapis działającego programu na wypadek wyjścia z użycia bibliotek i zależności przez niego wykorzystywanych. Kontenery posiadają własne, wydzielone obszary pamięci RAM oraz dysku, a także prywatny adres IP. Są one wzajemnie odizolowane i działają niezależnie, jednak mogą się ze sobą komunikować w ściśle określonych kanałach wymiany informacji [62]. Kontener, zapisany w formie pliku *obrazu* wykonywany jest przez *środowisko wykonawcze kontenera* [4]. Przykładem środowiska wykonawczego jest Docker.

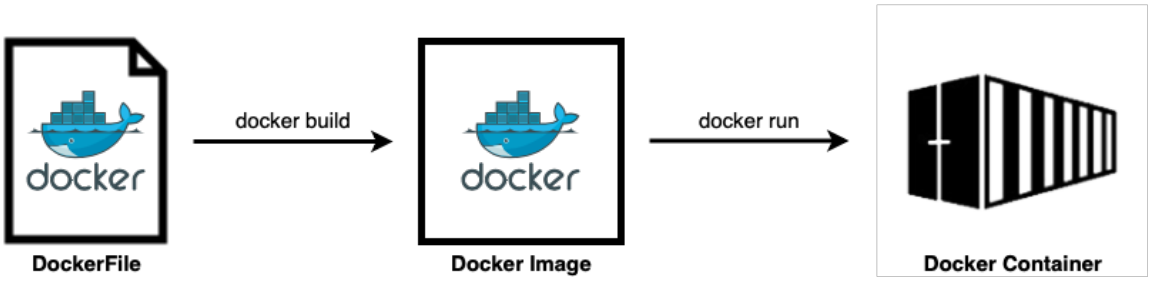
Rysunek 2.9: Docker [25].

Docker to platforma typu Open Source, umożliwiająca umieszczanie aplikacji w kontenerach oraz zarządzanie nimi. Oferuje on narzędzia służące do tworzenia, wdrażania, uruchamiania, aktualizowania oraz zatrzymywania kontenerów z wykorzystaniem komend, dzięki którym konteneryzacja staje się prostsza i bezpieczniejsza. Nazwa *Docker* odnosi się zarówno do firmy - Docker Inc. (sprzedającej komercyjną wersję platformy) jak i do projektu Open Source Docker (współtworzonego przez firmę Docker Inc. oraz inne organizacje) [45].

Podstawowe pojęcia związane z konteneryzacją oraz Dockerem:

- **DockerFile** - plik tekstowy, zawierający listę instrukcji interfejsu wiersza poleceń (ang. *Command Line Interface* (*CLI*)), na których podstawie Docker po uruchomieniu tworzy *obraz* [45].
- **Docker Image** (obraz) - zawiera kod źródłowy umieszczanej w kontenerze aplikacji wraz ze wszystkimi bibliotekami i koniecznymi do uruchomienia zależnościami. Składa się z warstw, odpowiadających kolejnym wersjom obrazu. Zapisywanie wszystkich poprzednich wersji umożliwia ewentualne wycofywanie zmian lub ich ponowne wykorzystanie np. w innych projektach. Najnowsza wersja zapisywana jest w *warstwie kontenera*. Przy użyciu jednego obrazu można uruchomić wiele aktywnych instancji kontenerów, wykorzystujących wspólny stos [45].
- **Docker Container** (kontener) - odseparowane środowisko wewnątrz którego działa dana aplikacja, stworzone na podstawie obrazu. Jest to działająca instancja obrazu, ich aktywna i wykonywalna treść [45].

- **Docker Compose** (kompozytor) - narzędzie umożliwiające tworzenie i uruchamianie środowiska wymagającego pracy wielu kontenerów [12].



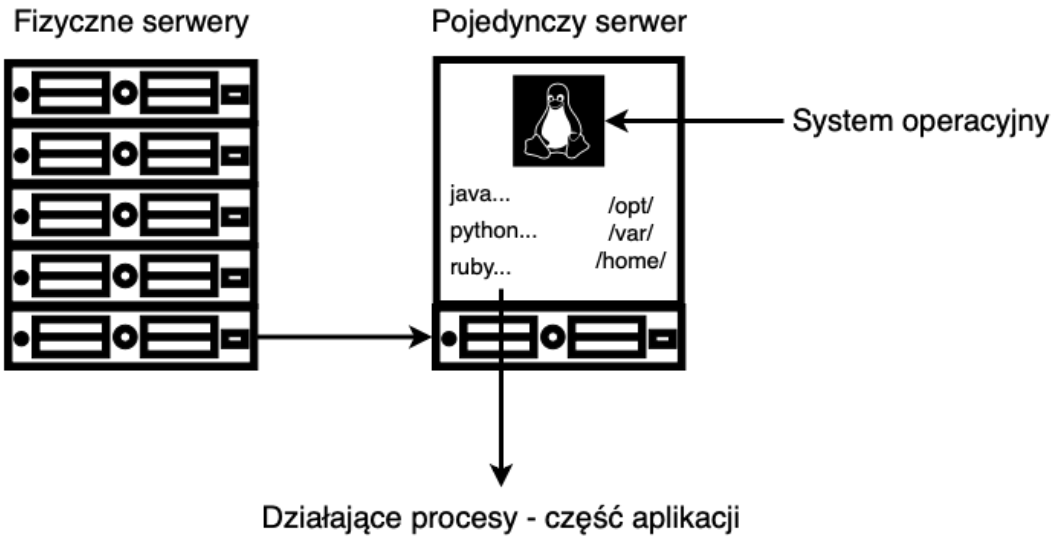
Rysunek 2.10: Proces powstawania kontenera - elementy wraz z podstawowymi komendami (źródło: opracowanie własne z wykorzystaniem programu draw.io).

2.3.2 Rozwój konteneryzacji

Podrozdział ten napisany został w oparciu o wykład Jakuba Bujnego *„Kubernetes bez tajemnic”* [1]. Przedstawi on w sposób obrazowy cel powstania konteneryzacji oraz różnicę pomiędzy konteneryzacją a wirtualizacją. Konteneryzacja jest narzędziem wykorzystywanym od wielu lat w różnych formach, jednak przedstawiony opis będzie dotyczyć jej obecnej, w pełni rozwiniętej oraz popularnej postaci.

2.3.2.1 Maszyny fizyczne

Chcąc uruchomić aplikację bez wykorzystania wirtualizacji, chmury obliczeniowej oraz konteneryzacji, wykorzystuje się jedynie fizyczne zasoby danego serwera. Posiada on zainstalowany system operacyjny (np. Linux), który generuje procesy działające w tle oraz charakteryzuje się określonymi parametrami (moc procesora, wielkość pamięci).

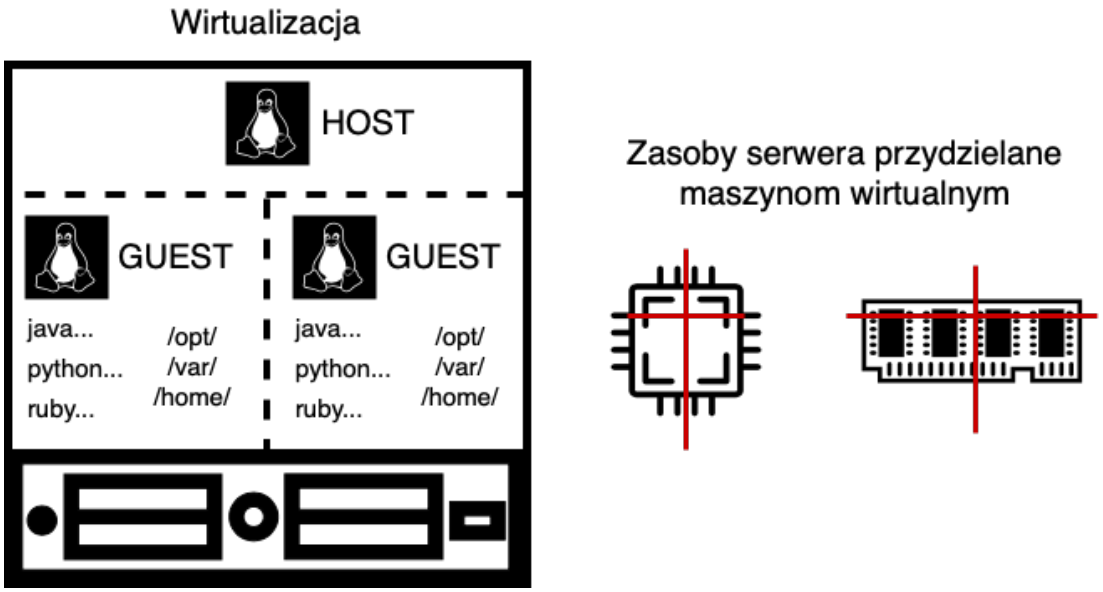


Rysunek 2.11: Pojedynczy serwer fizyczny z zainstalowanym systemem operacyjnym (Linux) oraz aplikacją (Java, Python, Ruby) (źródło: opracowanie własne z wykorzystaniem programu draw.io).

Do pewnego stopnia rozwiązanie to jest wystarczające - gdy nasza aplikacja nie konsumuje w znacznym stopniu zasobów maszyny oraz gdy w odpowiedni sposób ją wykorzystujemy. Jednak w miarę wzrostu skali ilość potencjalnych zagrożeń i problemów wzrasta. Często w kodzie aplikacji znajduje się błąd (ang. *bug*), który z czasem zaczyna zagarniać pamięć maszyny, uniemożliwiając działanie pozostałym procesom. Aplikacja może być także podatna na zdalne wykonanie kodu, umożliwiające uszkodzenie serwera przez osoby trzecie. Zdarza się również, że użytkownik przez przypadek użyje w wierszu poleceń komendy, która usunie całą zawartość głównego folderu maszyny (np. `rm -rf /`). Chcąc minimalizować możliwą ilość niebezpieczeństw i niepowodzeń zaczęto wykorzystywać potencjał wirtualizacji na szeroką skalę.

2.3.2.2 Maszyny wirtualne

Zasada działania wirtualizacji opisana została w rozdziale 2.1. Jej wykorzystanie umożliwiło wirtualizację systemów operacyjnych na maszynie - hoście. Dzięki temu procesy stały się odizolowane, zmalało więc ryzyko zagarniania zasobów przez jeden z nich. Ponadto użytkownicy w razie popełnionego błędu uszkadzali wyłącznie część systemu - nie całość maszyny.

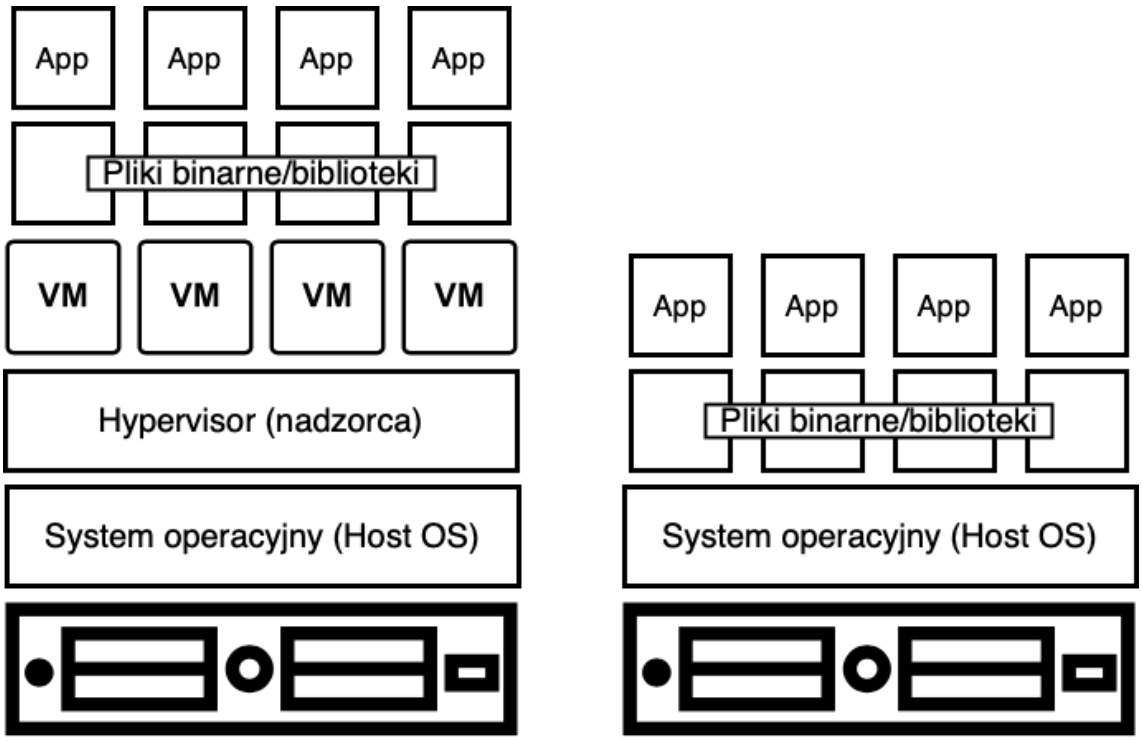


Rysunek 2.12: Wirtualizacja pojedynczego serwera oraz przedstawienie „sztywnego" podziału zasobów (źródło: opracowanie własne z wykorzystaniem programu draw.io).

Wirtualne maszyny zapewniły izolację zasobów i stanowiły kompletne, niezależne środowiska. Mimo to one również cechowały się określonymi wadami. Przede wszystkim zasoby fizycznego serwera przydzielane są maszynom wirtualnym „na sztywno" - tak, jak zostało to przedstawione na rysunku. Każda z nich otrzymuje określoną ilość mocy obliczeniowej oraz pamięci, wynikającej z podziału względem wszystkich. Ponadto wykorzystywanie maszyn wirtualnych prowadziło do spadku wydajności aplikacji względem uruchamiania ich na maszynie fizycznej. Wynika to z faktu wirtualizacji systemu operacyjnego, który konsumuje zasoby sprzętowe i stanowi dodatkową warstwę abstrakcji wraz z hypervisorem. Każda kolejna warstwa działa na niekorzyść poprzedniej, powodując spadek wydajności aplikacji jaka działa na maszynie [62].

2.3.2.3 Konteneryzacja

Z uwagi na na problemy z jakimi mierzono się przy wykorzystaniu maszyn wirtualnych, konteneryzacja znacznie zyskała na popularności. Umożliwia ona izolację zasobów, analogicznie jak w przypadku wirtualizacji, jednocześnie redukując ilość warstw abstrakcji poprzez usunięcie zduplikowanego systemu operacyjnego oraz hypervisora.

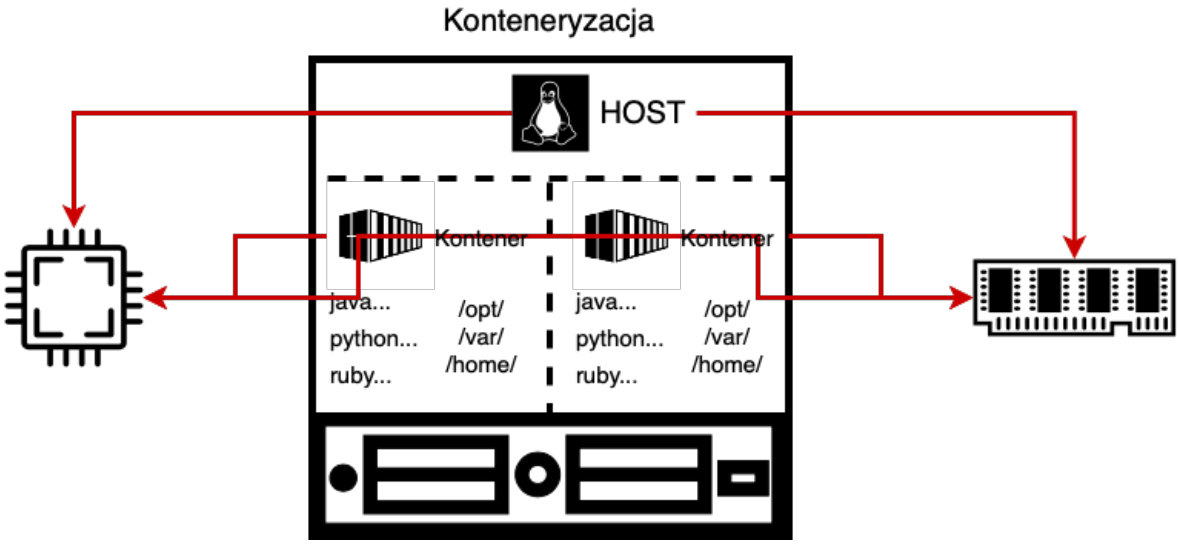


Rysunek 2.13: Porównanie ilości warstw abstrakcji w przypadku wirtualizacji (po lewej) i konteneryzacji (po prawej) (źródło: opracowanie własne z wykorzystaniem programu draw.io).

Obrazy maszyn wirtualnych funkcjonują w niezależnych systemach operacyjnych, podczas gdy obrazy kontenerów działają w obrębie jądra tego samego systemu operacyjnego. Kontenery posiadają własne systemy plików i zmienne środowiskowe, dzięki czemu są samowystarczalne. Ponadto są one odizolowane nie tylko od systemu operacyjnego hosta, ale również od siebie samych [52].

Pomimo znacznego wzrostu wydajności (ok. 30% względem maszyn wirtualnych [4]), porównywalnego do działania aplikacji na fizycznym serwerze, zachowane zostają zalety wirtualizacji. Należą do nich: skalowalność, separacja oraz łatwość w przenoszeniu instancji [62]. Ponadto podział zasobów jest znacznie prostszy - kontenery mają dostęp do tych samych zasobów co maszyna hosta, tak jak zostało to przedstawione na rysunku. Gdy jest to konieczne - administrator serwera może ograniczyć zużycie zasobów przez konkretne kontenery odpowiednimi, wprowadzonymi ręcznie limitami. Z perspektywy maszyny kontener jest jednym z wielu działających procesów, dzięki czemu dobrze zaprojektowany obraz kontenera jest ok. 100 razy mniejszy niż typowy obraz maszyny wirtualnej, który wynosi ok. 1 GiB [1] [4].

[1]GiB - Gibibajt, definiowany jako 1 024 mebibajtów (MiB). 1 [MiB] = 1024 × 1024 [B]. Jest to jednostka danych Międzynarodowej Komisji Elektrotechnicznej (IEC) [4].



Rysunek 2.14: Konteneryzacja serwera oraz przydzielanie jego zasobów kontenerom (źródło: opracowanie własne z wykorzystaniem programu draw.io).

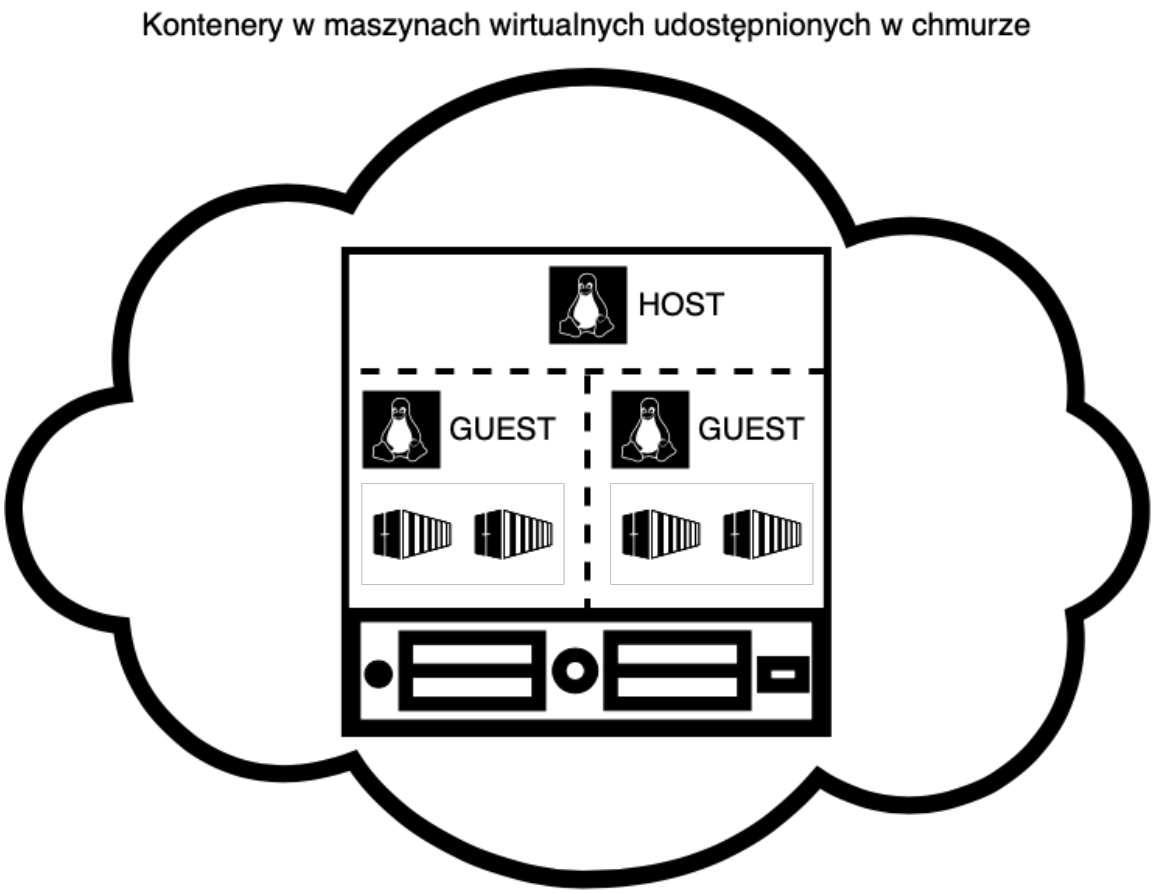
2.3.2.4 Uruchamianie kontenerów w chmurze

Popularność rozwiązań kontenerowych została zauważona przez dostawców chmury publicznej. Z tego powodu zaczęli oni udostępniać maszyny wirtualne z możliwością konteneryzacji. Ponadto powstało wiele dedykowanych kontenerom usług, np. Azure Container Instances, Azure WebApps for Containers, Amazon Elastic Container Service, Google Kubernetes Engine itp.

Uruchamianie kontenerów na maszynach wirtualnych wbrew pozorom posiada wiele zalet. W dalszym ciągu zachowuje się efektywne wykorzystanie zasobów maszyny, jednocześnie optymalizując koszty - na jednej maszynie wirtualnej może działać wiele kontenerów, a więc wiele mikroserwisów (definicja mikroserwisów przytoczona została w rozdziale 2.4.2.2). Z rozwiązania tego korzysta m.in. Netflix. Dawniej na każdą maszynę wirtualną przypadał jeden mikroserwis, działający i skalujący się w niej.

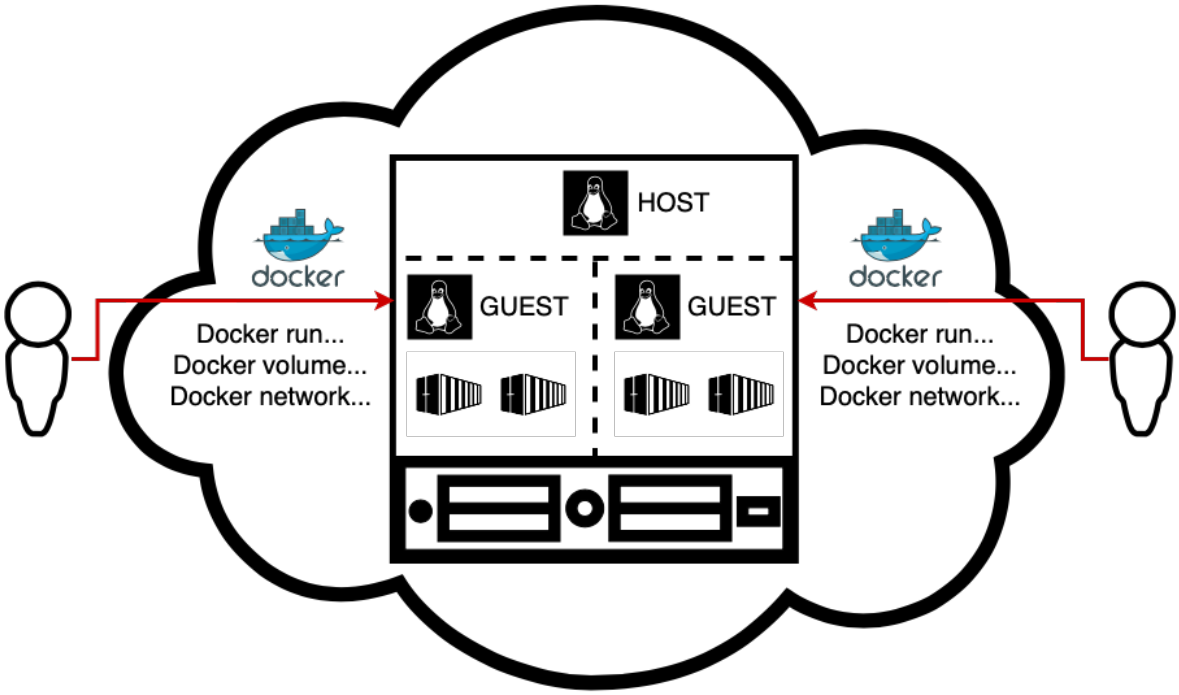
Brendan Burns i David Oppenheimer opisali możliwości kontenerów w artykule *Design Patterns for Container-based Distributed Systems* [13] w następujący sposób:

„Dzięki hermetycznemu zamknięciu, zadbaniu o zależności i wprowadzeniu atomowego sygnału wdrażania („sukces”/„błąd”) [kontenery] znacznie poprawiają dotychczasowy stan wdrażania oprogramowania w centrum danych lub chmurze. Jednak kontenery mogą być potencjalnie czymś więcej niż tylko lepszym narzędziem do wdrażania - wierzymy, że ich działanie może być analogiczne do obiektów w obiektowych systemach oprogramowania i jako takie umożliwi opracowanie wzorców projektowania systemów rozproszonych.”



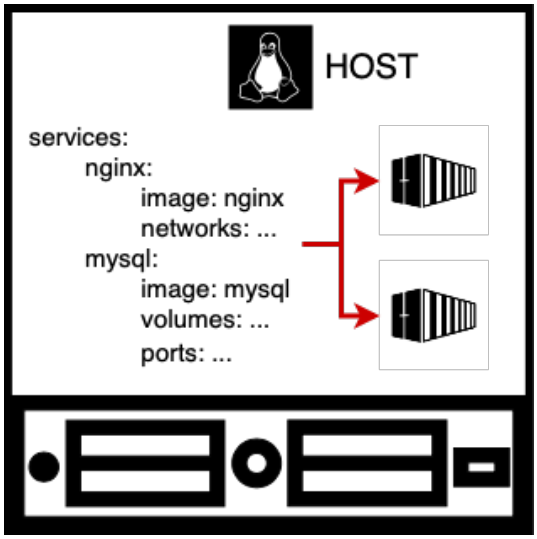
Rysunek 2.15: Uruchamianie kontenerów w chmurze obliczeniowej (źródło: opracowanie własne z wykorzystaniem programu draw.io).

Chcąc skonfigurować powyższą infrastrukturę z wykorzystaniem Dockera, należy wykorzystać wiersz poleceń i szereg odpowiednich komend. Umożliwiają one: zalogowanie się do maszyny wirtualnej (`ssh`), stworzenie kontenera oraz jego niezbędnych elementów (`docker run, docker volume`), stworzenie architektury sieci (`docker network`) itd. W miarę wzrostu skali ręczne wpisywanie przytoczonych komend (ręczna konfiguracja) staje się niemal niemożliwe. Ciężko jest również opracować skrypty z rozwiniętą logiką, zdolnych do iterowania się w zależności od stanu konfiguracji w danym momencie. Z tego względu imperatywne podejście do konfiguracji infrastruktury kontenerowej zastąpione zostało podejściem deklaratywnym, w wyniku czego powstał *Docker Compose* (2.3.1).



Rysunek 2.16: Uruchamianie kontenerów bez kompozytora (źródło: opracowanie własne z wykorzystaniem programu draw.io).

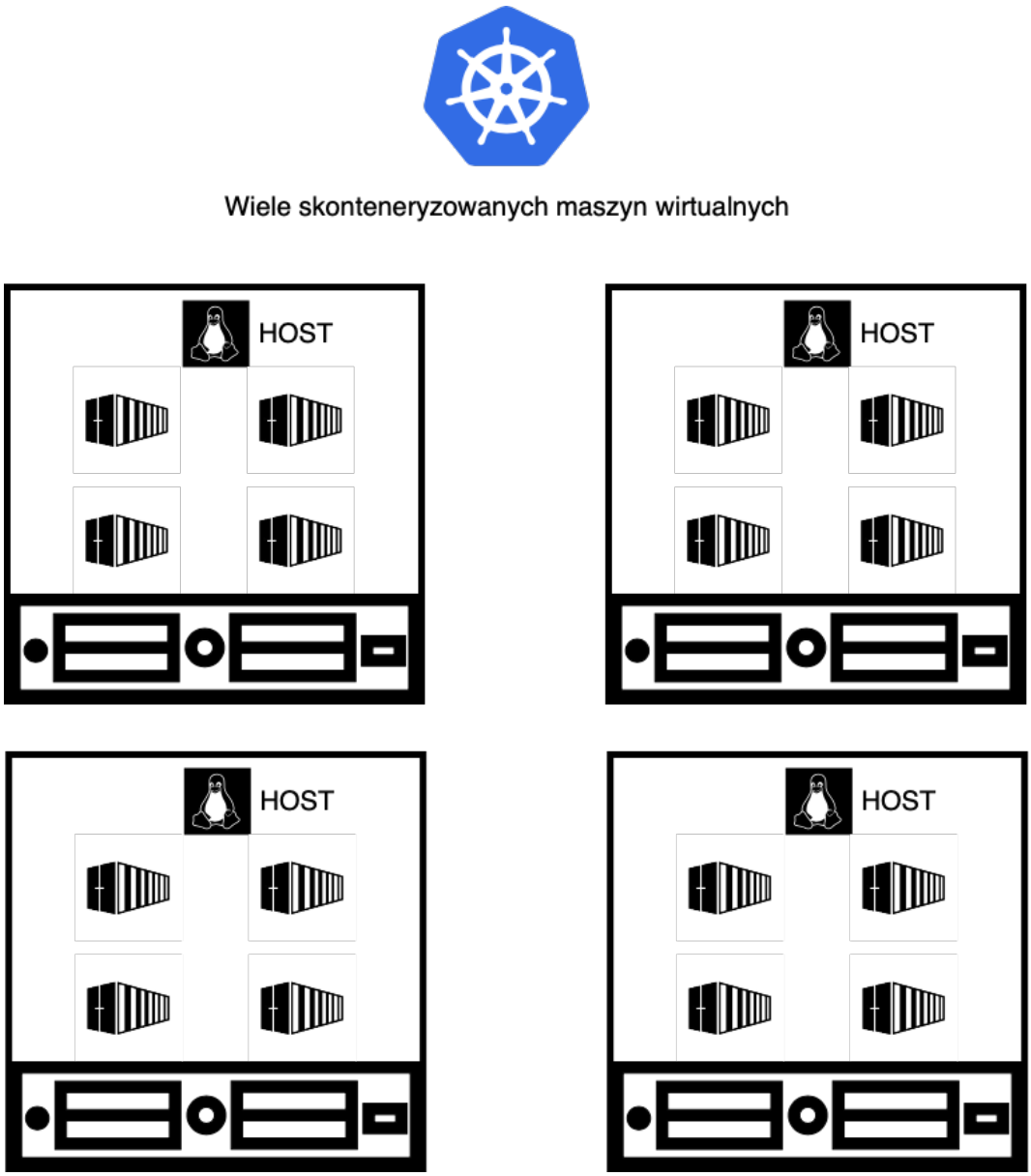
Docker Compose to narzędzie umożliwiające wykonywanie *plików konfiguracyjnych* (2.3.1) z poziomu wiersza poleceń (`docker compose`). Pliki te zawierają opis pożądanej konfiguracji w formie zestawu komend, na podstawie których Docker tworzy obrazy kontenerów. Narzędzie to jest szczególnie przydatne podczas pracy z wieloma kontenerami. Często jedna aplikacja podzielona jest na wiele kontenerów - docker compose umożliwia wykorzystanie DockerFile do skonfigurowania jej w taki sposób, aby wszystkie jej elementy działały równocześnie w odizolowanym środowisku [52].



Rysunek 2.17: Docker Compose (kompozytor)
(źródło: opracowanie własne z wykorzystaniem programu draw.io).

2.3.3 Orkiestracja

Orkiestracja staje się narzędziem niezbędnym w miarę wzrostu skali produkcji. Podczas rozwoju aplikacji zwiększa się zużycie zasobów maszyny, stąd konieczność posiadania wielu serwerów. Większa ilość maszyn wymaga zaawansowanej konfiguracji. Mikroserwisy muszą być zdolne do wzajemnego porozumiewania się, aby aplikacja działała w prawidłowy sposób. Ponadto administratorzy infrastruktury muszą dbać o prawidłowy podział ruchu sieciowego użytkowników, aby nie obciążać wyłącznie jednej maszyny (ang. *Load balancing*). Istnieje również ryzyko awarii jednej z maszyn, co destabilizuje działanie całości aplikacji - szczególnie, gdy maszyna ta utrzymywała kluczową funkcjonalność orpogramowania. W odpowiedzi na te i wiele innych potencjalnych problemów, związanych z zarządzaniem rozrastającą się infrastrukturą kontenerową, powstała orkiestracja.



Rysunek 2.18: Orkiestracja z wykorzystaniem Kubernetes
(źródło: opracowanie własne z wykorzystaniem programu draw.io).

2.3.3.1 Definicja orkiestracji

Orkiestracja to sposób automatyzowania konfiguracji, koordynacji pracy oraz zarządzania systemami i oprogramowaniem. Ma ona na celu optymalizację częstego wykonywania powtarzalnych procesów [62]. Orkiestracja infrastruktury opartej na kontenerach możliwa jest dzięki orkiestratorom kontenerów (ang. *container orchestrator*). Są to oprogramowania łączące wiele różnych maszyn w jeden spójny klaster. *„Połączone jednostki obliczeniowe, postrzegane przez użytkownika jako pojedynczy bardzo wydajny komputer (na którym mogą działać kontenery).”* [4].

Podstawowe pojęcia związanie z orkiestracją [4]:

- **Orkiestracja** - koordynacja i sekwencjonowanie wielu różnych działań na rzecz wspólnego celu.
- **Planowanie** - zarządzanie dostępnymi zasobami oraz przydzielanie zadań tam, gdzie mogą zostać uruchomione w najbardziej wydajny sposób.
- **Zarządzanie klastrami** (ang. *cluster management*) - łączenie wielu serwerów (zarówno fizycznych jak i wirtualnych) w zunifikowaną, odporną na awarie grupę.
- **Orkiestrator kontenerów** (ang. *container orchestrator*) - pojedyncza usługa, odpowiedzialna za planowanie i koordynację pracy klastra oraz zarządzanie nim.
- **Konteneryzacja** - stosowanie kontenerów jako standardowej metody wdrażania i uruchamiania oprogramowania.

Najpopularniejszą platformą służącą do orkiestracji kontenerów jest obecnie *Kubernetes*. Narodził się on w 2014 r., jako projekt o otwartym kodzie źródłowym stworzony przez Google. Nazwa nawiązuje do greckiego słowa oznaczającego sternika (stąd też logo platformy). Kubernetes umożliwił wdrożenie orkiestratora kontenerów przez każdą osobę chcącą z niego skorzystać. Google oparło jego podstawy na systemie *Borg* - wewnętrznym orkiestratorze firmy, ściśle powiązanym z jej zastrzeżonymi technologiami. Kubernetes w bardzo szybkim czasie zyskał popularność i wygrał walkę z funkcjonującą wówczas, komercyjną konkurencją, jako bezpłatne i otwarte oprogramowanie [4].

Rysunek 2.19: Kubernetes [55].

Kubernetes zarządza klastrem maszyn o wysokiej dostępności, działających w ramach spójnej całości. Poprzez system obiektów abstrakcyjnych umożliwia on uruchamianie aplikacji w kontenerach bez konieczności przypisywania ich do konkretnej maszyny. Aplikacje te muszą zostać wcześniej skonteneryzowane [54].

2.3.3.2 Praca z obiektami

Obiekty w Kubernetes to jednostki reprezentujące stan klastra. Opisują one jakie skonteneryzowane aplikacje zostały uruchomione (oraz na jakich węzłach (definicja węzła opisana została w rozdziale 2.3.3.3)), dostępne dla nich zasoby oraz zasady dotyczące ich zachowania (sposób uaktualniania, ponownego uruchamiania, odporność na awarie). Obiekty określają pożądany stan klastra, który Kubernetes ma za zadanie stworzyć oraz utrzymać. Praca z obiektami Kubernetes jest możliwa dzięki wykorzystaniu jego API [2] [57].

Tworzenie obiektów Kubernetes możliwe jest dzięki deklaratywnym *manifestom*. Pliki manifestu, zapisywane w formacie YAML lub JSON, opisują obiekt jaki użytkownik pragnie stworzyć. Pliki te wysyłane są następnie jako żądania API i wykonywane przez Kubernetes (w podobny sposób jak DockerFile w Dockerze). Manifest składa się z czterech podstawowych elementów [96]:

- **apiVersion** - określa wersję API Kubernetesa wykorzystaną do stworzenia danego obiektu.
- **kind** - określa typ tworzonego obiektu (np. Deployment, ReplicaSet, CronJobs).
- **metadata** - określa cechy danego obiektu (np. tagi, przestrzeń nazw [3]), służy do jego jednoznacznej identyfikacji.
- **spec** - opisuje sposób tworzenia i zarządzania obiektem, jego pożądany stan.

Kubernetes w stały i aktywny sposób zarządza rzeczywistym stanem obiektu tak, aby był on zgodny ze stanem żądanym (opisanym w manifeście). Jest to szczególnie ważne w przypadku możliwych awarii. Jeśli jedna z maszyn przestanie działać, wówczas Kubernetes zareaguje na różnicę między zaistniałym stanem infrastruktury a stanem pożądanym, uruchamiając instancję zastępczą [57].

Wyróżnia się następujące sposoby zarządzania obiektami w Kubernetes [57]:

- **Zarządzanie imperatywne** (ang. *Imperative commands*) - zarządzając obiektami imperatywnie użytkownik operuje na nich w sposób bezpośredni. Jest to zalecany sposób uruchomienia jednorazowego zadania w klastrze. Ponieważ technika ta działa bezpośrednio na aktywnych obiektach, nie tworzy historii poprzednich konfiguracji.
- **Imperatywna konfiguracja obiektów** (ang. *Imperative object configuration*) - polecenie `kubectl` określa rodzaj wykonywanej na obiekcie operacji (jego tworzenie, zastępowanie itp.), jego cechy oraz nazwę manifestu do wykonania. Każdy

[2] API (ang. *Application Programming Interface*) - kod, kontrolujący wszystkie punkty dostępowe aplikacji lub serwera. Przesyła on zapytania do danej aplikacji oraz pozwala odesłać informację zwrotną. API stanowi rodzaj pośrednika, umożliwiając programistom tworzenie interakcji pomiędzy wieloma różnymi aplikacjami. Połączone aplikacje mogą wykonywać żądane funkcje, np. udostępnianie danych lub wykonywanie predefiniowanych procesów [105]

[3] Przestrzeń nazw (ang. *namespace*) - zapewnia mechanizm izolowania grup obiektów w obrębie jednego klastra. Nazwy obiektów muszą być unikatowe wewnątrz konkretnej przestrzeni nazw do której zostały przypisane, ale nie we wszystkich przestrzeniach jakie posiada klaster [57]

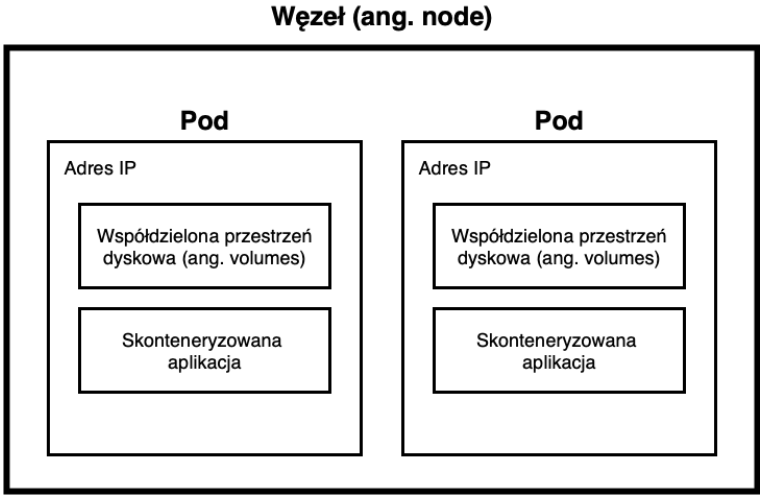
kolejny wykonany manifest nadpisuje poprzedni, usuwając wszystkie zmiany jakie zostały wprowadzone w obiekcie. Jest to zalecany sposób w przypadku tworzenia obiektów, których specyfikacje aktualizowane są niezależnie od pliku konfiguracyjnego.

- **Deklaratywna konfiguracja obiektów** (ang. *Declarative object configuration*) - użytkownik operuje na plikach konfiguracyjnych przechowywanych lokalnie i nie definiuje operacji, które mają zostać na nich wykonane. Operacje tworzenia, aktualizowania i usuwania są automatycznie wykrywane dla każdego obiektu przez `kubectl`, co umożliwia pracę na katalogach, gdzie dla różnych obiektów konieczne mogą być różne rodzaje operacji. Konfiguracja deklaratywna zachowuje historię zmian w nim wprowadzanych.

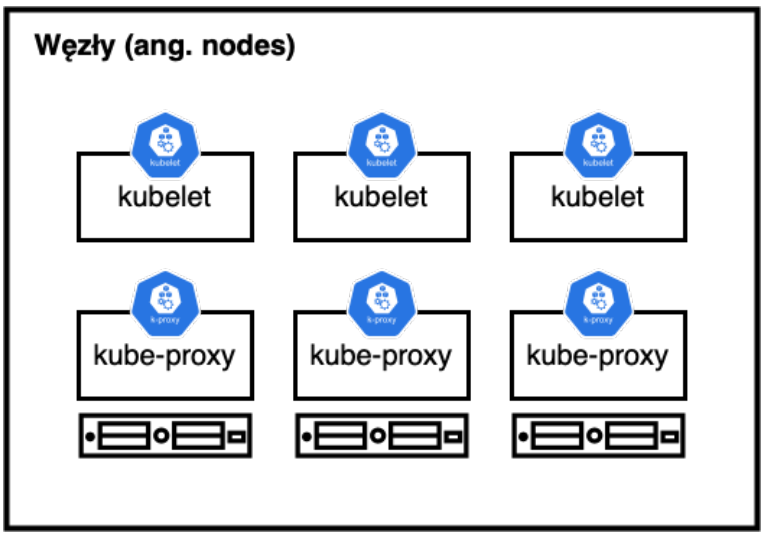
2.3.3.3 Architektura Klastra

Najbardziej podstawowym elementem klastra Kubernetes są *węzły* (ang. *nodes*). Węzeł reprezentuje maszynę roboczą, fizyczną lub wirtualną, na której uruchomiona została skonteneryzowana aplikacja. Każdy klaster musi posiadać minimalnie jeden węzeł [56]. Na każdym węźle uruchamiane są określone składniki, umożliwiające uruchamianie kontenerów w ramach podów (niepodzielnych elementów, reprezentujących jeden lub wiele kontenerów) [56]:

- **kubelet** - proces odpowiedzialny za komunikację pomiędzy warstwą sterowania Kubernetes (ang. *control plane*) a węzłami. Zarządza on kontenerami działającymi na danej maszynie, odpowiada za uruchamianie kontenerów w ramach poda.
- **kube-proxy** - proxy sieciowe, uczestniczące w tworzeniu serwisu. Utrzymuje ono reguły sieciowe na węźle, dzięki czemu sieci na zewnątrz i wewnątrz klastra mogą komunikować się z podami.
- **Container runtime** (proces wykonawczy kontenera) - oprogramowanie służące do uruchamiania kontenerów (np. Docker). Pobiera ono obraz kontenera z repozytorium, rozpakowuje go i uruchamia.

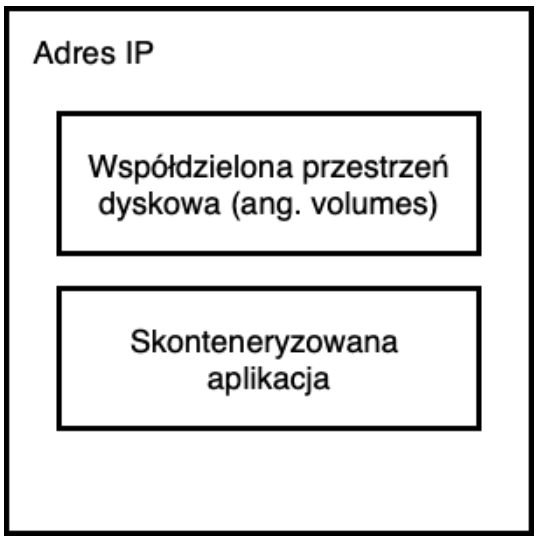


Rysunek 2.20: Węzeł Kubernetes (źródło: opracowanie własne z wykorzystaniem programu draw.io).



Rysunek 2.21: Składniki węzłów Kubernetes (źródło: opracowanie własne z wykorzystaniem programu draw.io).

Na węzłach rozmieszczane są *pody* - abstrakcyjne, niepodzielne obiekty Kubernetes. Reprezentują one jeden lub wiele kontenerów wraz z ich wspólnymi zasobami, takimi jak: współdzielona przestrzeń dyskowa (ang. *volumes*), zasoby sieciowe (adres IP klastra), informacje służące do uruchamiania każdego z kontenerów (wersja obrazu, numery portów). Kubernetes tworzy pody zawierające kontenery - nie same kontenery bezpośrednio w klastrze. Każdy pod pozostaje związany z węzłem na którym został uruchomiony aż do jego wyłączenia lub usunięcia. W przypadku awarii, pod zostaje uruchomiony na innym, działającym węźle [56].

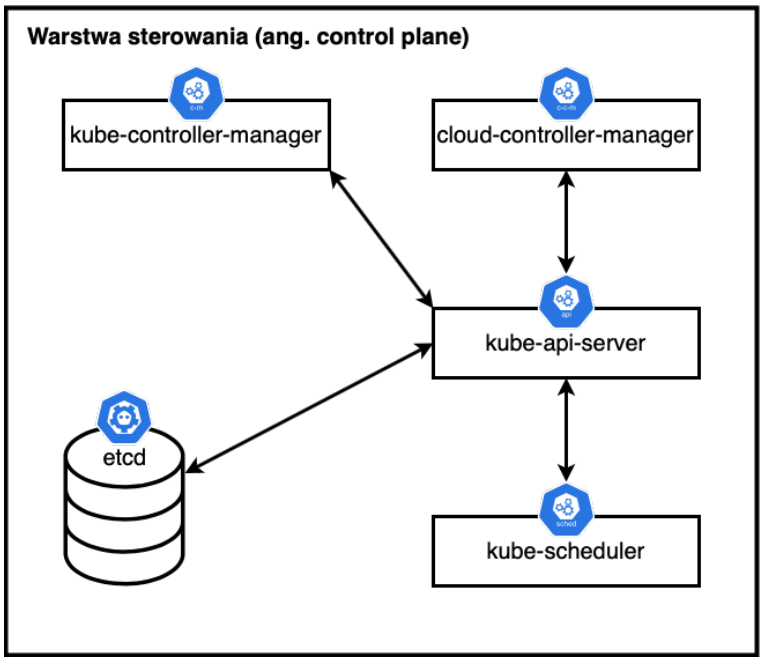


Rysunek 2.22: Pod Kubernetes (źródło: opracowanie własne z wykorzystaniem programu draw.io).

Warstwa sterowania (ang. *control plane*) zarządza węzłami i podami wewnątrz klastra. Automatycznie zleca ona uruchamianie podów na różnych węzłach, opierając się na dostępnych na każdym z nim zasobach. W środowisku produkcyjnym warstwa sterowania rozłożona jest zazwyczaj na kilka maszyn, a klaster uruchomiony jest na wielu

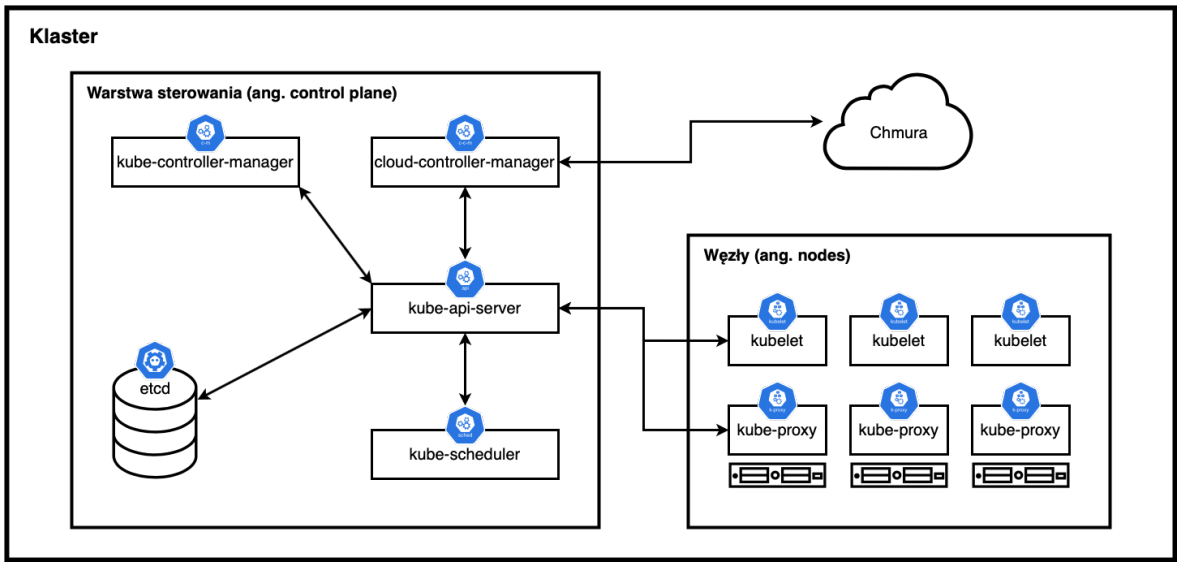
węzłach. Zapewnia to większą niezawodność oraz odporność na awarie. Warstwa sterowania dzieli się na komponenty, które podejmują decyzje dotyczące klastra oraz reagują na zdarzenia w nim zachodzące (zarówno uruchamianie nowych podów jak i awarie) [56]. Wyróżnia się:

- **kube-apiserver** - odpowiedzialny za udostępnianie API Kubernetes. Umożliwia on komunikację pomiędzy elementami klastra.
- **kube-scheduler** - śledzi tworzenie nowych podów i przypisuje im węzły, na których powinny zostać uruchomione.
- **kube-controller-manager** - uruchamia kontroler - pętlę nieustannie monitorującą stan klastra. Wprowadza ona zmiany w zależności od różnic zaistniałych względem stanem żądanym, opisanym w manifeście, a stanem faktycznym. Controler manager odpowiedzialny jest za ogólne utrzymanie klastra, replikację jego komponentów oraz zwalczanie awarii. Każdy kontroler to oddzielny proces, jednak wszystkie one skompilowane są do jednego programu binarnego i uruchamiane w formie jednego procesu. Wyróżniamy: node controller (odpowiedzialny za węzły klastra), replication controller (odpowiedzialny za pody), endpoints controller (łączący serwisy i pody), service account & token controllers (tworzący domyślne konta i tokeny dostępu API).
- **cloud-controller-manager** - zarządza usługami realizowanymi po stronie chmur obliczeniowych. Umożliwia on połączenie klastra z API dostawcy usług chmurowych oraz rozdziela składniki operujące na platformie chmurowej od tych, które dotyczą wyłącznie klastra.
- **etcd** - magazyn przechowujący wszystkie dane o klastrze Kubernetes.



Rysunek 2.23: Warstwa sterowania Kubernetes (źródło: opracowanie własne z wykorzystaniem programu draw.io na podstawie [56]).

Ogólna architektura klastra, zawierająca opisane powyżej elementy, prezentuje się następująco:



Rysunek 2.24: Architektura klastra Kubernetes (źródło: opracowanie własne z wykorzystaniem programu draw.io na podstawie [56]).

2.4 Tworzenie infrastruktury

2.4.1 Definicja infrastruktury

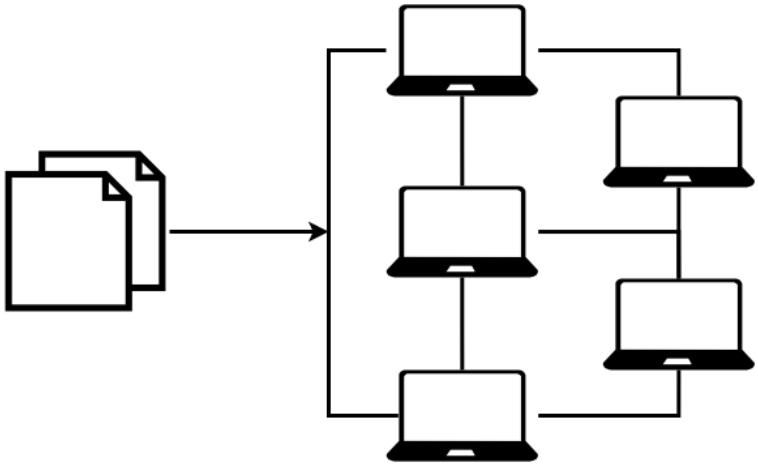
Infrastruktura jest pojęciem bardzo szerokim. Przede wszystkim odnosi się ono do ogółu zasobów fizycznych, wykorzystywanych w firmach lub gospodarstwach domowych. Wówczas możemy ją podzielić na *Infrastrukturę tradycyjną* (on premise), należącą do firmy/osoby fizycznej i *Infrastrukturę Chmurową* (IaaS) - szeroko opisaną w rozdziale 2.2. Infrastruktura to również *software* - oprogramowanie, działające na danych zasobach. Tworzenie infrastruktury, zarówno w przypadku projektów dla ogromnych korporacji jak i na potrzeby mniejszych celów (podobnie jak w niniejszej pracy) opiera się na stworzeniu środowiska, na którym dana aplikacja (lub szereg aplikacji) będzie funkcjonować.

Definicja Infrastruktury dostępna na stronie firmy *Devire* [83]:

„*Infrastruktura informatyczna to cały hardware oraz software, czyli sprzęt i oprogramowanie, które współgrają ze sobą. Dzięki temu tworzą całokształt rozwiązań sprzętowo-programowych i organizacyjnych, stanowiących podstawę wdrożenia i eksploatacji systemów informatycznych wspomagających zarządzanie przedsiębiorstwami czy instytucjami. Wszystko stanowi spójną całość, która jest wymagana do rozwoju, testowania, monitorowania i obsługi usług informatycznych. Odpowiedni sprzęt odpowiadający za poprawne działanie systemu operacyjnego, baz danych czy aplikacji, a wszystko połączone ze sobą siecią zapewnia poprawne działanie całego środowiska. Główne elementy infrastruktury to: serwer, system operacyjny, baza danych, sieć.*”

2.4.1.1 Infrastruktura jako kod

Infrastruktura jako kod (ang. *Infrastructure as Code* (*IaC*)) to sposób zarządzania infrastrukturą informatyczną (maszynami wirtualnymi, sieciami itp.) z wykorzystaniem kodu (skryptów). Podejście IaC tworzy takie samo środowisko za każdym razem, gdy zostanie wykorzystane i jest szczególnie popularne w kulturze *DevOps*[69]. IaC opiera się zatem na pisaniu skryptów i aplikacji wspomagających proces konfiguracji infrastruktury, aby uniknąć każdorazowego tworzenia jej manualnie. Słowo „kod" zawarte w nazwie odnosi się do jednego, spójnego skryptu (napisanego w języku deklaratywnym), służącego do automatyzacji tworzenia pożądanej infrastruktury [84].



Rysunek 2.25: Infrastruktura jako kod (IaC) (źródło: opracowanie własne z wykorzystaniem programu draw.io).

Pojęcia kluczowe do zrozumienia IaC:

- **DevOps** - połączenie dwóch obszarów firm: zespołu odpowiedzialnego za rozwój (*developing*) oprogramowania (Dev) oraz zespołu odpowiedzialnego za operacje (*operations*), pracę z istniejącą platformą (Ops). Jest to zarówno kultura, metodologia pracy jak i nazwa stanowiska. W znaczeniu metodologii DevOps kładzie nacisk na współpracę powyższych jednostek w celu uzyskania wyższej jakości dostarczanych produktów oraz usprawnienia procesów. DevOps wiążę się również ze zwinnym modelem pracy (*Agile*) - udostępnianiem produktu (działającej infrastruktury/oprogramowania) w działających fragmentach w określonych odstępach czasu (*sprintach*) [17].
- **CI/CD** - (ang. *Continuous Integration/Continuous Delivery* - Ciągła Integracja/Ciągłe Wdrażanie) praktyka wykorzystywania w celu budowania i rozwoju projektów informatycznych. „Ciągła Integracja" oznacza tworzenie i testowanie części tworzonego oprogramowania w krótkich odstępach czasu. Jego kod umieszcza się we wspólnym, zdalnym repozytorium (GIT) i niejednokrotnie automatyzuje z wykorzystaniem narzędzi takich jak *Jenkins*. „Ciągłe dostarczanie" odnosi się do zestawu procesów dostarczających oprogramowanie w sposób zautomatyzowany na dane środowisko (np. testowe lub produkcyjne). CI/CD stanowi podstawę pracy DevOps [11].

IaC, podobnie jak Kubernetes, opiera się na stosowaniu deklaratywnych plików tam, gdzie jest to możliwe. Określają one czego wymaga środowisko (np. wersję, konfigurację serwera), nie określają natomiast sposobu jego instalowania. W zależności od platformy na której powstaje dana infrastruktura, pliki te mogą różnić się formatem (np. YAML, JSON, XML) [69]. Istnieje szereg narzędzi usprawniających tworzenie IaC i jej późniejsze wykorzystywanie, np. *Ansible, Puppet*, również *Docker* [84].

Główne zalety stosowania IaC [84]:

- umożliwienie zespołom DevOps budowania środowisk jak najbardziej zbliżonych do środowisk produkcyjnych i testowania na nich aplikacji na wczesnych etapach jej tworzenia;
- tworzenie nowych, stabilnych środowisk w szybki sposób oraz na dużą skalę;
- kod infrastruktury stanowi również rodzaj dokumentacji dla aplikacji na niej wdrażanych - opisuje szczegółowo wersje oraz narzędzia i jest dostępny we współdzielonym repozytorium;
- jeśli skrypty są napisane w sposób prawidłowy, infrastruktura może powstać zarówno w chmurze (prywatnej, publicznej, hybrydowej) jak i na prywatnym serwerze;
- rejestrowanie zmian dzięki zastosowaniu kontroli wersji w kodzie infrastruktury, umożliwiające sprawne przywrócenie ostatniej działającej wersji w razie awarii.

2.4.2 Modele architektury oprogramowania

Pojęcie *Architektury Oprogramowania* (ang. *Software Architecture*) odnosi się do sposobu zorganizowania systemu (jego komponentów, środowiska pracy i reguł), określających sposób jego tworzenia. Ma on zapewniać niezawodność, elsatyczność, bezpieczeństwo oraz skalowalność tworzonego systemu, a także relacje i interakcje między poszczególnymi jego elementami [95].

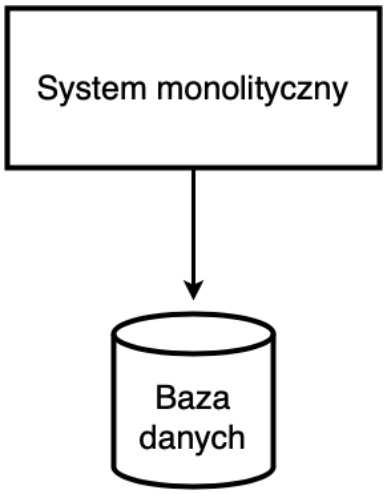
Wybór modelu architektury oprogramowania zależy od poziomu złożoności tworzonego systemu oraz pełnionej przez niego funkcji. Jest to decyzja kluczowa podczas planowania tworzenia infrastruktury, z racji zasadniczych różnic pomiędzy poszczególnymi modelami. Ich podstawowe definicje zostały opisane w poniższych rozdziałach.

2.4.2.1 Systemy monolityczne

Systemy monolityczne (systemy oparte na architekturze monolitycznej) są modelem tradycyjnym. Polegają one na umieszczaniu wszystkich funkcji danego oprogramowania w jednej, wspólnej bazie kodu i wrażaniu ich w formie pojedynczego pliku [2]. Inaczej mówiąc, gdy wszystkie funkcje systemu instalowane są wspólnie, system można uznać za monolityczny [75]. W ten sposób tworzono oprogramowania przez wiele lat [2].

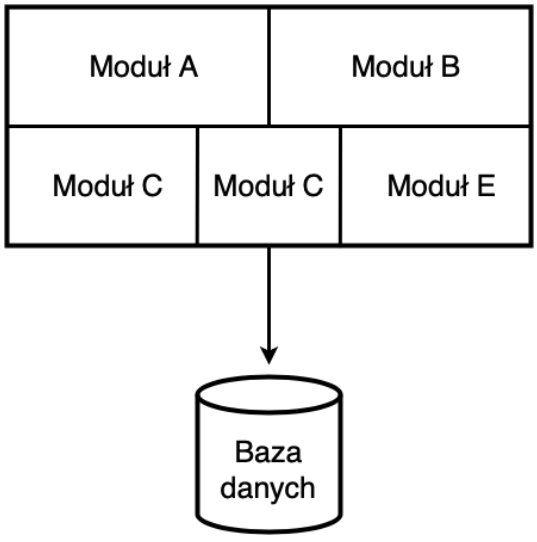
Systemy monolityczne podzielić można na trzy główne rodzaje, scharakteryzowane poniżej [75].

- **Jednoprocesowe systemy monolityczne** - najbardziej powszechny rodzaj systemów monolitycznych. Cały kod systemu instalowany jest w formie jednego procesu, chcoć można na jego podstawie uruchomić wiele instancji (na przykład w celu zapewnienia lepszego skalowania) [75].



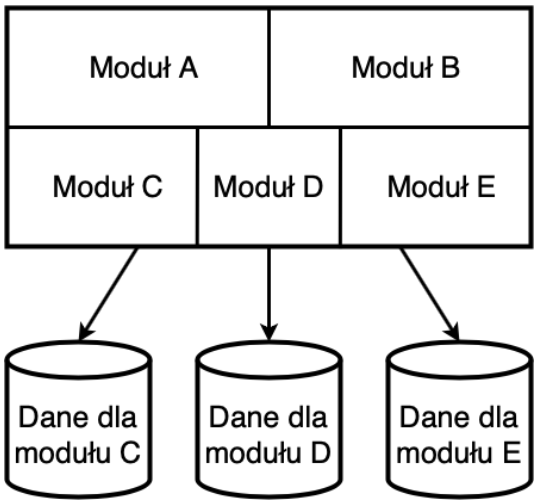
Rysunek 2.26: Jednoprocesowy system monolityczny (źródło: opracowanie własne z wykorzystaniem programu draw.io na podstawie [75]).

Odmianą jednoprocesowych systemów monolitycznych jest **modułowy system monolityczny**. Wówczas proces podzielić można na rozwijane niezależnie moduły, połączone na potrzeby instalacji. Jest to rozwiązanie umożliwiające uniknięcia wyzwań związanych z tworzeniem systemu w modelu mikrousług, przy jednoczesnym zachowaniu zalet równoległej pracy nad wieloma modułami. Posiada jednak wady. Bazy danych nie posiadają zazwyczaj dekompozycji stosowanej na poziomie kodu, co może powodować wiele problemów w przypadku tworzenia systemów na dużą skalę [75].



Rysunek 2.27: Modułowy system monolityczny (źródło: opracowanie własne z wykorzystaniem programu draw.io na podstawie [75]).

Idea modułowego monolitu była przez lata rozwijana, co doprowadziło do stworzenia rozwiązań (przedstawionych na rysunku poniżej), w których baza danych dzielona jest analogicznie wobec modułów. Wymagają one jednak ogromnej wprawy w konfiguracji, nawet bez potrzeby modyfikacji kodu [75].



Rysunek 2.28: Modułowy system monolityczny z podzieloną bazą danych (źródło: opracowanie własne z wykorzystaniem programu draw.io na podstawie [75]).

- **Rozproszone systemy monolityczne** - obejmują wiele usług, jednak w przeciwieństwie do architektury mikrousług - są one instalowane jako całość. Jest to architektura o ścisłym powiązaniu komponentów, w której wdrożenie prostej modyfikacji o pozornie lokalnym zasięgu uszkodzić może pozostałe części systemu [75]. *„(...) rozproszone systemy monolityczne mają wady systemów rozproszonych, a także wady jednoprocesowych systemów rozproszonych, nie posiadając przy tym wystarczająco dużo zalet żadnych z nich."* [75]

- **Systemy typu czarna skrzynka od niezależnych dostawców** - niektóre programy od niezależnych dostawców traktować można jak systemy monolityczne, poddawane dekompozycji w ramach migracji [4], np. system do obsługi listy płac, system kadrowy. Wówczas nie ma możliwości modyfikacji kodu opracowanego przez dostawcę [75]. Tego typu oprogramowania to na przykład produkty typu SaaS (2.2.3.3).

Główną wadą systemów monolitycznych jest ilość problemów, jakie generuje wprowadzenie zmian w kodzie. Komponenty w aplikacji są ze sobą ściśle powiązane, zatem zmiany wpływają na całość kodu. Pojedyncze błędy jakie mogą wyniknąć w procesie nanoszenia zmian, wpływają wówczas na cały system, ponadto każda zmiana wymusza ponowne wdrożenie całej aplikacji. W przypadku rozbudowanych systemów proces ten może zająć wiele dni. Aplikacje monolityczne są trudne do skalowania - wymagają skalowania całej aplikacji [2]. Nie wiadomo również, w przypadku pracy zespołu, kto

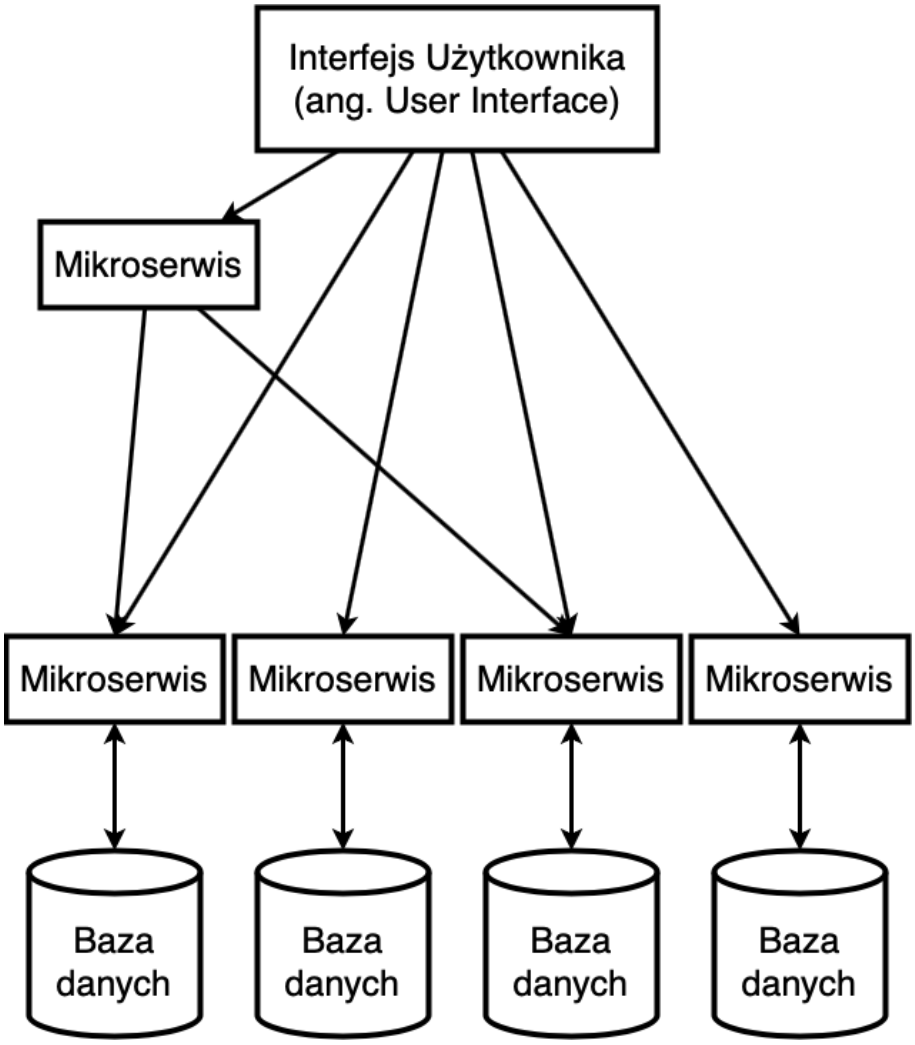
[4]Migracja - *„Migracja danych jest nieodzownym elementem migracji systemów i aplikacji. Polega na przeniesieniu danych z jednego systemu informatycznego do drugiego, ze starej wersji programu do nowej. Szybka i sprawna migracja danych z wcześniej wykorzystywanych w firmie aplikacji daje możliwość zachowania ciągłości pracy i realizacji podstawowych celów biznesowych."* [89]

jest właścicielem poszczególnych komponentów i kto podejmuje decyzje. Prowadzi to do *„konfilktów w procesie dostarczania”* [75].

Zaletą systemów monolitycznych jest prostota ich instalacji oraz uproszczenie monitorowania. Ponadto kod jest gotowy do ponownego użycia - w systemie rozproszonym należy podjąć decyzję, czy skopiować kod, podzielić biblioteki czy przenieść wspólne funkcje do usługi [75].

2.4.2.2 Systemy mikrousługowe

Mikrousługi (ang. *microservices*) to niezależnie instalowane usługi, modelowane na podstawie dziedziny biznesowej której dotyczy tworzony system [75]. Są one niewielkie, posiadają własne cykle życia i współpracują wzajemnie ze sobą, tworząc nowe podejście do systemów rozproszonych [74]. Ich komunikacja następuje za pośrednictwem sieci. *„Architektura oparta na mikrousługach wykorzystuje wiele współdziałających ze sobą mikrousług.”* [75]. Jest to rodzaj *SOA - Service-oriented architecture* - architektury opartej na usługach, w której kluczowa jest możliwość niezależnej instalacji usług [75].



Rysunek 2.29: Architektura mikrousług (źródło: opracowanie własne z wykorzystaniem programu draw.io na podstawie [75]).

W przypadku ogromnych systemów o wielu funkcjonalnościach, w ramach systemu monolitycznego tworzono moduły. Podejście to umożliwiało odnalezienie miejsca w kodzie, odpowiedzialnego za daną funkcję, w którym powinny zostać naniesione ewentualne zmiany. Mikrousługi umożliwiają *spójność* - grupowanie powiązanych ze sobą fragmentów kodu oraz *zasadę pojedynczej odpowiedzialności* - *„Pogrupuj ze sobą te elementy, które będą się zmieniać z tego samego powodu i rozdziel te, które zmieniają się z różnych powodów”* (Robert C. Martin) [74]. Pozwala to uniknąć nadmiernej rozbudowy kodu.

Najważniejsze korzyści wynikające z zastosowania architektury mikrousług [74]:

- **niejednorodność technologii** - każda mikrousługa, działająca w ramach spójnego systemu, może korzystać z innej technologii (np. różnych języków programowania, różnego typu baz danych i magazynów obiektów). Umożliwia to wykorzystywanie odpowiednich technologii wobec konkretnej funkcji jaką pełni dana mikrousługa. Nie ma potrzeby standaryzacji narzędzi na potrzeby całego systemu;
- **odporność na błędy** - awaria dotycząca jednej składowej systemu nie dotyczy jego całości i nie ma ryzyka jej rozprzestrzenienia się. Mikrousługi posiadają swoje granice, izolujące wszelkiego rodzaju błędy i problemy;
- **skalowalność** - możliwość skalowania wyłącznie tych mikrousług, które tego wymagają;
- **łatwość wdrażania** - wprowadzanie zmian i ich wdrażanie w pojedynczej mikrousłudze następuje niezależnie od reszty systemu. Ponadto zmiany mogą zostać cofnięte. Dzięki temu udostępnianie nowej funkcjonalności jest znacznie prostsze, z czego korzyści czerpie np. Amazon lub Netflix;
- **interoperatywność** - możliwość wielokrotnego wykorzystania danej funkcjonalności - w różny sposób i w różnych celach;
- **wymienność** - koszty zastąpienia jednej mikrousługi inną lub jej całkowite usunięcie są znacznie mniejsze niż w przypadku wymiany systemów monolitycznych. Wiąże się z tym również mniejsze ryzyko.

2.4.3 Model Cloud Native

W 2015 r. powstała organizacja *Cloud Native Computing Foundation* (CNCF) [31], mająca na celu *„wspieranie społeczności wokół konstelacji wysokiej jakości projektów, które orkiestrują kontenery jako część architektury mikrousług”* [4]. Jest to część organizacji *Linux Foundation*. CNCF zrzesza programistów, użytkowników końcowych i dostawców chmur obliczeniowych. Kubernetes jest jednym z jej projektów [4].

Samo działanie w chmurze i wykorzystanie konteneryzacji nie czyni aplikacji natywną (ang. *cloud native application*), choć są to ważne aspekty. Poniżej przedstawione zostały podstawowe cechy charakterystyczne dla systemów *cloud native*, wyszczególnione w literaturze (John Arundel, Justin Domingus *„Kubernetes - rozwiązania chmurowe w świecie DevOps”* [4]).

- **Zdolność do automatyzacji** - Kubernetes zapewnia standardowe interfejsy aplikacji, wdrażanych i zarządzanych przez maszyny, tym samym zwalniając z tego obowiązku programistów.
- **Wszechobecność i elastyczność** - mikrousługi w kontenerach można z łatwością przenosić pomiędzy węzłami oraz klastrami, ponieważ są one oddzielone od zasobów fizycznych.
- **Odporność i skalowalność** - aplikacje cloud native są rozproszone, zapewniając wysoką dostępność dzięki redundancji i płynnemi rozkładowi.
- **Dynamiczność** - orkiestrator kontenerów umożliwia maksymalnie efektywne wykorzystanie dostępnych zasobów oraz aktualizację usług bez zmniejszania ruchu.
- **Obserwowalność** - zdolność do monitorowania, rejestrowania i śledzenia zachodzących zmian, problematyczna w przypadku systemów rozproszonych.
- **Rozproszenie** - rozproszony, zdecentralizowany charakter wykorzystywanej chmury obliczeniowej. Aplikacje cloud native składają się z wielu współpracujących mikrousług.

Choć systemy monolityczne są znacznie prostsze do zrozumienia i monitorowania, jednak nie zapewniają funkcjonalności kluczowych dla złożonych systemów. Aplikacje natywne, będące systemami rozproszonymi, są z natury złożone i równie podatne na awarie. Oferują one jednak więcej decydujących zalet, jak zostało to opisane w rozdziale 2.4.2.2. Mimo to warto zaznaczyć, że aplikacje monolityczne również mogą zostać uruchomione w chmurze - dzięki wykorzystaniu konteneryzacji. Jest to krok na drodze do migracji od monolitu do mikrousług [4].

Rozdział 3

Wykorzystane narzędzia

3.1 Aplikacja

Symulacja eksperymentu Rutherforda napisana została w języku programowania Python [85]. Jej kod (`rutherford-scattering.py`), wraz z plikami konfiguracyjnymi infrastruktury (opisanymi w rozdziale 5), umieszczony został w zdalnym repozytorium kodu na platformie GitHub [36]. Repozytorium to dostępne jest pod adresem: [106].

W procesie tworzenia kodu aplikacji powstało wiele jego wersji, różniących się strukturą oraz wykorzystanymi bibliotekami. Początkowo symulacja napisana została z wykorzystaniem biblioteki *VPython* [97] (3.1.1.1), tak aby eksperyment Rutherforda przebiegał w trzech wymiarach. Niestety już na etapie konteneryzacji generowała ona wiele aspektów ciężkich do rozwikłania, co szerzej opisane zostało w rozdziale 4.1.

Końcowa wersja aplikacji napisana została w bibliotece *Matplotlib* [65], co umożliwiło jej skuteczną konteneryzację oraz wdrożenie w chmurze obliczeniowej. Jej kod szczegółowo opisany został w rozdziale 4.2. Jest ona mniej spektakularna, gdyż symuluje eksperyment jedynie w dwóch wymiarach, jednak prawidłowo działa w stworzonym środowisku i wyświetla się za pośrednictwem okna przeglądarki danego użytkownika.

3.1.1 Python

Rysunek 3.1: Python [86].

Python to język programowania ogólnego przeznaczenia, powstały na początku lat dziewięćdziesiątych. Swoją powszechność zawdzięcza mnogości oferowanych bibliotek, dzięki którym możliwe jest jego wszechstronne wykorzystanie - również w nauce. Obecnie jest on standardem w zastosowaniach analizy danych (ang. *data science*) oraz uczenia maszynowego (ang. *machine learning*) [35]. W Fizyce Python wykorzystywany jest przede wszystkim do tworzenia symulacji - tak jak w niniejszej pracy. Jest narzędziem modelowania komputerowego, które na poziomie podstawowych aplikacji nie wymaga specjalistycznej wiedzy. Ponadto oferuje on szeroką dokumentację umożliwiającą jego skuteczne wykorzystanie.

3.1.1.1 VPython

VPython to jedna z bibliotek języka Python. Powstała ona w dwutysięcznym roku jako moduł graficzny 3D dla Pythona o nazwie *„Visual"*. Nazwa *VPython* wynika z połączenia nazwy modułu i języka. Biblioteka ta służy do generowania obiektów w trzech wymiarach i obsługuje algebrę wektorową, co czyni ją szczególnie przydatną w procesie tworzenia symulacji naukowych. W trakcie działania stworzonej aplikacji, użytkownik ma możliwość obracania, powiększania oraz pomniejszania symulacji [97].

Klasyczna biblioteka VPython (VPython 6) nie jest już wspierana przez sam język, wobec czego funkcjonuje za pośrednictwem strony *Glowscript* [39]. Również VPython 7, z możliwością instalacji (przy pomocy komendy `pip install vpython`) wymaga połączenia z Glowscript - co zostało opisane w rozdziale 4.1.3.

3.1.1.2 Matplotlib

Matplotlib to biblioteka języka Python, popularna w środowisku naukowym. Umożliwia tworzenie statycznych, animowanych i interaktywnych wizualizacji, przypominających te stworzone w programie MATLAB [1] [64]. Matplotlib został wykorzystany m.in. do stworzenia pierwszego obrazu czarnej dziury, wraz z biblioteką *NumPy* [29].

3.1.1.3 NumPy

NumPy to biblioteka języka Python, której nazwa stanowi skrót od słów *„Numerical Python"*. Powstała w 2005 roku, stworzona przez Travisa Oliphanta jako projekt Open Source. NumPy, podobnie jak Matplotlib, jest wykorzystywana głównie do obliczeń naukowych. Umożliwia ona pracę z tablicami, oferując zestaw narzędzi matematycznych, logicznych i sortujących, m.in.: dyskretne transformaty Fouriera, podstawową algebrę liniową, symulacje losowe i operacje statystyczne [77].

3.1.1.4 Flask

Flask to mikro-framework (platforma programistyczna umożliwiająca tworzenie aplikacji), umożliwiający tworzenie aplikacji webowych przy pomocy języka Python. Nie wymaga on określonych narzędzi ani bibliotek, nie oferuje bazy danych, sprawdzania formularzy ani pozostałych komponentów typowych dla rozbudowanych stron internetowych. Mimo to obsługuje on rozszerzenia, które podobne funkcjonalności mogą zapewnić. Umożliwiają one integrację z wybranymi przez twórcę aplikacji bazami danych, technologie uwierzytelniania itp. Twórcy Flaska dążą ku temu, aby rdzeń aplikacji webowej był prosty, ale rozszerzalny [30].

[1] MATLAB - platforma programistyczna i obliczeniowa, wykorzystywana przez inżynierów i naukowców do analizowania danych, opracowywania algorytmów i tworzenia modeli [64].

3.1.2 GitHub

GitHub umożliwia zarządzanie *repozytoriami Git*. *Git* to rozproszony system kontroli wersji, ułatwiający tworzenie złożonych projektów w obrębie zespołów ogólnopojętych informatyków. Dzięki niemu możliwe jest śledzenie historii zmian, praca nad jednym plikiem bez jego nadpisywania, sprawdzanie statusu pracy oraz przywracanie wcześniejszych wersji projektu. Jest to zatem narzędzie bezpiecznej, równoległej pracy wielu osób, szczególnie istotne w procesie tworzenia IaC (2.4.1.1) [99].

Rysunek 3.2: GitHub [38].

GitHub to usługa hostingu, działająca w oparciu o chmurę. Repozytorium GitHub, reprezentujące dany projekt, tworzone jest w ramach konta - indywidualnego lub organizacyjnego. Może ono być prywatne - widoczne wyłącznie dla jego twórcy i wybranych osób, którym zostanie udostępione lub publiczne - dostępne dla wszystkich. W repozytorium znajdują się wszystkie katalogi oraz pliki należące do danego projektu [99].

GitHub, poza możliwością wspólnej pracy nad projektem, jest również rodzajem portfolio dla twórcy repozytoriów. Ponadto umożliwia on stworzenie bezpiecznej, zdalnej kopii indywidualnej pracy w chmurze oraz dzielenie się kodem źródłowym z innymi. W tym celu został on wykorzystany w niniejszej pracy [99].

3.2 Infrastruktura

Infrastruktura pracy powstała w oparciu o konteneryzację oraz chmurę obliczeniową. Do konteneryzacji wykorzystano opisane w rozdziale 2.3 narzędzia - Docker oraz Kubernetes. Wybrana chmura obliczeniowa to Microsoft Azure, z uwagi na możliwość posiadania konta w subskrypcji studenckiej (oferującego wiele darmowych usług oraz środki pieniężne do wykorzystania na start) oraz oferowany serwis - *Azure Kubernetes Service*, szczegółowo opisany w podrozdziale 3.2.2.

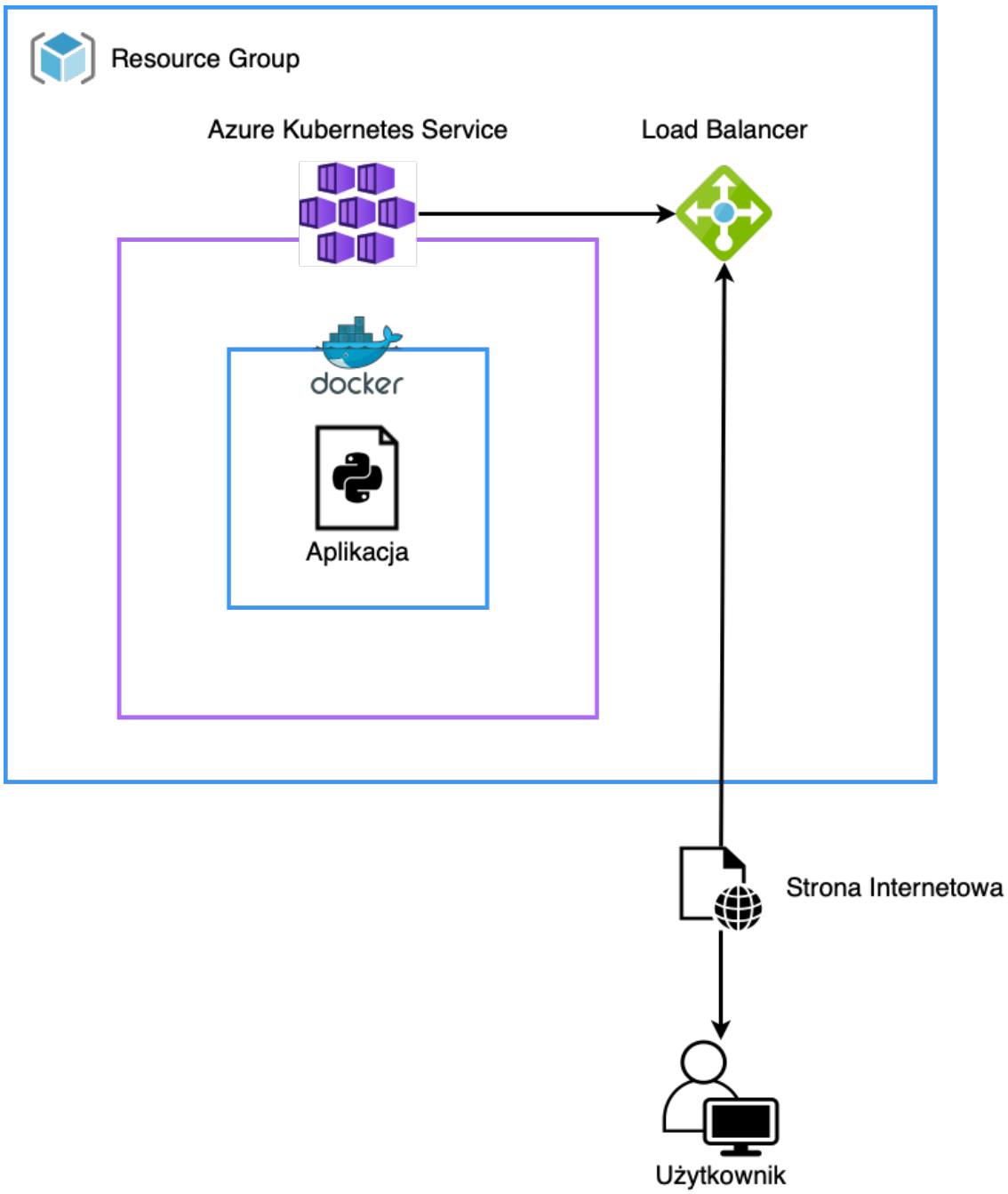
Zagadnienie konteneryzacji jest w niniejszej pracy ściśle powiązane z chmurą obliczeniową. Istnieje wiele dróg korzystania z Kubernetes. Pierwszą z nich jest hosting - samodzielna instalacja i konfiguracja Kubernetes na komputerach które posiadamy lub kontrolujemy. Rozwiązanie to można zastosować w przypadku maszyn wirtualnych. Była to jedna z możliwości branych pod uwagę w trakcie tworzenia pracy. Początkowo miała się ona opierać na stworzeniu trzech maszyn wirtualnych *EC2* w chmurze Amazon Web Services i zainstalowaniu na nich w sposób ręczny Kubernetes. Wówczas jedna z nich staje się warstwą sterowania, a pozostałe dwie węzłami roboczymi. Koszty utrzymania klastra są w tym przypadku niewielkie, ale wymaga to wiele pracy oraz

wiedzy, która umożliwi poprawną konfigurację klastra oraz jego późniejsze utrzymanie. Kubernetes jest oprogramowaniem niezwykle złożnym, stąd jego ogromne możliwości, jednak kompleksowe zrozumienie wszystkich elementów niezbędnych do jego prawidłowego funkcjonowania stanowi duże wyzwanie i wymaga wiele czasu.

Z uwagi na ten fakt dostawcy chmur oferują usługi zwalniające twórcę klastra z podobnej odpowiedzialności - *zarządzane usługi Kubernetes* [4]. Odciążają one twórców z obowiązków związanych z samodzielnym uruchamianiem i konfiguracją Kubernetes, szczególnie warstwy sterowania. Oznacza to, że płacimy dostawcy chmury publicznej (np. Microsoft Azure, jak w przypadku niniejszej pracy) za uruchomienie i utrzymanie klastra, bazując na określonej konfiguracji jaką chcemy osiągnąć. Definiujemy ilość węzłów jaką chcemy stworzyć, ilość kopii (replik) skonteneryzowanej aplikacji, minimalne oraz maksymalne zużycie zasobów sprzętowych, na których działa Kubernetes, port na którym aplikacja zostanie udostępiona w sieci itd. Warstwa sterowania zarządzana jest przez dostawcę chmury. Jest to niesamowicie popularne rozwiązanie również wśród osób zaawansowanych. Podobne serwisy ułatwiają udostępnianie skonteneryzowanych aplikacji w Internecie oraz oferują dodatkowe, powiązane usługi, np. monitoring, tworzenie alertów i tym podobne. Opisane usługi Kubernetes są droższe w przypadku pojedynczego klastra, jednak na dużą skalę, np. w korporacjach korzysta się niemalże wyłącznie z podobnych rozwiązań.

Niniejsza praca ostatecznie powstała z wykorzystaniem *Azure Kubernetes Service* z uwagi na ogromną łatwość tworzenia klastra z jego wykorzystaniem oraz doświadczenie, jakie nabyłam pracując z tą usługą na co dzień.

3.2.1 Model architektury

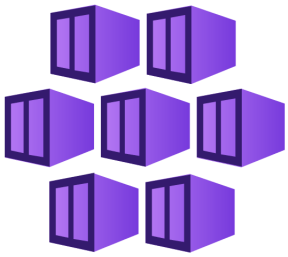


Rysunek 3.3: Model infrastruktury pracy (źródło: opracowanie własne z wykorzystaniem programu draw.io).

Architektura pracy przedstawiona została na rysunku 3.3. Infrastruktura składa się z następujących elementów: aplikacji napisanej w języku Python (4.2), oprogramowania Docker, które umożliwia stworzenie obrazu aplikacji oraz mikrousług chmury Microsoft Azure - Azure Kubernetes Service (3.2.2) i Azure Load Balancer (3.2.3). Mikrousługi te zgromadzone są w *Resource Group* [10]. Zawiera ona *pokrewne zasoby dla rozwiązania platformy Azure* [10] - umożliwia zarządzanie grupami zasobów w obrębie chmury Microsoft Azure.

3.2.2 Azure Kubernetes Service

Azure Kubernetes Service (*AKS*) [5] to zarządzana przez Microsoft Azure usługa Kubernetes. Umożliwia ona tworzenie klastrów za pomocą interfejsu webowego lub narzędzia wiersza poleceń - *Azure az*, wykorzystanego w niniejszej pracy. Korzystając z AKS twórca nie posiada bezpośredniego dostępu do warstwy sterowania (węzła *master*), ma natomiast dostęp do węzłów roboczych (*worker node*). Cena utrzymania klastra zależy przede wszystkim od ilości stworzonych węzłów [4].

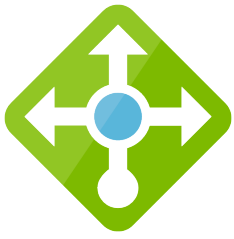


Rysunek 3.4: Azure Kubernetes Service (AKS) [6].

Klaster, powstały w usłudze Azure Kubernetes Service, składa się w niniejszej pracy z trzech węzłów - dwóch *worker nodes* oraz jednego *master node*, będącego zarządzaną przez Azure warstwą sterowania. Skonteneryzowana aplikacja, zgodnie z zadaną konfiguracją, posiada dwie repliki (dwa pody). Znajdują się one na wybranych przez Kubernetes węzłach, w sposób zapewniający jak największe bezpieczeństwo w razie awarii.

3.2.3 Azure Load Balancer

Azure Load Balancer [7] umożliwia udostępnienie aplikacji pod określonym adresem w Internecie, a także równoważy jej obciążenie w przypadku wielu korzystających z niej użytkowników jednocześnie. Równoważenie obciążenia obsługuje zarówno ruch wewnętrzny jak i zewnętrzny.



Rysunek 3.5: Azure Load Balancer [7].

W przypadku niniejszej pracy Load Balancer pośredniczy pomiędzy użytkownikiem (osobą chcącą wyświetlić aplikację webową) a Azure Kubernetes Service, na którym ta aplikacja się znajduje. Jest on odpowiedzialny w głównej mierze za umożliwienie użytkownikowi dostępu do aplikacji z poziomu przeglądarki internetowej, przypisując jej publiczny adres IP.

3.2.4 GitHub Actions

GitHub Actions [37] to jedna z funkcjonalności platformy GitHub. Umożliwia ona automatyzację określonych zadań przy pomocy dostępnych, gotowych lub spresonalizowanych przez użytkowników skryptów. Wykonują się one w obrębie repozytorium w wyniku wystąpienia konkretnych, wybranych zdarzeń. W przypadku niniejszej pracy, każdorazowo po wprowadzeniu jakichkolwiek zmian w repozytorium (po wykonaniu komendy `git push`), następuje proces budowania nowego obrazu (Docker Image) aplikacji. Proces ten został szczegółowo opisany w rozdziale 5.1.3.

Rysunek 3.6: GitHub Actions [37].

Część II

Część praktyczna

Rozdział 4

Tworzenie aplikacji

Część praktyczna pracy podzielona została na dwa główne rozdziały: Tworzenie Aplikacji 4 oraz Tworzenie infrastruktury 5. Pierwszy z nich opisuje sposób powstawania oraz działanie symulacji eksperymentu Rutherforda, drugi natomiast przybliża metodę jej konteneryzacji i tworzenia infrastruktury w chmurze obliczeniowej.

Jak zostało to opisane w rozdziale 3.1, w procesie tworzenia pracy powstały dwie główne wersje aplikacji. Pierwsza z nich opiera się na bibliotece VPython, druga natomiast na bibliotekach Matplotlib, NumPy oraz Flask. Poniższe podrozdziały opisują proces ich powstawania oraz wyjaśniają znaczenie poszczególnych fragmentów kodu.

4.1 Aplikacja napisana w bibliotece VPython

Aplikacja napisana w bibliotece VPython powstała na podstawie filmu dostępnego na platformie YouTube: *„Rutherford Scattering (A Journey through Modern Physics 4)"*, autorstwa twórcy kanału „Let's Code Physics" [80]. Kod aplikacji, przedstawiony poniżej, znajduje się w repozytorium na platformie GitHub [106] pod nazwą `rutherford-vpython.py`.

4.1.1 Omówienie aplikacji

Zasada działania aplikacji jest bardzo prosta. W pierwszej linii kodu zaimportowana zostaje jedynie biblioteka VPython. Następnie, w linijkach 3-5 pojawiaja się definicja początkowo pustej listy, do której w późniejszej części kodu zostaną dodane wartości definiujące cząsteczki wystrzeliwane w stronę atomu złota. `dt` oraz `k` to odpowiednio krok, określający szybkość animacji oraz stała siły elektrostatycznej, która definiuje siłę interakcji. W linijce ósmej rozpoczyna się definicja parametrów początkowych atomu złota, widocznego w symulacji w formie sfery (symulacja w VPython, jak zostało to opisane w rozdziale 3.1.1.1, odbywa się w trzech wymiarach). W linijce jedenastej zdefiniowana została funkcja, która zawiera pętlę symulacji. Trwa ona do momentu wystrzelenia pięciuset cząsteczek alfa (jąder helu). Moment wystrzelenia każdej z nich jest losowy. Cząsteczki te są następnie dodawane do listy *Alphas* i stamtąd wywoływane w symulacji (generowane) w linijce dwudziestej trzeciej. `def main():` definiuje parametry funkcji `particle_definition` i przypisuje je zmiennej `Gold` wywoływanej następnie w funkcji `simulation_run(Gold)`.

Działanie symulacji opiera się na metodzie całkowania trajektorii cząsteczek alfa z wykorzystaniem schematu Eulera [63]. W linijkach 24-27, w funkcji `for a in Alphas:` znajdują się wzory określające siłę, prędkość oraz położenie cząsteczek: $F = \frac{kZZ'}{r^2}$, $v = \frac{F}{m}$, $x = x + v \cdot dt$. Metoda Eulera jest *warunkowo stabilna* - jej dokładność zależy od wielkości kroku całkowania. Aby rozwiązanie numeryczne dobrze przybliżało rzeczywistą trajektorę cząstki α, krok dt powinien być dostateczie mały. Zależność trajektorii cząsteczek alfa od kroku całkowania w symulacji, przedstawiona została w rozdziale 4.1.2.1.

Znaczenie konkretnych linii kodu zostało w nim opisane w formie komentarzy, jak poniżej:

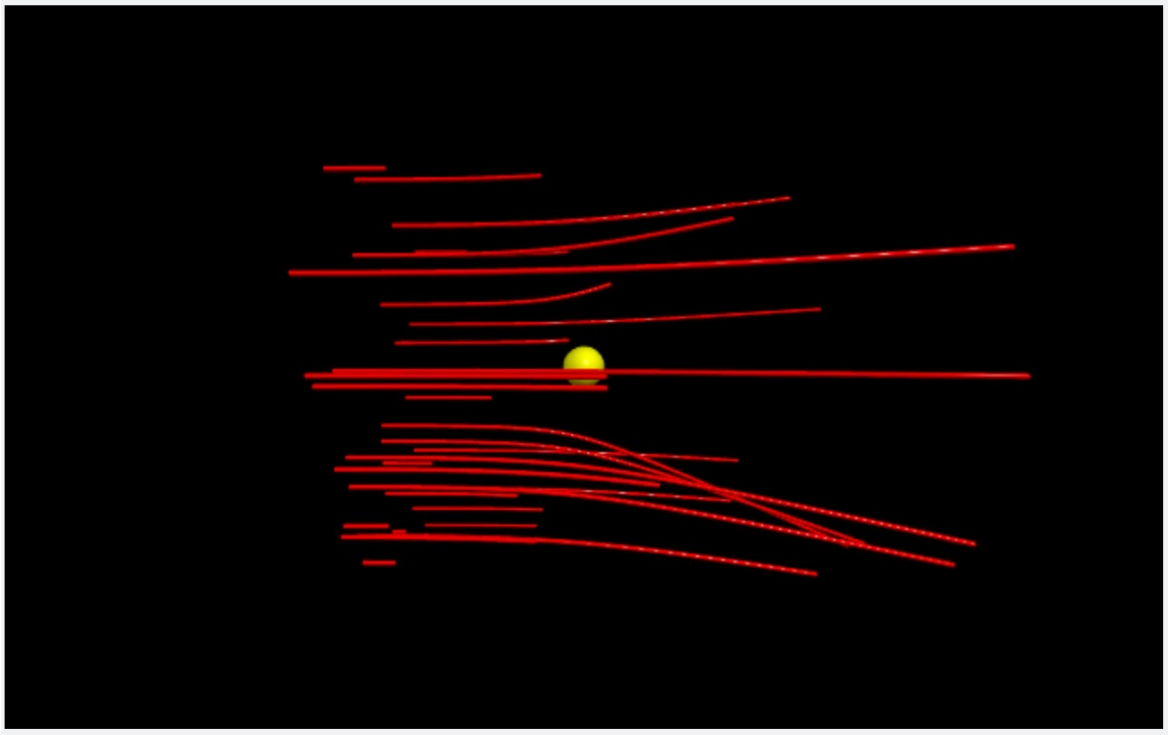
```
1 from vpython import *
2
3 Alphas = [] # pusta lista na wygenerowane cząstki
4 dt = 0.1 # krok (1/dt - szybkość animacji)
5 k = 2e-5 # stała siły elektrostatycznej; kontroluje siłę interakcji
6
7
8 def particle_definition(charge,radius): # definicja atomu złota
9     return sphere(pos=vector(0,0,0),color=color.yellow,charge=charge,
    radius=radius)
10
11 def simulation_run(Gold): # symulacja
12     while (len(Alphas)<500): # animacja do momentu wystrzelenia
    ustalonej liczby cząstek alfa
13         rate(1/dt) # szybkość animacji
14
15         r = random() # losowy moment wystrzelenia cząstki alfa
16         if (r < 0.1): # 10% szansa na strzał
17             x = -1.0
18             y = 2*random() - 1
19             z = 2*random() - 1
20             # Tworzenie cząsteczki alfa i dodawanie jej do listy.
21             Alphas.append(simple_sphere(pos=vector(x,y,z),velocity=
    vector(0.1,0,0),charge=2,mass=4,radius=0.01,color=color.red,
    make_trail=True))
22
23         for a in Alphas: # Iterowanie po cząstkach Alfa
24             a.force = k*a.charge*Gold.charge/mag(a.pos)**2 * hat(a.pos
    )
25             a.velocity += a.force/a.mass*dt
26             a.pos += a.velocity*dt
27             if (mag(a.pos) > sqrt(3)):
28                 a.velocity = vector(0,0,0)
29
30 def main(): # wywołanie gotowej symulacji
31     Gold = particle_definition(79,0.1)
32     simulation_run(Gold)
33
34 if __name__ == "__main__":
35     main()
```

4.1.2 Wynik symulacji

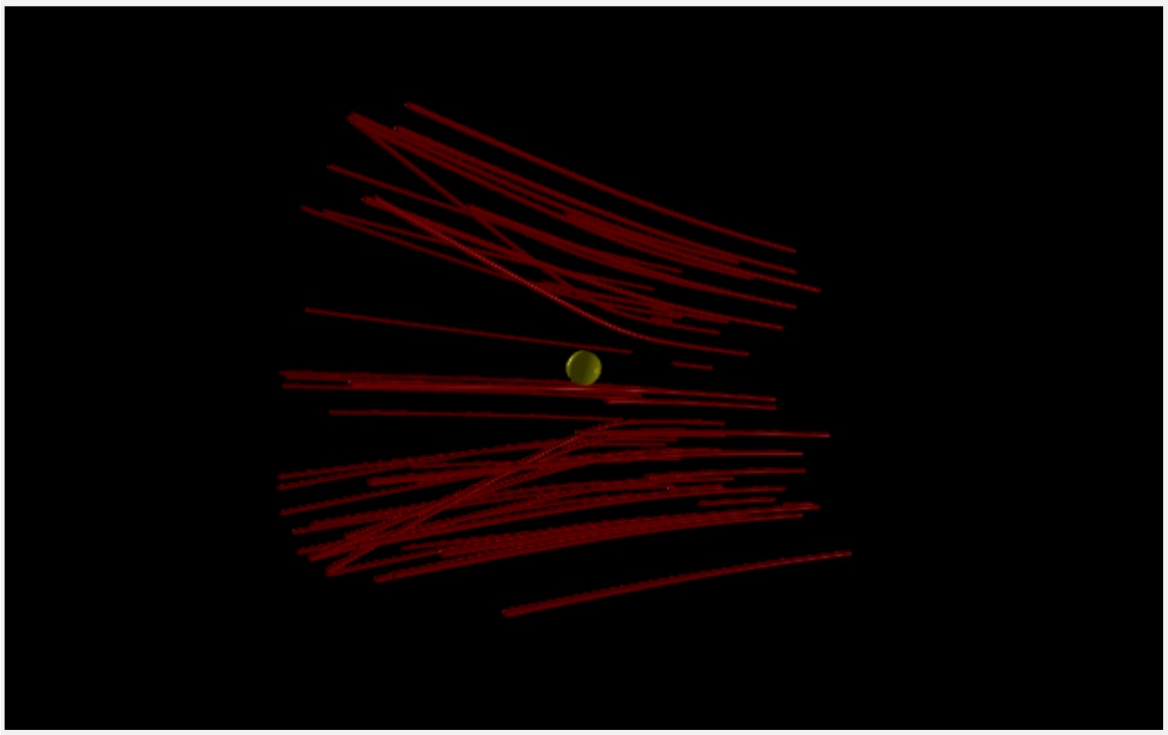
Aplikację uruchomić można z poziomu wiersza poleceń komendą `python3 rutherford-vpython.py`, po uprzednim wejściu do folderu, w którym się ona znajduje (stosując komendę: `cd ŚCIEŻKA DO PLIKU`, np.: `cd Desktop/rutherford--scattering`). Uruchomienie aplikacji, która nie została wcześniej skonteneryzowana, wymaga instalacji zdefiniowanych w kodzie bibliotek. Próba pominięcia tego kroku skutkować może błędem:

```
ModuleNotFoundError: No module named 'NAZWA BIBLIOTEKI'.
```

Niezbędne biblioteki można zainstalować przy pomocy komendy `pip3 install NAZWA BIBLIOTEKI`. Po zainstalowaniu wymaganej biblioteki (VPython) i uruchomieniu aplikacji, w oknie przeglądarki wyświetlona zostanie symulacja w trzech wymiarach:



Rysunek 4.1: Wynik symulacji napisanej w bibliotece VPython widoczny w oknie przeglądarki internetowej - rzut lewostronny.

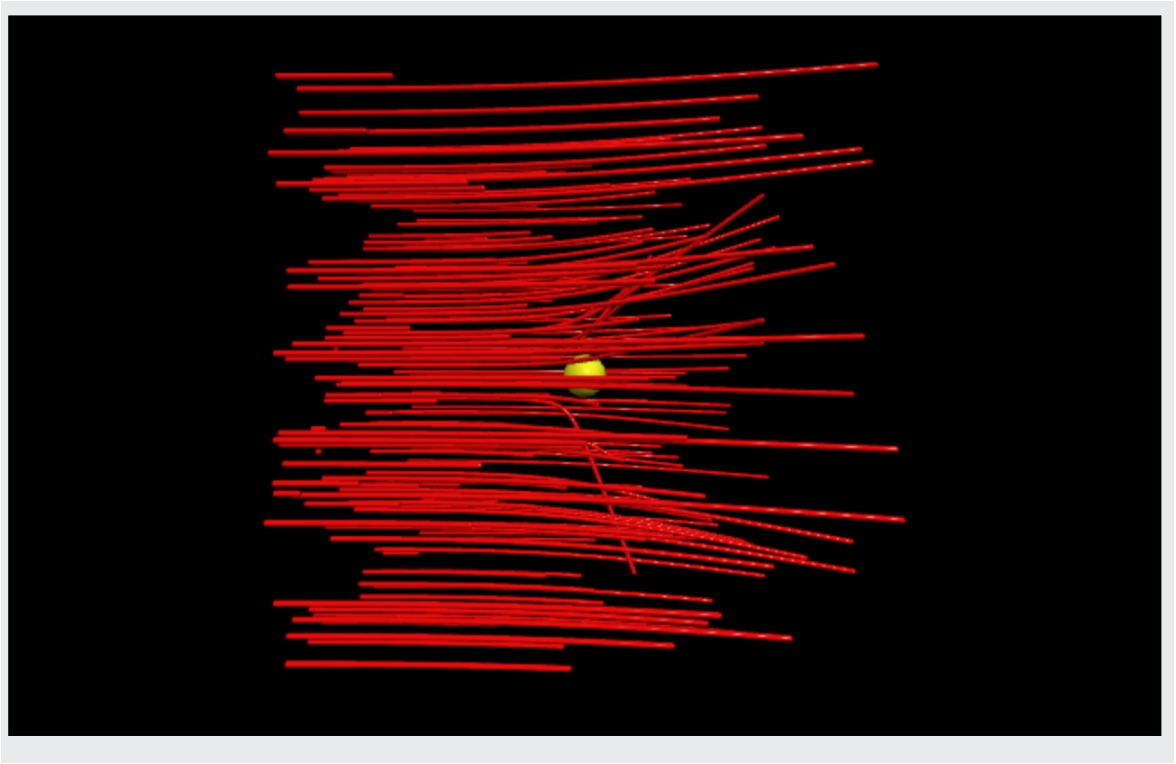


Rysunek 4.2: Wynik symulacji napisanej w bibliotece VPython widoczny w oknie przeglądarki internetowej - rzut prawostronny.

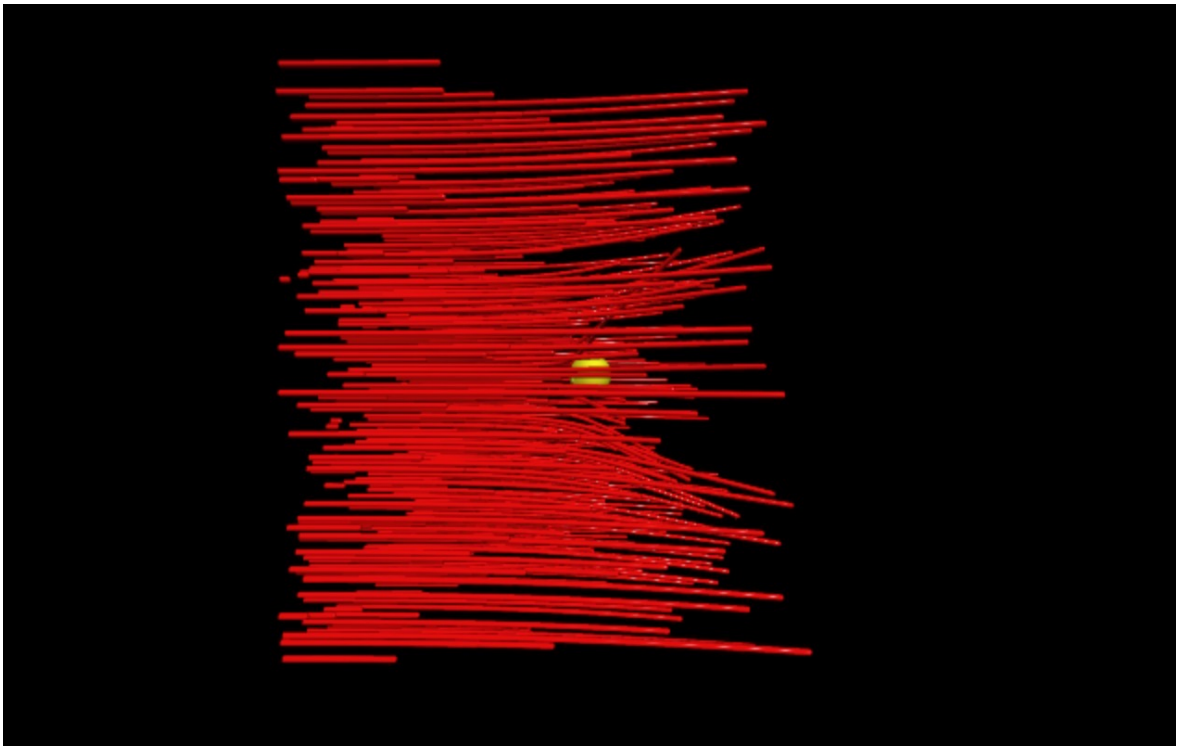
Jak widać na powyższych rysunkach, symulacją można dowolnie obracać przy pomocy kursora myszy. Po zamknięciu okna przeglądarki należy w wierszu poleceń zastosować kombinację klawiszy `ctrl+c` aby zakończyć działanie aplikacji.

4.1.2.1 Zależność trajektorii od kroku całkowania

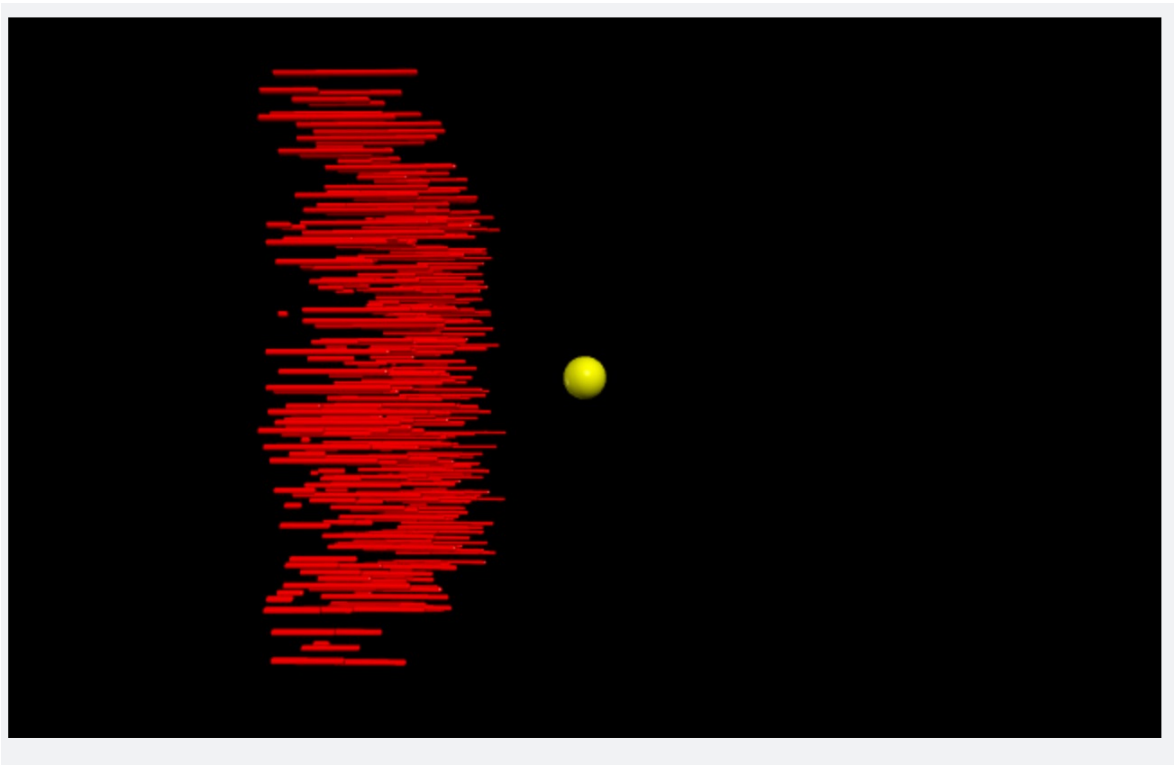
Krok całkowania `dt` zdefiniowany został w czwartej linijce kodu aplikacji. Określa on szybkość animacji, a dokładniej szybkość wystrzeliwania pożądanej liczby cząsteczek alfa w stronę atomu złota. Im mniejszy krok, tym więcej cząsteczek zostanie wystrzelonych na raz. Widać to na rysunkach 4.3 oraz 4.4. Przy pewnej dostatecznie małej wartości kroku (`dt = 0.001`), zbyt wiele cząsteczek zostaje wystrzelonych w jednym momencie, co powoduje znaczne spowolnienie działania aplikacji (jak zostało to przedstawione na rysunku 4.5). Wówczas cząsteczki nie mają szans na dotarcie w otoczenie atomu złota. W miarę zwiększania kroku całkowania, wystrzeliwanie kolejnych cząsteczek staje się coraz bardziej rozłożone w czasie, wobec czego symulacja działa efektywniej. Stopniowe zwiększanie kroku całkowania przedstawione zostało na rysunkach 4.6, 4.7 oraz 4.8. Przy zbyt dużym kroku całkowania cząsteczki są wystrzeliwane zbyt rzadko, wobec czego działanie aplikacji również jest znacznie spowolnione.



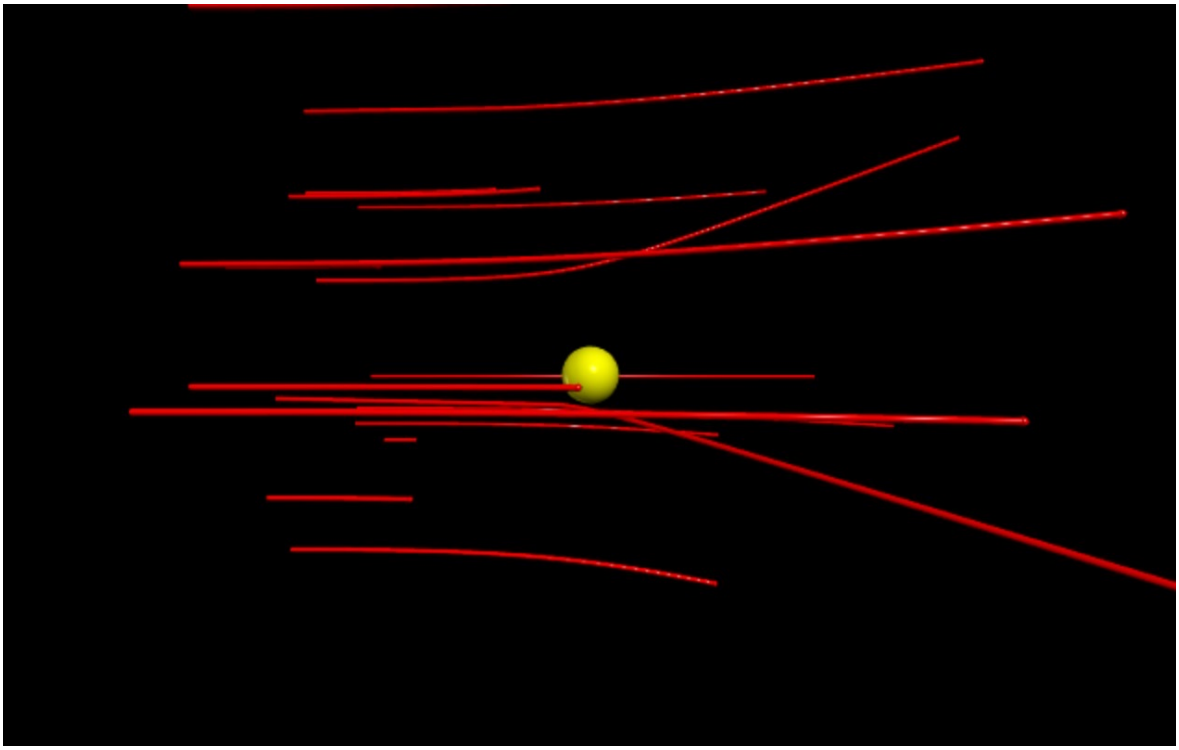
Rysunek 4.3: Wynik symulacji dla dt = 0.01.



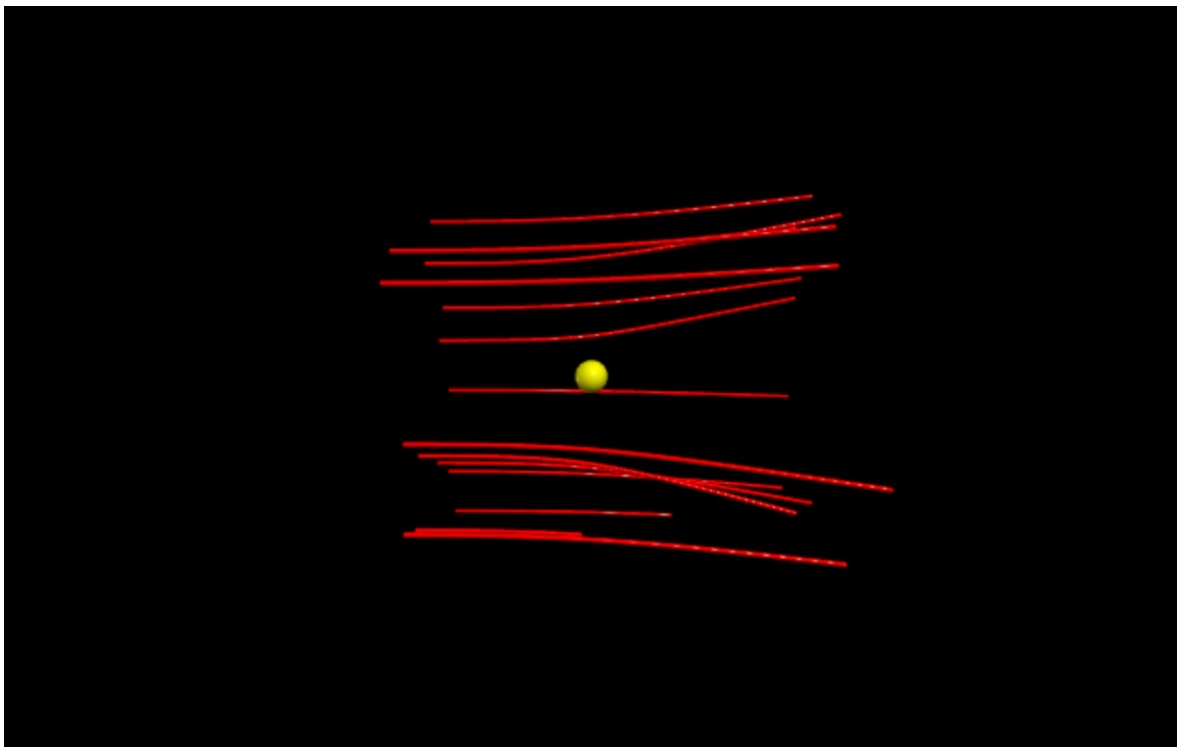
Rysunek 4.4: Wynik symulacji dla dt = 0.005.



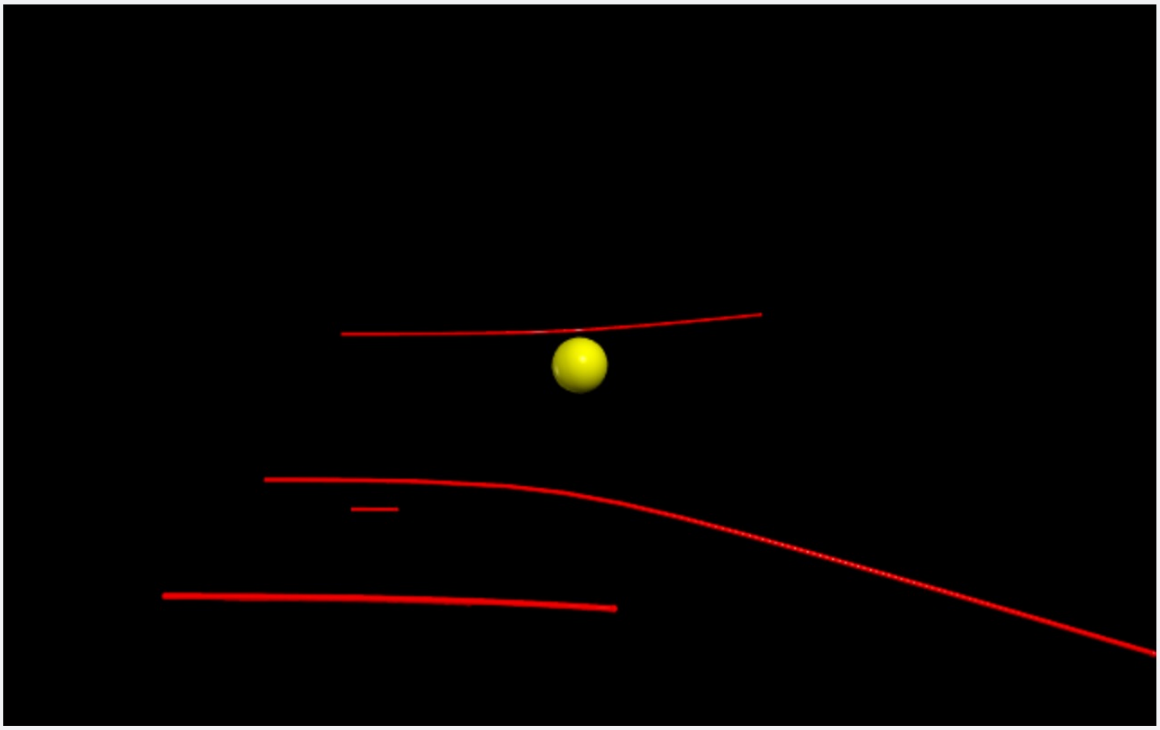
Rysunek 4.5: Wynik symulacji dla dt = 0.001.



Rysunek 4.6: Wynik symulacji dla dt = 0.25.



Rysunek 4.7: Wynik symulacji dla dt = 0.5.



Rysunek 4.8: Wynik symulacji dla dt = 1.

4.1.3 Przyczyny zaistniałych błędów

Biblioteka VPython udostępnia aplikację pod adresem `localhost 127.0.0.1` (używanym do lokalnej komunikacji, niezależnie od podłączenia do sieci). Dzięki temu symulacja wyświetlona zostaje w oknie przeglądarki internetowej. Jednak w przeciwieństwie do aplikacji napisanej w bibliotece Matplotlib (4.2.2), której udostępnienie w formie strony internetowej umożliwia Flask - aplikacja napisana w bibliotece VPython wyświetla się każdorazowo na innym porcie.

Port jest parametrem umożliwiającym połączenie. Służy on do *identyfikacji uruchamianych procesów na odległych sobie systemach* [22]. W przypadku biblioteki Flask, symulacja udostępiona zostaje każdorazowo na porcie `5000`. W przypadku biblioteki VPython port (a dokładniej gniazdo (ang. *socket*), które definiowane jest przez adres IP i port instancji końcowej) przydzielany jest losowo. Dzieje się tak z uwagi na fakt, że biblioteka VPython nie jest już oficjalnie wspierana w nowych wersjach języka Python, wobec czego funkcjonuje za pośrednictwem strony *Glowscript* [39].

Choć w niniejszej pracy lokalna instalacja biblioteki VPython (komendą `pip3 install vpython`) powiodła się i nie powstawała ona bezpośrednio na stronie Glowscript (jak zostało to ukazane w filmie, na którego podstawie powstała: [80]), jednak aplikacja podczas wyświetlania w przeglądarce łączyła się w tle ze stroną Glowscript (serwerem Glowscript) poprzez jej API. Problem ten ujawnił się podczas konteneryzacji.

Stworzenie kontenera wymaga napisania pliku konfiguracyjnego DockerFile, opisanego w rozdziale 2.3.1. Zawiera on między innymi informacje o porcie, na którym kontener ma zostać udostępiony. W przypadku aplikacji udostępnianej w przeglądarce za pośrednictwem biblioteki Flask, jest to port `5000`, analogicznie jak dla symulacji. Kod DockerFile zamieszczony został w rozdziale 5.1.1. W przypadku aplikacji napisanej w języku VPython określenie portu w pliku konfiguracyjnym kontenera nie jest możliwe z racji jego losowości. Wobec tego aplikacja ta nie mogła zostać wykorzystana na potrzeby niniejszej pracy, stąd decyzja o napisaniu drugiej aplikacji przy użyciu biblioteki Matplotlib - nieustannie wspieranej przez język Python oraz Flask, umożliwiający udostępnienie jej w przeglądarce z wykorzystaniem statycznie przydzielonego portu.

4.2 Aplikacja napisana w bibliotece Matplotlib

Aplikacja napisana w bibliotece Matplotlib powstała na podstawie kodu dostępnego w repozytorium GitHub pod adresem: [88]. Zmodyfikowany na potrzeby niniejszej pracy kod znajduje się na platformie GitHub [106] pod nazwą `rutherford--scattering.py`.

4.2.1 Omówienie aplikacji

Kod aplikacji powstał w głównej mierze w oparciu o bibliotekę Maplotlib, ale również NumPy oraz Flask. Szczególnie istotne są fragmenty kodu konieczne do prawidłowego uruchomienia symulacji w przeglądarce internetowej, dzięki zastosowaniu Flask. Są to:

- inicjacja Flask w kodzie: `app = Flask(__name__)`,

- dekorator Flask: `@app.route('/')`,
- funkcja zapisująca wynik symulacji w formacie `.png` w folderze *static*: `plt.savefig('./static/new_plot.png')`,
- funkcja generująca stronę w formacie `html`, na której wyświetlana jest zapisana grafika: `return render_template('index.html')`.

Foldery *static* oraz *templates* znajdują się w repozytorium kodu [106]. W folderze `static` zapisywany jest wynik symulacji w postaci graficznego pliku `.png` (został on przedstawiony w rozdziale 4.2.2 na rysunku 4.12). Plik ten zostaje nadpisany każdorazowo po uruchomieniu aplikacji. Wyświetlenie grafiki w przeglądarce internetowej możliwe jest dzięki plikowi `index.html`, znajdującemu się w folderze *templates*. Zawiera on podstawowy szablon strony interetowej. Jego kod wygląda następująco:

```
1 <!doctype html>
2 <html>
3     <body>
4 
5         <h1>Rutherford Scattering</h1>
6 
7 
8         <img src="static/new_plot.png" width="1280" height="720">
9 
10    </body>
11 </html>
```

Wyświetla on tytuł strony (*Rutherford Scattering*) w linijce piątej, w ósmej natomiast importuje zapisaną grafikę symulacji z folderu *static* oraz wyświetla ją w określonej skali.

Kod symulacji przedstawiony został poniżej. Jego ogólna zasada działania jest zbliżona do zasady działania symulacji napisanej w bibliotece VPython. Po zaimportowaniu niezbędnych bibliotek (w linijkach 1-10), następuje określenie stylu wyświetlania symulacji (`plt.style.use('seaborn-whitegrid ')`) oraz tego, co ma ona ukazywać (`fig , xy plt.subplots ()`). Następnie (w linijkach 17-27) zdefiniowane zostają podstawowe dane, wykorzystywane w późniejszych fragmentach kodu. Są one szczegółowo zakomentowane wewnątrz niego. Funkcja `getYY` reprezentuje wzór $\frac{F}{m}$, gdzie F - siła Coulomba. Po zastosowaniu w linijce 32 dekoratora Flask, następuje definicja głównej funkcji aplikacji (`def simulation_run ():`), w której wykonywane są niezbędne obliczenia.

Na początku, w linijce 34 utworzona zostaje pusta lista `params []`), do której dodane zostaną elementy z pętli `for`. Pętle te (`for i in range()`) określają zakres, w jakim pojawiać będą się wystrzelone w stronę atomu złota cząsteczki. W linijce 45 (`for y0 in params:`) rozpoczyna się obliczanie przebiegu trajektorii tychże cząsteczek. `x, y` to położenia, natomiast `Vx, Vy` to prędkości. Do ich obliczenia wykorzystany został tzw. *Algorytm Skokowy* (ang. *Leapfrog integration*) [104]. Służy on do całkowania równań w postaci $\dot{v} = F(x)$, $\dot{x} \equiv v$. Metoda ta opiera się na aktualizowaniu położenia $x(t)$ oraz prędkości $v(t) \equiv \dot{x}(t)$ w naprzemiennych krokach czasowych, dzięki czemu

„przeskakują one nad sobą" – stąd nazwa algorytmu. Jest on metodą drugiego rzędu, w przeciwieństwie do metody Eulera, wykorzystanej w poprzedniej aplikacji. Dla małych kroków czasowych h, ruch może byc przybliżany ruchem jednostajnie przyspieszonym $x = x_0 + vh + \frac{1}{2}F/mh^2$, gdzie F/m to przyspieszenie. Stąd w symulacji wykorzystano algorytm skokowy, w którym:

$$a_i = F(x_i), \tag{4.1}$$

$$x_{i+1} = x_i + v_i h + \frac{1}{2} a_i h^2, \tag{4.2}$$

$$v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1})h. \tag{4.3}$$

Po obliczeniu położeń i prędkości w linijkach 45-58, następuje pętla `while`, trwająca dopóki wartości x i y znajdują się w określonym przedziale. `constAx` oraz `constAy` równe są sile Coulomba, pomnożonej przez składowe osi x i y. Zostają one wykorzystane w linijkach 68-83, w których następuje zaktualizowanie położeń i prędkości cząsteczek o kolejne kroki h, aż do momentu zakończenia działania pętli. Wówczas zostają one wyświetlone w formie trajektorii (`xy.plot(X, Y)`), wraz z pozostałymi elementami symulacji (w linijkach 85-95). Całość utworzonej symulacji zostaje zapisana w formie grafiki `new_plot.png` w folderze *static* oraz dodana do szablonu `index.html`, wyświetlanego w przeglądarce internetowej (`return render_template('index.html')`).

Szczegółowy opis poszczególnych linijek kodu został w nim zakomentowany, jak poniżej:

```
1 import math
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import matplotlib.ticker as ticker
6 import io
7 from flask import Response
8 from matplotlib.backends.backend_agg import FigureCanvasAgg as
     FigureCanvas
9 from matplotlib.figure import Figure
10 from flask import Flask, render_template
11 app = Flask(__name__)
12
13 plt.style.use('seaborn-whitegrid')
14 fig, xy = plt.subplots()
15
16 #Dane wejściowe
17 const = 1e-12
18 h = 1.0e-24  # krok
19 Vx0 = 1.0e7  # prędkość cząstki uciekającej z punktu 0 wzdłuż osi OX
20 Vy0 = 0  # prędkość cząstki uciekającej z punktu 0 wzdłuż osi OY
21 q = 1.6e-19  # ładunek izotopowy
22 ma = 1.661e-27  # masa atomowa (1 unit)
23 x0 = -const  # początek współrzędnej X
24 k = 1 / (4 * math.pi * 8.85e-12) # stała elektrostatyczna
25 m = 4 * ma # masa cząsteczkowa helu
26 q1 = 79 * q # ładunek jądra złota
```

```
27 q2 = 2 * q # ładunek jądra helu
28
29 def getYY(qa, qb, r1, r2, mass): # obliczanie siły Coulomba
30     return k * qa * qb / (math.sqrt(r1 ** 2 + r2 ** 2)) ** 2 / mass
31
32 @app.route('/') # dekorator Flask
33 def simulation_run():
34     params = []
35     for i in range(-5, 6, 1): #zakres pojawiania się cząsteczek
36         if i != 0:
37             params.append(i*1e-14)
38     for i in range(-50, 51, 10):
39         if i != 0:
40             params.append(i*1e-14)
41     for i in range(-500, 501, 100):
42         if i != 0:
43             params.append(i*1e-14)
44
45     for y0 in params: # obliczanie przebiegu trajektorii cząsteczek
46         X = [x0]
47         Y = [y0]
48         x = x0 + h * Vx0 + h ** 2 / 2 * getYY(q1, q2, x0, y0, m) * x0
    / math.sqrt(x0 ** 2 + y0 ** 2)
49         y = y0 + h * Vy0 + h ** 2 / 2 * getYY(q1, q2, y0, y0, m) * y0
    / math.sqrt(x0 ** 2 + y0 ** 2)
50
51         Vx = Vx0 + getYY(q1, q2, x, y, m) * x / math.sqrt(x ** 2 + y
    ** 2) * h
52         Vy = Vy0 + getYY(q1, q2, x, y, m) * y / math.sqrt(x ** 2 + y
    ** 2) * h
53
54         prev_Vx = Vx0
55         prev_Vy = Vy0
56
57         prev_x = x0
58         prev_y = y0
59
60         while np.isfinite(y) and -const < x < const and -5*const < y <
     5*const:
61             X.append(x)
62             Y.append(y)
63
64             constAx = getYY(q1, q2, x, y, m) * x / math.sqrt(x ** 2 +
    y ** 2)
65             constAy = getYY(q1, q2, x, y, m) * y / math.sqrt(x ** 2 +
    y ** 2)
66
67             # aktualizacja położeń na osiach
68             val1 = x
69             x = h ** 2 * constAx + 2 * x - prev_x
70             prev_x = val1
71
72             val2 = y
73             y = h ** 2 * constAy + 2 * y - prev_y
74             prev_y = val2
75
76             # aktualizacja prędkości cząstek poruszających wzdłuż osi
```

```
77              val3 = Vx
78              Vx = prev_Vx + constAx * h
79              prev_Vx = val3
80
81              val4 = Vy
82              Vy = prev_Vy + constAy * h
83              prev_Vy = val4
84
85          plt.title("Symulacja cząstek") # tytuł symulacji, wyświetlany
        na górze wykresu
86          plt.xlabel("X") # nazwa osi x
87          plt.ylabel("Y") # nazwa osi y
88          xy.xaxis.set_major_locator(ticker.MultipleLocator(const/10)) #
         generowanie punktów na osi x co 0.1
89          xy.yaxis.set_major_locator(ticker.MultipleLocator(const/10)) #
         generowanie punktów na osi y co 0.1
90          plt.grid(True) # wyświetlanie siatki
91          xy.plot(X, Y) # wyświetlanie trajektorii cząsteczek helu
92
93      xy.scatter(0, 0) # wyświetlanie atomu złota
94      plt.xlim(-const, const) # skala osi x
95      plt.ylim(-const, const) # skala osi y
96      plt.savefig('./static/new_plot.png') # zapisywanie grafiki
        symulacji do pliku .png
97
98      return render_template('index.html') # plik html wyświetlany wraz
        z symulacją dzięki Flask
99
100 if __name__ == "__main__":
101     app.run(debug = True)
```

4.2.2 Wynik symulacji

Aplikację uruchomić można z poziomu wiersza poleceń komendą `python3 rutherford-scattering.py`, po uprzednim wejściu do folderu, w którym się ona znajduje (stosując komendę: `cd ŚCIEŻKA DO PLIKU`, np.: `cd Desktop/rutherford--scattering`). Uruchomienie aplikacji, która nie została wcześniej skonteneryzowana, wymaga instalacji zdefiniowanych w kodzie bibliotek. Próba pominięcia tego kroku skutkować może błędem:

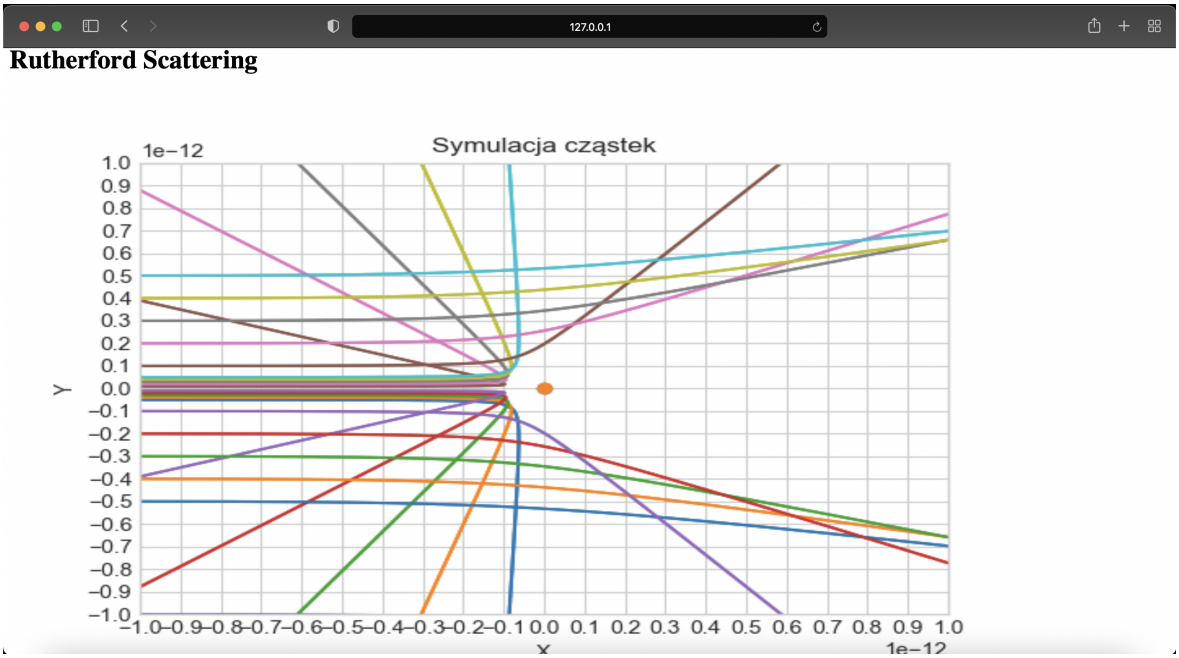
```
ModuleNotFoundError: No module named 'NAZWA BIBLIOTEKI'.
```

Niezbędne biblioteki można zainstalować przy pomocy komendy `pip3 install NAZWA BIBLIOTEKI`. Po zainstalowaniu wymaganych bibliotek i uruchomieniu aplikacji widoczne stają się poniższe informacje:

```
[gabrielabialoskorska@MacBook-Air Rutherford-Scattering % python3 rutherford-scattering.py
 * Serving Flask app 'rutherford-scattering' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 203-491-217
```

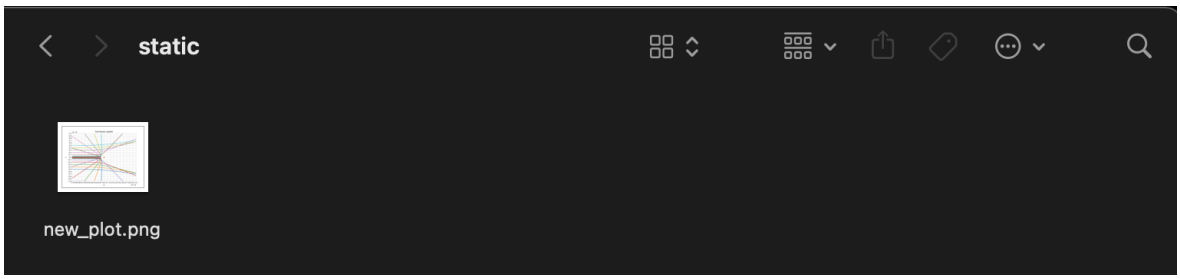
Rysunek 4.9: Uruchomienie aplikacji napisanej w bibliotece Matplotlib.

Jak można zauważyć, biblioteka Flask udostępnia aplikację pod adresem localhost `127.0.0.1` (używanym do lokalnej komunikacji, niezależnie od podłączenia do sieci) na porcie `5000`. Aby wyświetlić wynik symulacji, należy skopiować pełen jej adres, widoczny powyżej w linijce *Running on* (`http://127.0.0.1:5000/`) i wkleić go w oknie przeglądarki internetowej. Po chwili widoczna staje się grafika:



Rysunek 4.10: Wynik symulacji napisanej w bibliotece Matplotlib widoczny w oknie przeglądarki internetowej.

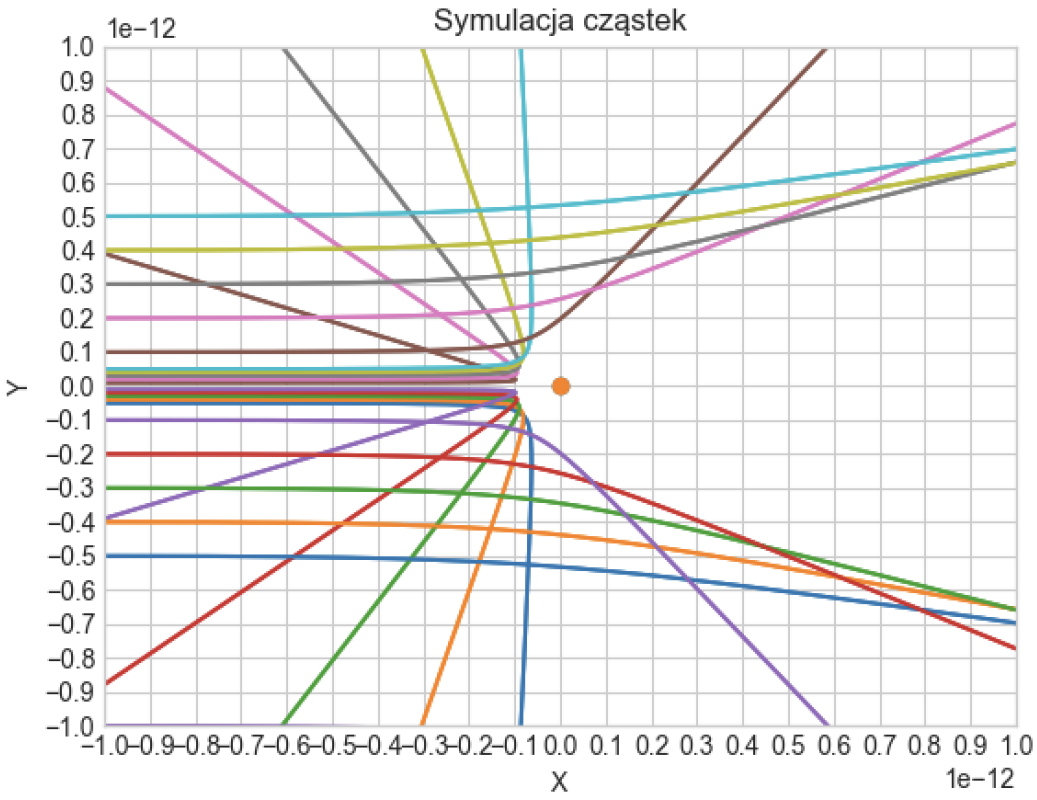
Po zamknięciu okna przeglądarki należy w wierszu poleceń zastosować kombinację klawiszy `ctrl+c` aby zakończyć działanie aplikacji. Wynik symulacji w postaci graficznego pliku `.png` zostaje zapisany w folderze *static*, jak zostało to przedstawione poniżej.



Rysunek 4.11: Folder, w którym zapisany zostaje wynik symulacji (new_plot.png).

Grafika ta nadpisuje się każdorazowo po uruchomieniu aplikacji.

Wygląda ona następująco:



Rysunek 4.12: Graficzny plik z wynikiem symulacji (new_plot.png).

Rozdział 5

Tworzenie infrastruktury

5.1 Konteneryzacja aplikacji

Kontener uruchamiany jest na podstawie istniejącego obrazu, dzięki czemu może on być przenoszony pomiędzy środowiskami. Sposób zachowania kontenera po uruchomieniu oraz pliki znajdujące się wewnątrz obrazu określa DockerFile. Jest to zbiór instrukcji tworzących kolejne warstwy obrazu kontenera [72].

Konteneryzacja aplikacji wymaga wykonania poniżej opisanych kroków, zgodnie z rysunkiem 2.10.

- Stworzenie pliku Dockerfile, określającego sposób uruchamiania aplikacji, port na którym zostanie ona udostępniona oraz jej *requirements* - wymagane przez aplikację biblioteki. Wykorzystany w niniejszej pracy Dockerfile opisany został w rozdziale 5.1.1.
- Stworzenie obrazu aplikacji (Docker Image) w oparciu o DockerFile, przy pomocy wiersza poleceń (5.1.2) lub w sposób zautomatyzowany (5.1.3).
- Uruchomienie kontenera w celu sprawdzenia, czy zadział on w sposób prawidłowy.

Poniższe rozdziały szczegółowo opisują sposób ich wykonania.

5.1.1 Omówienie pliku Dockerfile

Napisany na potrzeby niniejszej pracy plik Dockerfile znajduje się w repozytorium [106] pod nazwą `Dockerfile`. Jego kod wygląda w następujący sposób:

```
1 # syntax=docker/dockerfile:1
2
3 FROM docker.io/python:3.9
4
5 ADD . /app
6 WORKDIR /app
7
8 RUN pip3 install --no-cache-dir -r requirements.txt
9
10 EXPOSE 5000
```

```
11
12 CMD gunicorn --threads=2 --bind 0.0.0.0:5000 rutherford-scattering:app
13 # CMD python3 rutherford-scattering.py
```

- `# syntax=docker/dockerfile:1` - dyrektywa *syntax* definiuje wersję języka (składni), w jakiej Dockerfile został napisany (lokalizację obrazu tejże wersji). W tym przypadku wykorzystano oficjalną wersję obrazu dystrybuowanego przez Docker, dostępną w repozytorium `docker/dockerfile` na platformie *Docker Hub*. Jest to usługa przypominająca GitHub, która umożliwia pobieranie obrazów Dockera z publicznych repozytoriów użytkowników Docker Hub [73]. `docker/dockerfile:1` oznacza najnowszą stabilną wersję obrazu składni [20].

- `FROM docker.io/python:3.9` - instrukcja *FROM* definiuje podstawowy obraz dla późniejszych instrukcji [20]. W tym przypadku jest to oficjalny obraz języka Python, ponieważ konteneryzowana aplikacja napisana została z jego wykorzystaniem.

- `ADD . /app` - polecenie *ADD* umożliwia kopiowanie plików lub katalogów do określonej lokalizacji (ścieżki) do obrazu Docker Image, jaki ma zostać stworzony w oparciu o DockerFile [71]. W tym przypadku zdefiniowane zostało, że pobrany przez dowolnego użytkownika obraz aplikacji (powstały na podstawie niniejszego Dockerfile) oraz wszystkie pliki jakie stworzy uruchomiony na jego podstawie kontener, znalazły się w katalogu `. /app` na maszynie tego użytkownika.

- `WORKDIR /app` - definiuje katalog roboczy, w jakim funkcjonować będzie kontener. W tym przypadku ponownie jest to katalog `/app`, ponieważ to w nim znajdują się wszystkie niezbędne do funkcjonowania aplikacji pliki. Jest to analogiczne do uruchamiania aplikacji bez wykorzystania konteneryzacji (4.2.2) - najpierw należy podać ścieżkę do pliku, następnie uruchomić symulację.

- `RUN pip3 install -no-cache-dir -r requirements.txt` - instrukcja *RUN* służy do wykonywania zadanych poleceń [20]. W tym przypadku wykonuje komendę `pip3 install requirements.txt` z flagą `-no-cache-dir`. Komenda ta instaluje biblioteki niezbędne do działania aplikacji, zawarte w pliku `requirements.txt` w repozytorium pracy pod adresem: [106]. Plik ten wygląda następująco:

  ```
  1    Flask
  2    matplotlib
  3    numpy
  4    gunicorn
  ```

  Flaga `-no-cache-dir` umożliwia utrzymanie obrazu tworzonego na podstawie niniejszego DockerFile w najmniejszej możliwej formie. Zapobiega ona przechowywaniu plików związanych z instalacją bibliotek w pamięci podręcznej maszyny użytkownika.

- `EXPOSE 5000` - instrukcja *EXPOSE* określa port, którego kontener używać będzie po uruchomieniu. W tym przypadku jest to port `5000`, ponieważ to na nim Flask domyślnie udostępnia aplikację.

- `CMD gunicorn -threads=2 -bind 0.0.0.0:5000 rutherford-scattering:app` - instrukcja *CMD pozwala na określenie domyślnego polecenia, które zostanie wykonane tylko podczas uruchamiania kontenera bez podawania dodatkowych argumentów* [72]. W tym przypadku komenda ta ma za zadanie uruchomić *Gunicorn* [41]. Jest to serwer HTTP WSGI (ang. *Python Web Server Gateway Interface*), który umożliwia uruchomienie aplikacji napisanej z wykorzystaniem biblioteki Flask. Aplikacja ta ma gotowy interfejs do wyświetlenia w przeglądarce, jednak wymaga serwera, na którym będzie funkcjonować - ponieważ nie jest to już maszyna lokalna, na której powstała aplikacja, wraz z jej adresem localhost. Flaga `-threads` definiuje ilość corów serwera, jaka ma zostać wykorzystana (w tym przypadku 2), `-bind` nasłuchuje na wszystkich interfejsach serwera na porcie 5000, `rutherford-scattering:app` jest nazwą aplikacji bez rozszerzenia `.py` - zamiast tego stosowane jest `:app`.

- `# CMD python3 rutherford-scattering.py` - w tym przypadku instrukcja CMD wykonuje komendę `python3`, która uruchamia aplikację `rutherford--scattering.py`. Polecenie to jest uruchamiane po starcie kontenera.

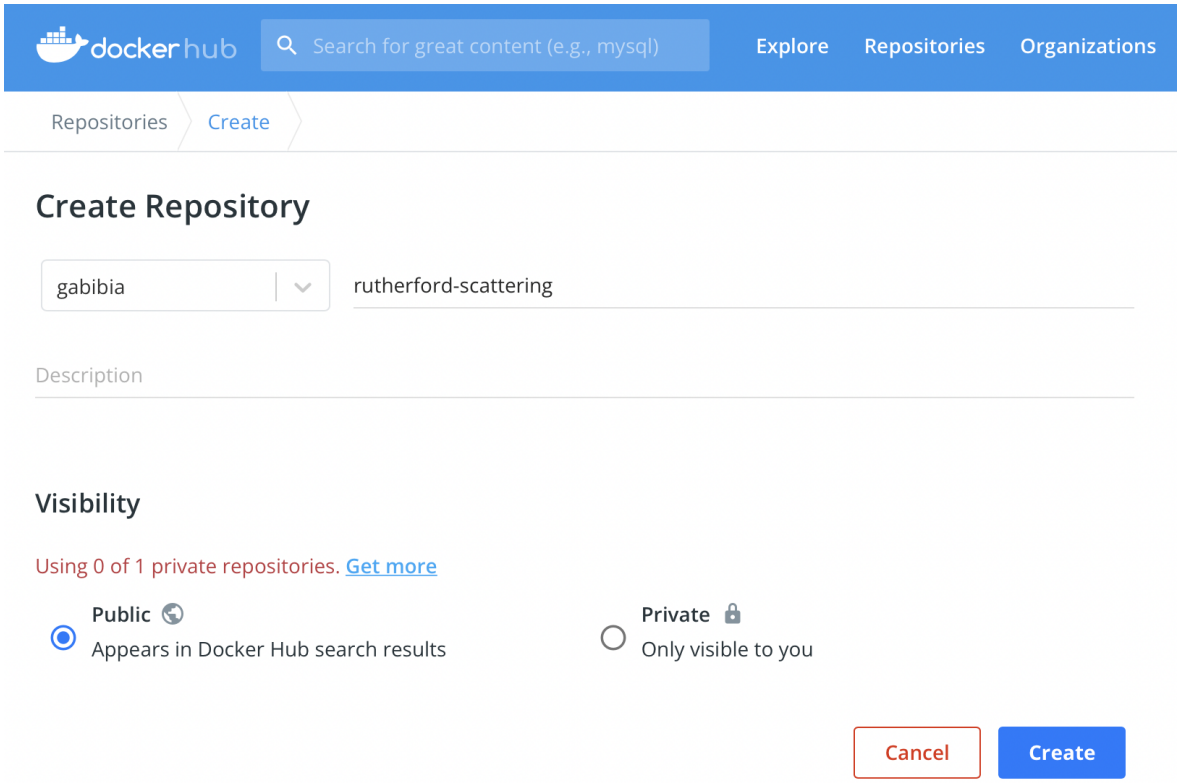
Na podstawie powyżej opisanego DockerFile powstaje obraz kontenera. Sposoby jego tworzenia opisane zostały w poniższych podrozdziałach. Na potrzeby późniejszego wdrożenia infrastruktury konieczny jest jedynie gotowy obraz kontenera, nie sam kontener, jednak on również zostanie stworzony jako test poprawnego działania DockerFile.

5.1.2 Tworzenie obrazu w sposób manualny

Tworzenie obrazu w sposób manualny oznacza uruchomienie w wierszu poleceń określonego zestawu komend, bazujących na DockerFile. Niezbędna ku temu jest instalacja oprogramowania Docker ze strony [34]. Gotowy obraz umieszcza się następnie na platformie Docker Hub [19], wobec czego konieczne jest stworzenie na niej konta. Kolejne kroki opisane zostały poniżej.

- **Logowanie do Docker Hub z poziomu wiersza poleceń** - mając utworzone konto na platformie Docker Hub, należy otworzyć wiersz poleceń i zalogować się do niego komendą `docker login -u NAZWA KONTA`. `NAZWA KONTA` odnosi się do nazwy konta na Docker Hub. Następnie wiersz poleceń będzie wymagać podania hasła do konta. Chcąc uniknąć bezpośredniego wpisywania hasła, dla bezpieczeństwa można posłużyć się tokenem. Token na platformie Docker Hub tworzy się wchodząc w profil użytkownika (klikając na zdjęcie profilowe w prawym górnym rogu), następnie w zakładkę *Account Settings* i wybierając *Security* z panelu po lewej stronie. Wówczas wybiera się opcję *„New Access Token”* i przypisuje mu uprawnienia *„Read, Write, Delete”*. Nazwa tokena jest dowolna.

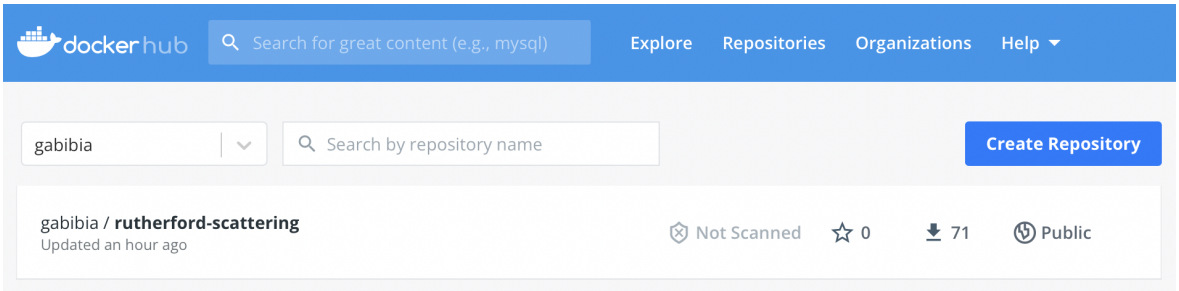
- **Tworzenie repozytorium Docker Hub** - analogicznie jak w przypadku platformy GitHub, należy stworzyć zdalne, publiczne repozytorium, do którego przesłany zostanie gotowy obraz. Aby tego dokonać, należy w górnym panelu wybrać zakładkę *Repositories*, a następnie *Create Repository*. W niniejszej pracy nazywa się ono `rutherford-scattering`.



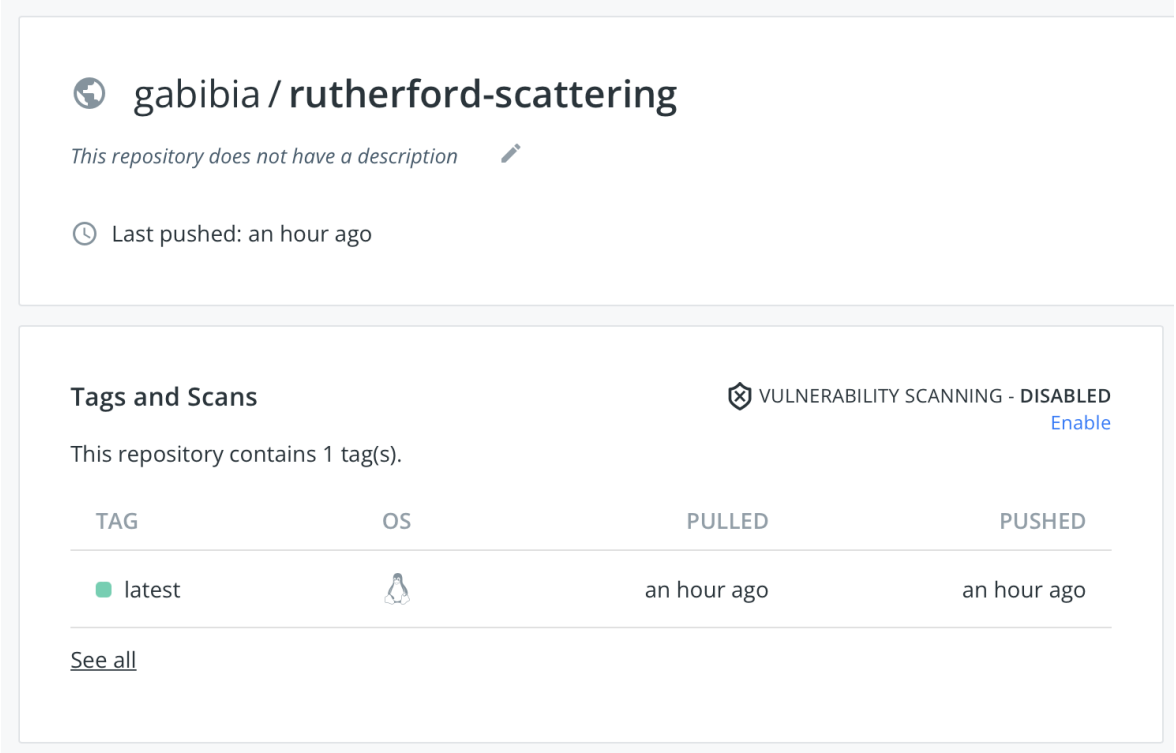
Rysunek 5.1: Tworzenie nowego repozytorium na platformie Docker Hub.

- **Tworzenie obrazu kontenera** - analogicznie jak dla Python, będąc w folderze w którym znajduje się plik DockerFile (`cd ŚCIEŻKA DO PLIKU`) należy uruchomić w wierszu poleceń komendę `docker build -t NAZWA OBRAZU .` . W przypadku niniejszej pracy nazwa obrazu to `rutherford-scattering`, należy jednak podać również nazwę użytkownika Docker Hub tworzącego niniejszy obraz (aby uniknąć sytuacji, w której wiele osób w ten sam sposób nazywa swoje obrazy). Wobec tego pełna komenda w przypadku tworzonego w pracy ubrazu wygląda następująco: `docker build -t gabibia/rutherford-scattering`. Jednocześnie ścieżka `gabibia/rutherford-scattering` stanowi odniesienie do repozytorium obrazu w Docker Hub. Flaga `-t` dodaje tag do budowanego obrazu, oznaczający jego wersję. Nadawać tagi obrazom można również po ich zbudowaniu, komendą `docker image tag NAZWA OBRAZU UŻYTKOWNIK/NAZWA OBRAZU: :latest`, gdzie `:latest` oznacza najnowszą, najbardziej aktualną jego wersję.
- **Publikowanie obrazu na platformie Docker Hub** - gdy obraz zostanie sukcesywnie zbudowany, może on zostać opublikowany na platformie Docker Hub, tak aby można go było w przyszłości wykorzystywać, podając jego adres (analogicznie jak w przypadku obrazów wykorzystanych w DockerFile, np. składni syntax). Będąc zalogowanym do Docker Hub z poziomu wiersza poleceń, należy wykonać komendę `docker image push UŻYTKOWNIK/NAZWA OBRAZU`. W niniejszej pracy wygląda ona następująco: `docker push gabibia/rutherford-scattering`. Aby sprawdzić, czy obraz został prawidłowo opublikowany, należy zalogować się na swoje konto Docker Hub z poziomu przeglądarki (na stronie [19]) i wejść do repozytorium, w którym powinien znajdować się obraz, jak zostało to przedstawione

na poniższych rysunkach.



Rysunek 5.2: Odnalezienie stworzonego repozytorium na stronie głównej konta Docker Hub.



Rysunek 5.3: Sprawdzenie, czy w repozytorium znajduje się obraz z tagiem `:latest`.

- **Sprawdzanie prawidłowego działania obrazu** - aby sprawdzić, czy obraz działa w sposób prawidłowy, należy na jego podstawie uruchomić kontener. W tym celu wykorzystuje się komendę `docker run -d -p PORT HOSTA:PORT KONTENERA UŻYTKOWNIK/NAZWA OBRAZU`. Flaga `-d` oznacza tryb *detached* (odłączony) - kontener uruchomiony w tym trybie zostanie automatycznie zatrzymany po zakończeniu procesu głównego, uruchamiającego kontener. Nie będzie on działał „w tle” po wykonaniu swojego zadania. Flaga `-p` oznacza *publish* - publikuje ona port kontenera na hoście, umożliwia udostępnienie go poza platformą Docker. W przypadku stworzonego na potrzeby pracy obrazu, pełna komenda wygląda następująco: `docker run -d -p 5000:5000 gabibia/rutherford-scattering`.

Ponieważ kontener działa na tym samym porcie, który wykorzystuje również na hoście, stąd fragment `5000:5000` - nie następuje w nim żadne przekierowanie portów. Jeśli obraz jest prawidłowy, zostanie uruchomiony kontener, a tym samym skonteneryzowana aplikacja.

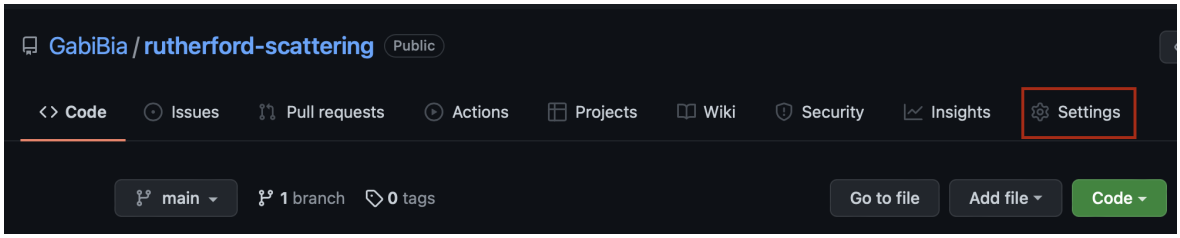
5.1.3 Tworzenie obrazu w sposób zautomatyzowany

Istnieje możliwość automatyzacji tworzenia nowego obrazu, aby uniknąć ręcznego wpisywania szeregu komend każdorazowo po wprowadzeniu zmian w kodzie konteneryzowanej aplikacji. W tym celu wykorzystuje się opisane w rozdziale 3.2.4 GitHub Actions.

W niniejszej pracy, każdorazowo po wprowadzeniu jakichkolwiek zmian w repozytorium (po wykonaniu komendy `git push`), następuje proces budowania nowego obrazu (Docker Image) aplikacji. Jest on zatem aktywowany poprzez wykonaną komendę `git push`. Dzięki temu wprowadzenie zmian np. w kodzie aplikacji automatycznie uruchamia proces tworzenia aktualnego dla niej obrazu kontenera, wobec czego aplikacja w repozytorium jest zawsze zgodna z aplikacją konteneryzowaną.

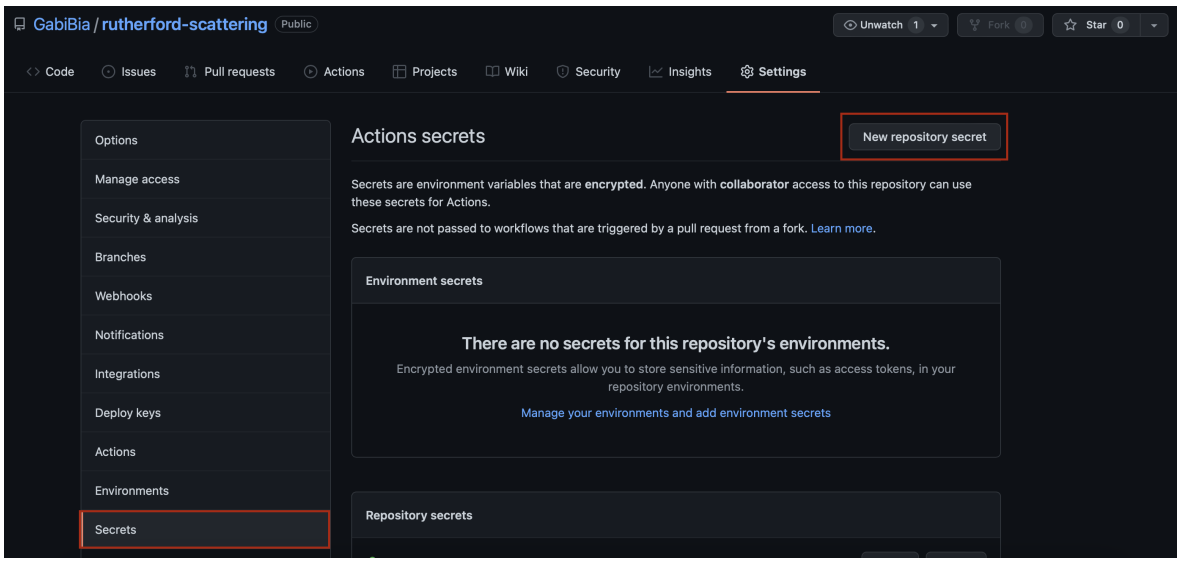
W celu stworzenia nowej akcji, należy wykonać poniżej opisane kroki.

- **Tworzenie sekretów w repozytorium GitHub** - stworzone sekrety umożliwią wykonanie jednego z kroków (logowanie do GitHub), zawartego w skrypcie automatyzującym budowanie obrazu. Aby tego dokonać, należy wejść w repozytorium na platformie GitHub, w którym chcemy zawrzeć automatyzację, i wybrać zakładkę *Settings*, jak zostało to przedstawione na rysunku poniżej:



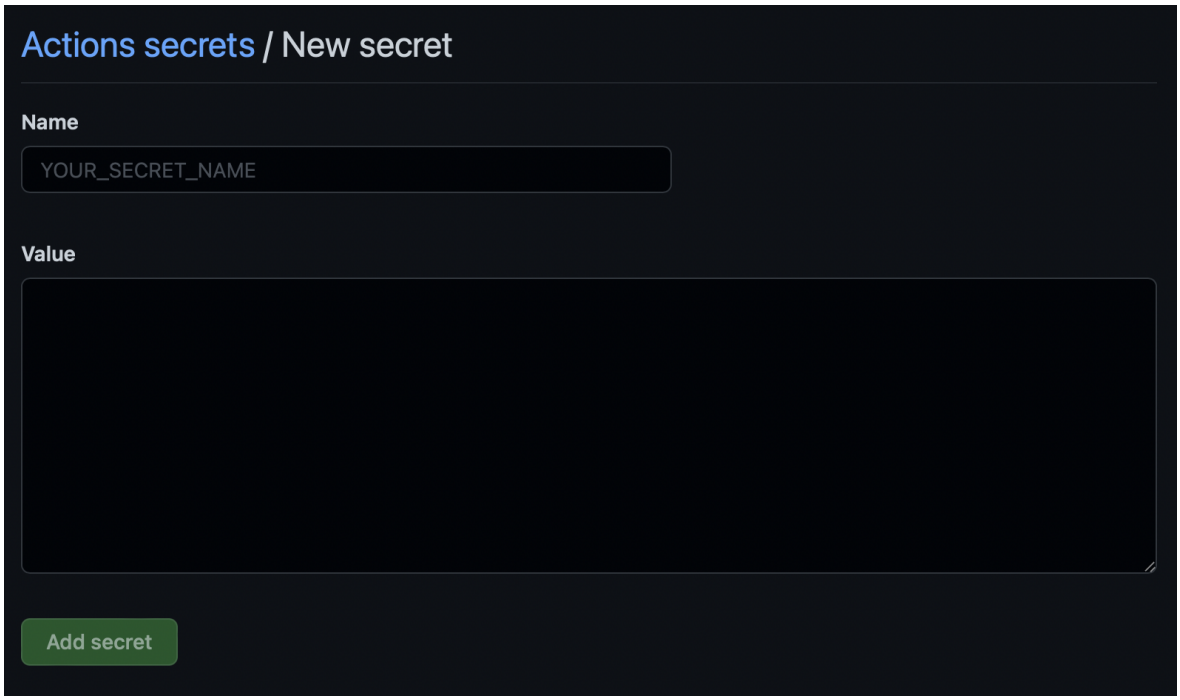
Rysunek 5.4: Tworzenie sekretów w repozytorium - ustawienia.

W panelu po lewej stronie otwartego okna należy wybrać zakładkę *Secrets*, a następnie *„New repository secret”*:



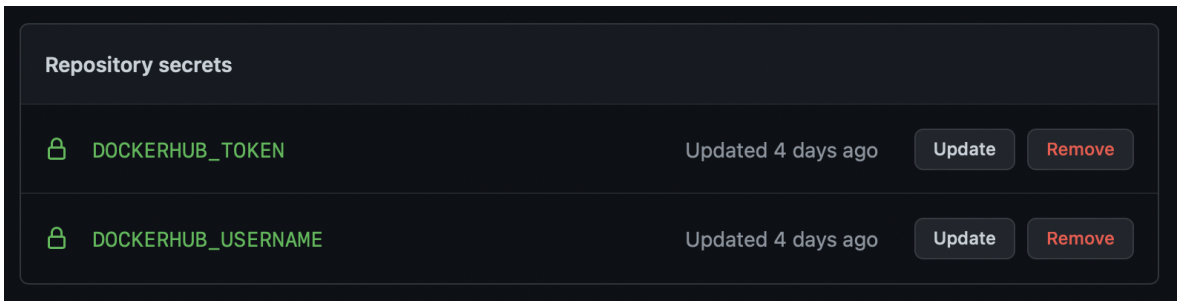
Rysunek 5.5: Tworzenie sekretów w repozytorium - nowy sekret.

W tym miejscu należy stworzyć dwa sekrety: jeden o nazwie `DOCKERHUB_TOKEN` z wartością (*value*) równą hasłu do konta GitHub, na którym znajduje się repozytorium oraz `DOCKERHUB_USERNAME` z wartością nazwy tego konta.



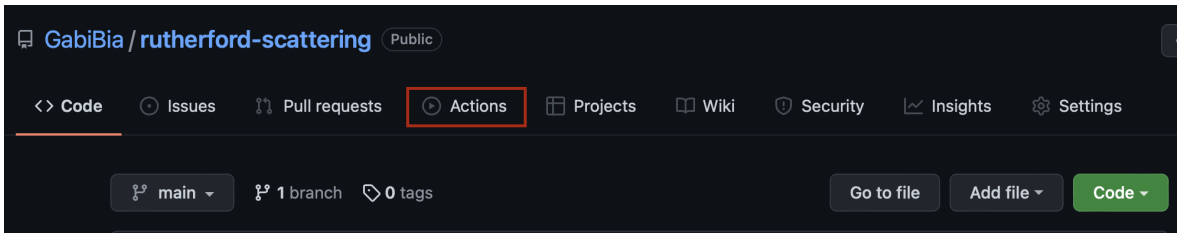
Rysunek 5.6: Tworzenie sekretów w repozytorium - wartość sekretu.

Stworzone sekrety będą od tej pory widoczne w zakładce *Settings* dla konkretnego repozytorium (w tym przypadku `rutherford-scattering`).



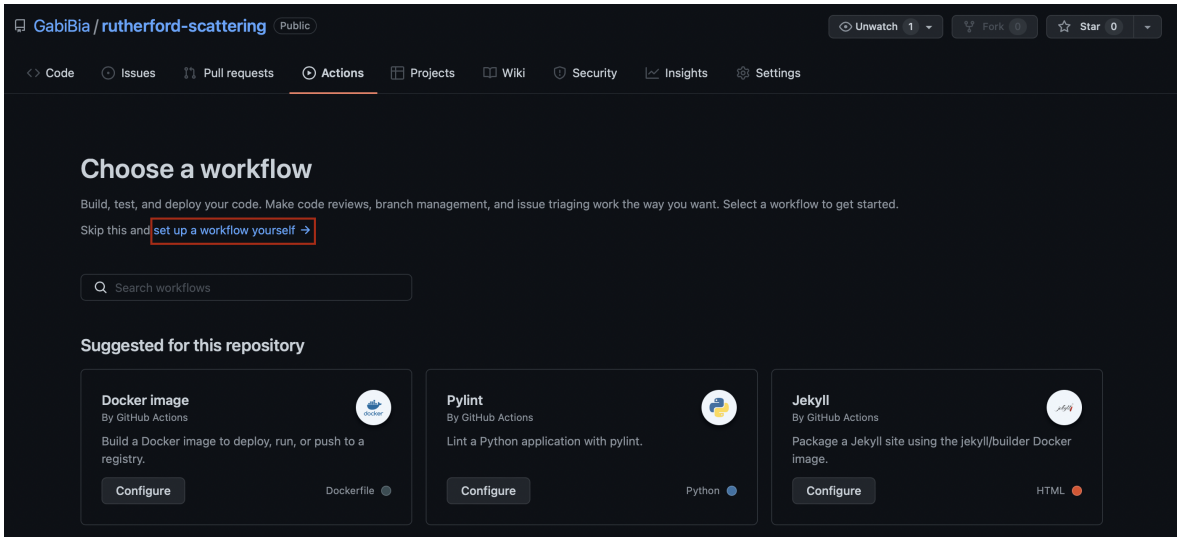
Rysunek 5.7: Tworzenie sekretów w repozytorium - gotowe sekrety.

- **Tworzenie nowej akcji na platformie GitHub** - aby w repozytorium projektu dodać skrypt automatyzujący, należy wejść w zakładkę *Actions*:



Rysunek 5.8: Tworzenie nowej akcji.

Następnie w panelu po lewej stronie należy wybrać opcję *„New workflow”*. Wówczas wyświetlone zostanie widoczne poniżej okno:



Rysunek 5.9: Tworzenie nowej akcji - *new workflow*.

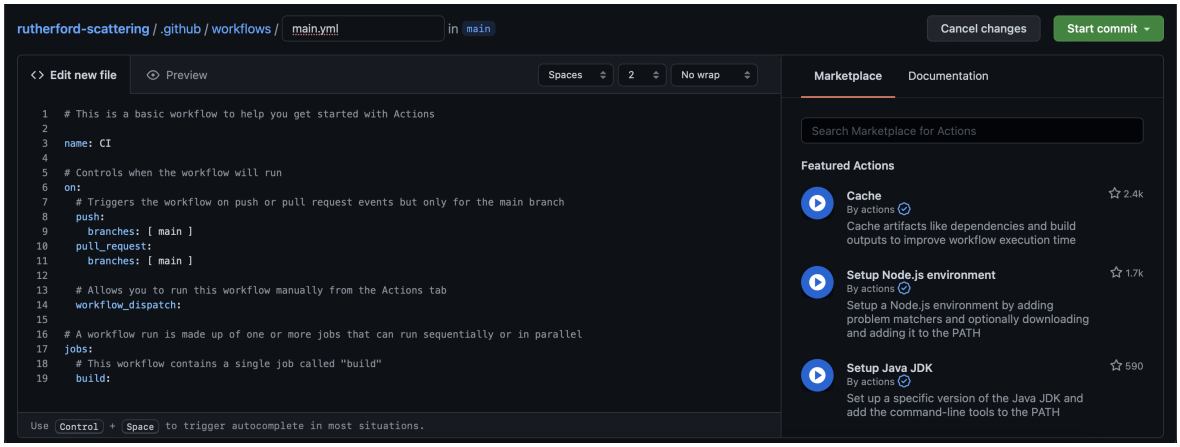
GitHub oferuje wiele gotowych akcji (w tym również akcję *Docker image*), jednak na potrzeby niniejszej pracy napisany został spersonalizowany skrypt automatyzujący (`docker-image.yml`):

```
name: Build a Dockerfile

on:
  push:
    branches:
      - 'main'

jobs:

  build_docker_container:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: Prepare
        id: prep
        run: |
          DOCKER_IMAGE=${{ secrets.DOCKERHUB_USERNAME }}/${
    GITHUB_REPOSITORY#*/}
          VERSION=latest
          # If this is git tag, use the tag name as a docker tag
          if [[ $GITHUB_REF == refs/tags/* ]]; then
            VERSION=${GITHUB_REF#refs/tags/v}
          fi
          TAGS="${DOCKER_IMAGE}:${VERSION}"
          # If the VERSION looks like a version number, assume
    that
          # this is the most recent version of the image and also
          # tag it 'latest'.
          if [[ $VERSION =~ ^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$
    ]]; then
            TAGS="$TAGS,${DOCKER_IMAGE}:latest"
          fi
          # Set output parameters.
          echo ::set-output name=tags::${TAGS}
          echo ::set-output name=docker_image::${DOCKER_IMAGE}

      - name: Set up QEMU
        uses: docker/setup-qemu-action@master
        with:
          platforms: all

      - name: Set up Docker Buildx
        id: buildx
        uses: docker/setup-buildx-action@master

      - name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build and push to DockerHub
        uses: docker/build-push-action@v2
        with:
          builder: ${{ steps.buildx.outputs.name }}
```

```
55            context: .
56            file: ./Dockerfile
57            platforms: linux/amd64, linux/arm64, linux/arm/v6,
      linux/arm/v7
58            push: true
59            tags: ${{ steps.prep.outputs.tags }}
60
```

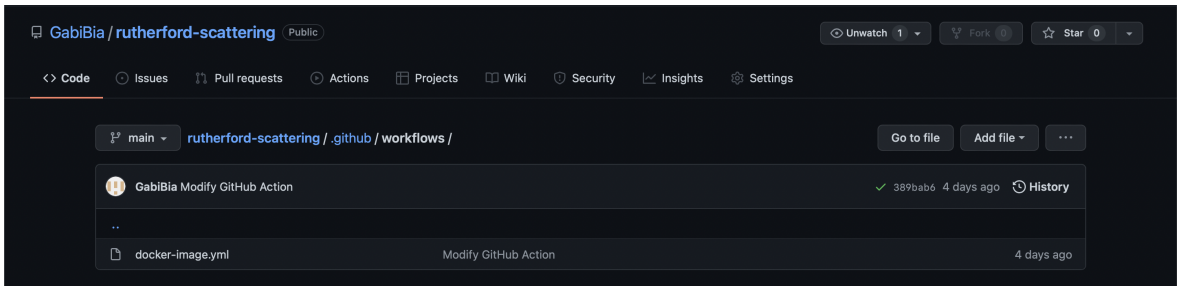
W sekcji `jobs` zdefiniowane zostały instrukcje, jakie skrypt wykonuje. Przygotowuje on najnowszą wersję obrazu, loguje się do konta GitHub przy pomocy stworzonych uprzednio sekretów, a następnie buduje oraz publikuje na platformie Docker Hub aktualną wersję obrazu w taki sposób, aby działała ona na określonych w linijce pięćdziesiątej siódmej platformach. Krok ten zapewnia, że kontener będzie kompatybilny z różnymi rodzajami maszyn (oraz Kubernetes, dzięki czemu możliwa będzie orkiestracja), nie tylko tą, na której powstał.

Wobec tego, że wykorzystany skrypt nie jest domyślny, należy w widocznym na rysunku 5.9 oknie wybrać opcję *„set up a workflow yourself”*, jak zostało to zaznaczone. Wówczas wyświetlą się poniższe opcje:



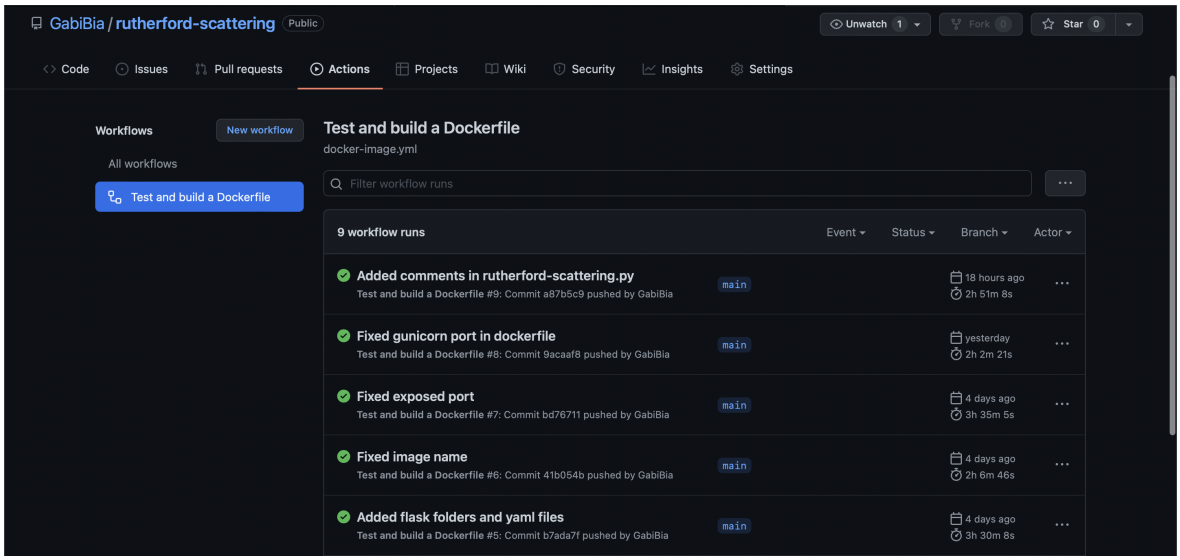
Rysunek 5.10: Tworzenie nowej akcji - dodawanie personalizowanego skryptu.

W powyższym oknie należy zmienić nazwę pliku (w przypadku niniejszej pracy jest to `docker-image.yml`) oraz przykładowy kod zastąpić powyżej przytoczonym, spersonalizowanym skryptem. Wówczas zostanie on dodany do repozytorium w nowym folderze `.github/workflows`:

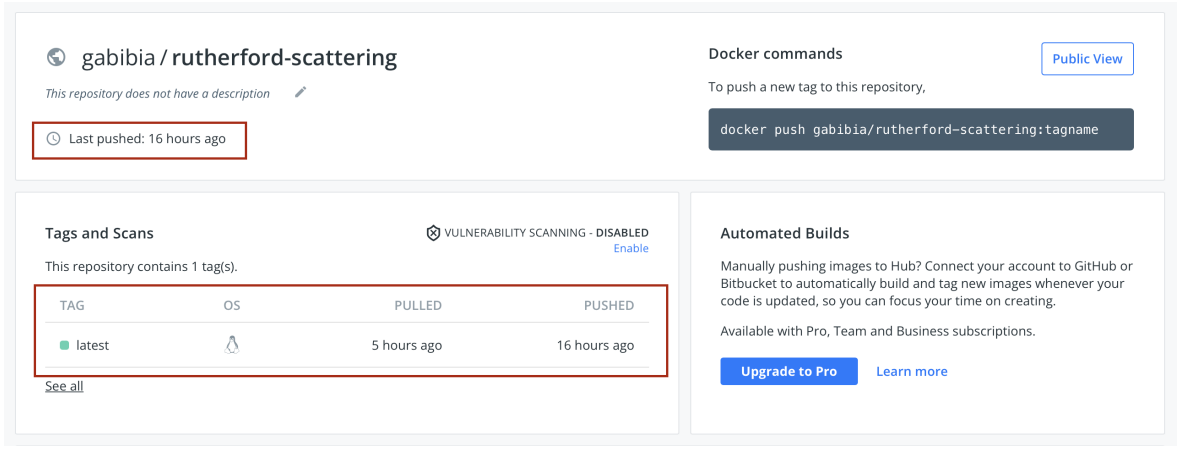


Rysunek 5.11: Tworzenie nowej akcji - ścieżka do skryptu.

- **Testowanie działania akcji** - aby zweryfikować działanie skryptu, należy wprowadzić zmianę w repozytorium (np. w kodzie aplikacji), następnie zastosować komendę `git push`. Wówczas zmiana naniesiona w lokalnym folderze projektu zostanie opublikowana w publicznym repozytorium. Tym samym akcja budująca nowy obraz kontenera powinna zostać automatycznie wykonana. Jeśli tak się stało, w zakładce *Actions* w repozytorium GitHub pojawi się informacja o działającym procesie. Jeśli zakończy się on powodzeniem (jak zostało to przedstawione na rysunku 5.12, dla wielu nowych zmian naniesionych w repozytorium na przestrzeni paru dni), wówczas w repozytorium na platformie Docker Hub zostanie opublikowana nowa wersja obrazu.



Rysunek 5.12: Tworzenie nowej akcji - weryfikowanie działania stworzonej akcji na platformie GitHub.



Rysunek 5.13: Tworzenie nowej akcji - weryfikowanie działania stworzonej akcji na platformie Docker Hub.

5.2 Tworzenie środowiska w chmurze obliczeniowej

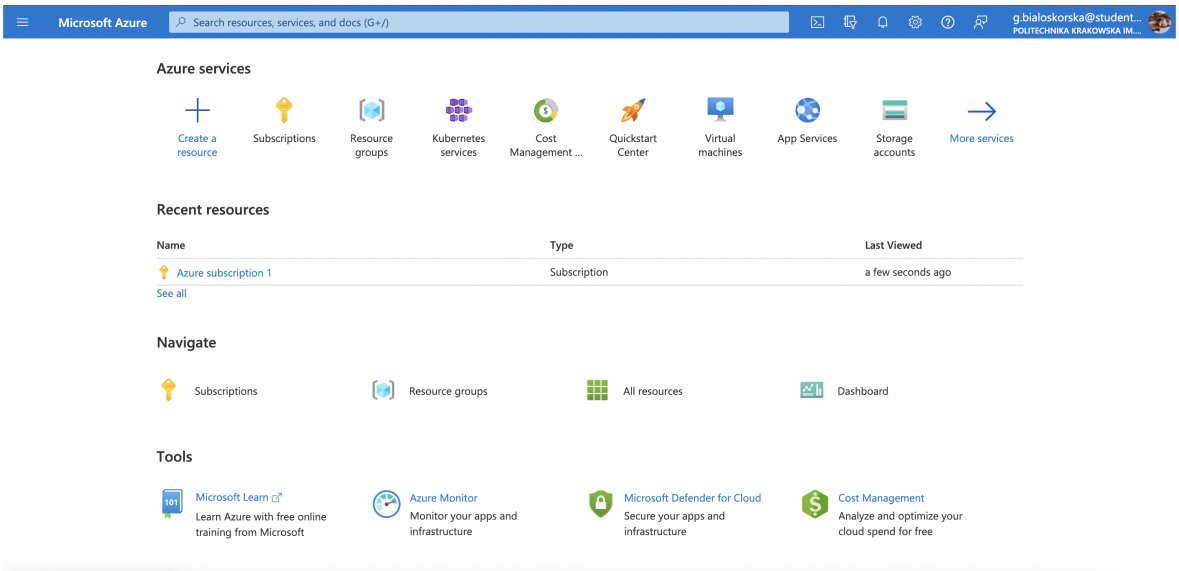
Jak zostało to przedstawione na rysunku 2.3.3.3, infrastruktura w chmurze Microsoft Azure wymaga stworzenia Resource Group oraz serwisów Azure Kubernetes Service i Load Balancer wewnątrz niej.

Niniejsza praca powstała na uczelnianym koncie Microsoft, jednak wymagała stworzenia nowej subskrypcji *Pay As You Go* [8] poza automatycznie przydzieloną subskrypcją studencką, z uwagi na wielkość usługi jaką jest AKS. Subskrypcja na platformie Azure jest podstawową, wyodrębnioną jednostką obejmującą określone zasoby do wykorzystania oraz charakteryzującą się danym modelem płatności. Infrastruktura pracy powstała w subskrypcji o nazwie *Azure Subscription 1*, w modelu opłaty za wykorzystane zasoby.

Ponieważ utrzymanie usługi AKS w dłuższej perspektywie czasu wymaga dużych nakładów finansowych, z tego powodu powstałe w poniższych podrozdziałach serwisy oraz Resource Group zostały usunięte po jej prawidłowym wdrożeniu i opisaniu w niniejszej pracy.

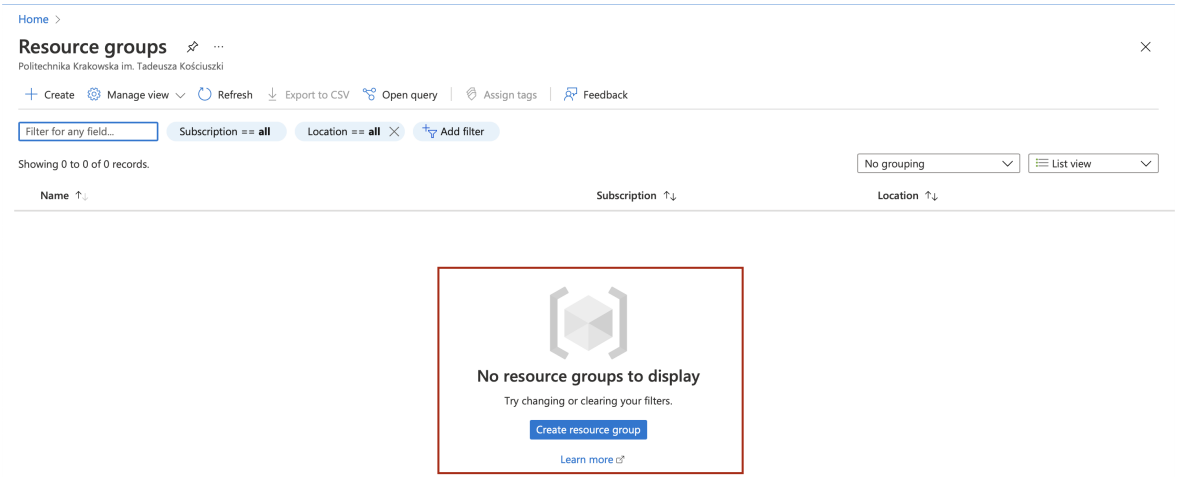
5.2.1 Tworzenie Resource Group

Po zalogowaniu się do konta Microsoft na stronie [9] wyświetlona zostaje strona główna Azure Portal:



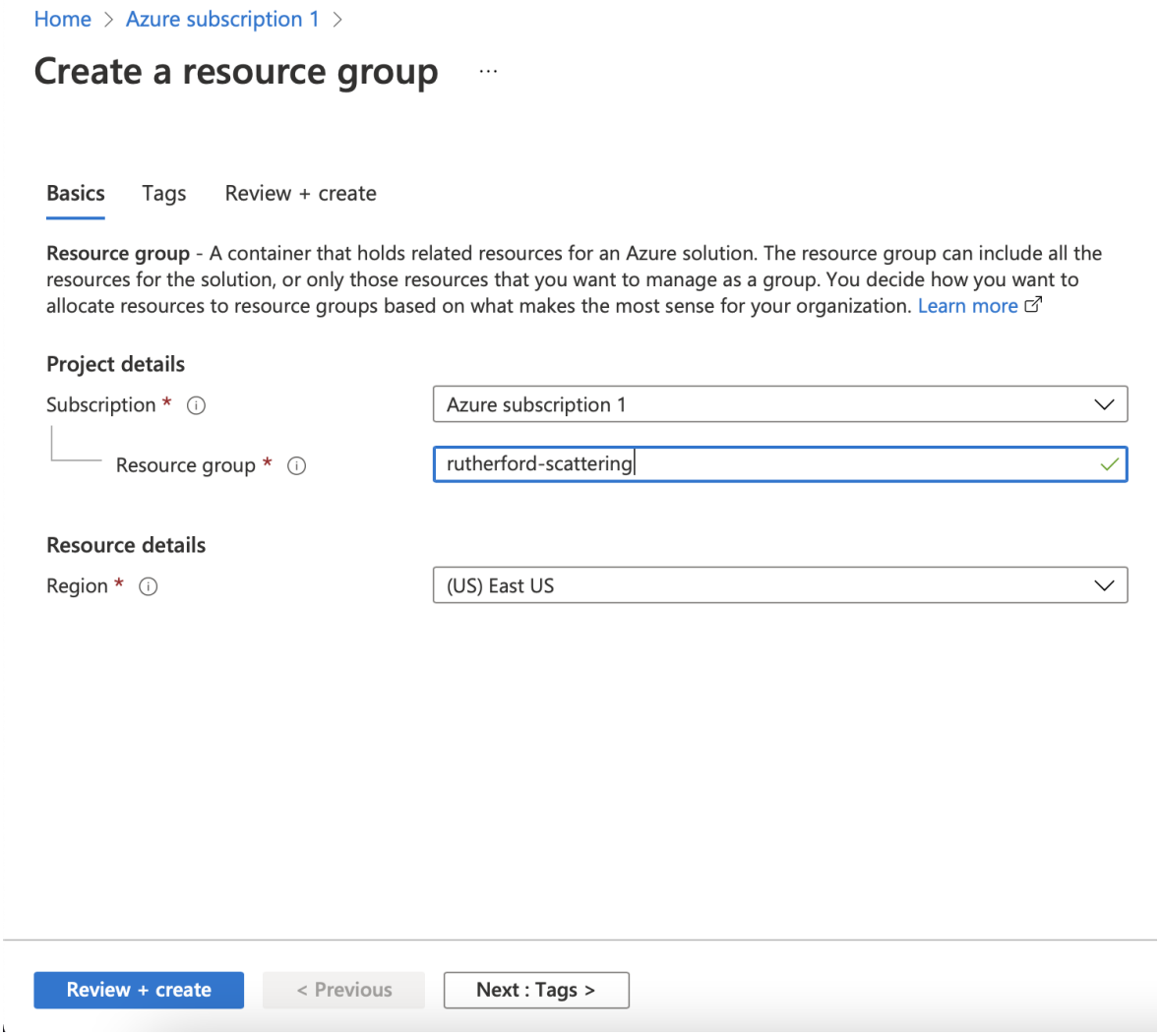
Rysunek 5.14: Azure Portal - strona główna.

Aby stworzyć nową Resource Group, należy na pasku *Azure Services* wybrać zakładkę *Resource Group*, a następnie *Create resource group*, jak poniżej:



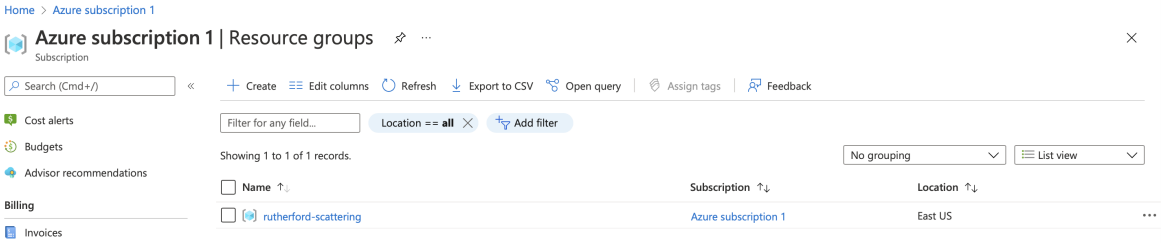
Rysunek 5.15: Tworzenie nowej Resource Group.

W wyświetlonym oknie należy zdefiniować subskrypcję oraz region, w których ma ona powstać oraz jej nazwę. W niniejszej pracy są to następujące parametry:



Rysunek 5.16: Parametry nowej Resource Group.

Resource Group wykorzystana na potrzeby niniejszej pracy nie wymaga definiowania tagów, wobec czego na dole okna należy wybrać opcję *Review + create*. Po chwili wewnątrz subskrypcji widoczna będzie nowa Resource Group:



Rysunek 5.17: Weryfikacja istnienia nowej Resource Group.

5.2.2 Tworzenie Azure Kubernetes Service i Load Balancer

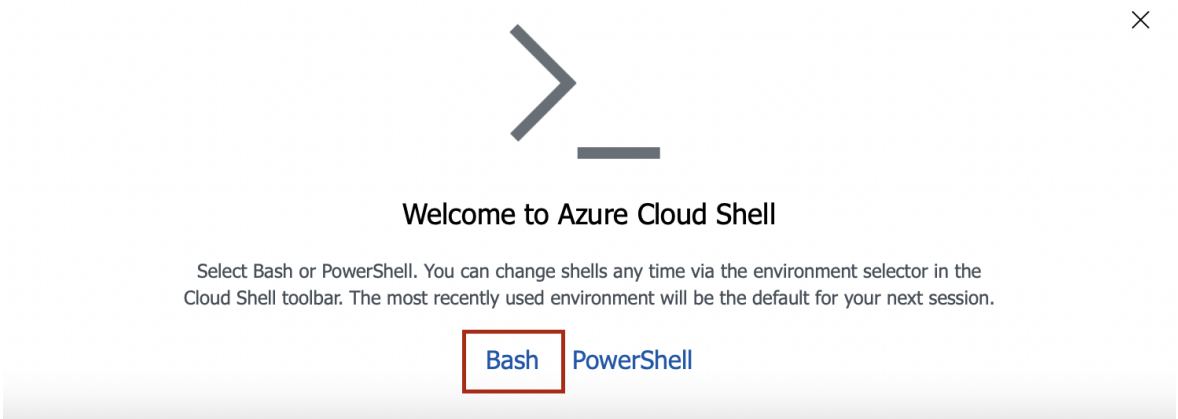
W portalu Azure serwisy stworzyć można na dwa sposoby: bezpośrednio w portalu, będącym interfejsem użytkownika (jak w przypadku Resource Group) lub za pośrednictwem wiersza poleceń *Cloud Shell*. Jest to terminal, uruchamiany z poziomu portalu Azure. W niniejszej pracy wykorzystano wiersz poleceń.

5.2.2.1 Uruchomienie Cloud Shell

Aby wykorzystać Cloud Shell, należy wybrać jego ikonę na górnym pasku narzędzi w portalu Azure [9]:

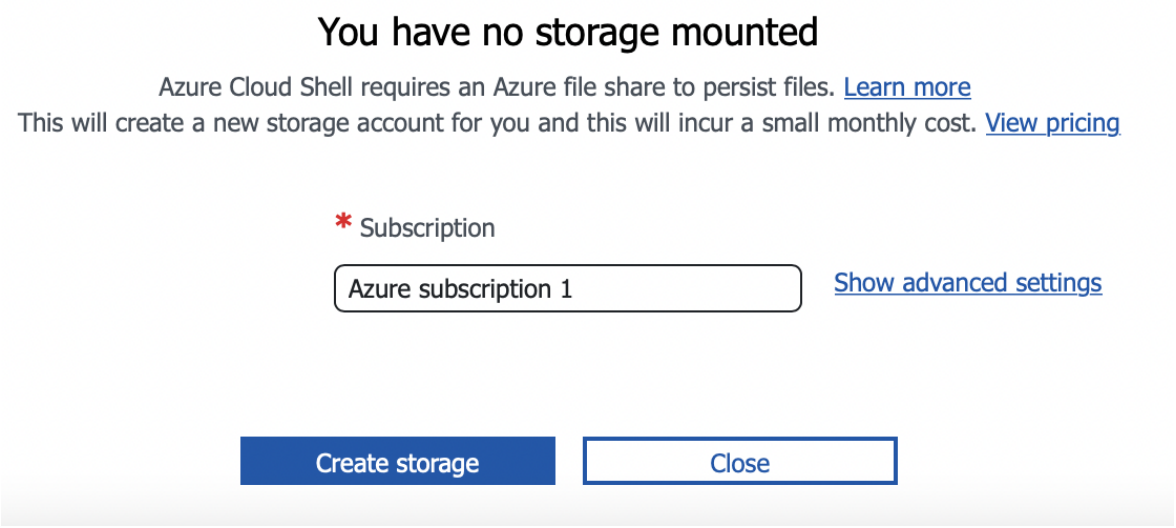
Rysunek 5.18: Uruchamianie Cloud Shell z paska narzędzi portalu Azure.

Wówczas na dole okna wyświetli się możliwość wyboru, w jakim języku użytkownik chce pracować korzystając z wiersza poleceń. W niniejszej pracy wykorzystany został Bash.



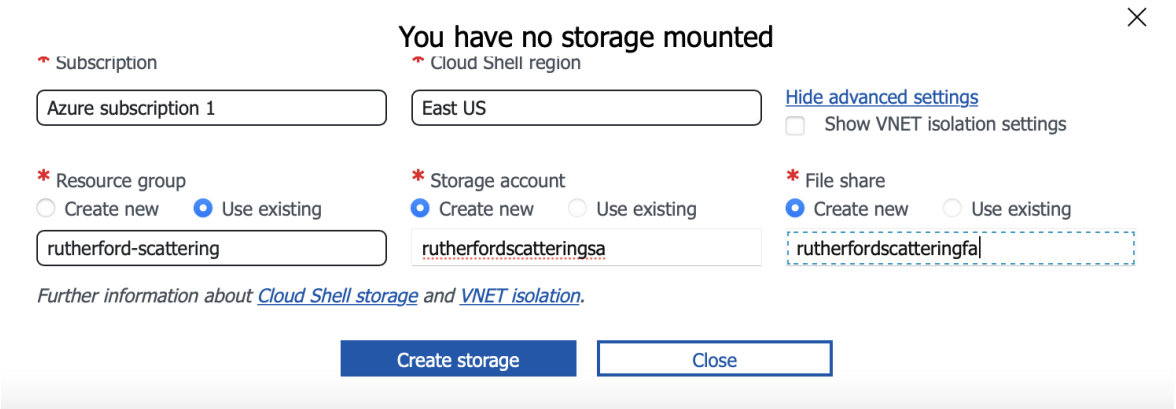
Rysunek 5.19: Wybór języka wiersza poleceń Cloud Shell.

Cloud Shell wymaga stworzenia nowego Storage Account (magazynu chmury Azure), w którym będzie on przechowywać wszystkie pliki stworzone podczas pracy w wierszu poleceń. W niniejszym oknie należy zatem wybrać odpowiedną subskrypcję oraz opcję *Show advanced settings*:



Rysunek 5.20: Tworzenie Storage Account dla Cloud Shell.

Następnie należy wykorzystać istniejącą już Resource Group oraz stworzyć nowy Storage Account i File Share, nadając im unikatowe w obrębie portalu Azure nazwy:



Rysunek 5.21: Definiowanie parametrów Storage Account dla Cloud Shell.

Mając zdefiniowane wszystkie wymagane parametry, należy wybrać opcję *Create storage* i zaczekać, aż wiersz poleceń zostanie autoamtycznie uruchomiony na dole okna portalu Azure.

5.2.2.2 Tworzenie Azure Kubernetes Service

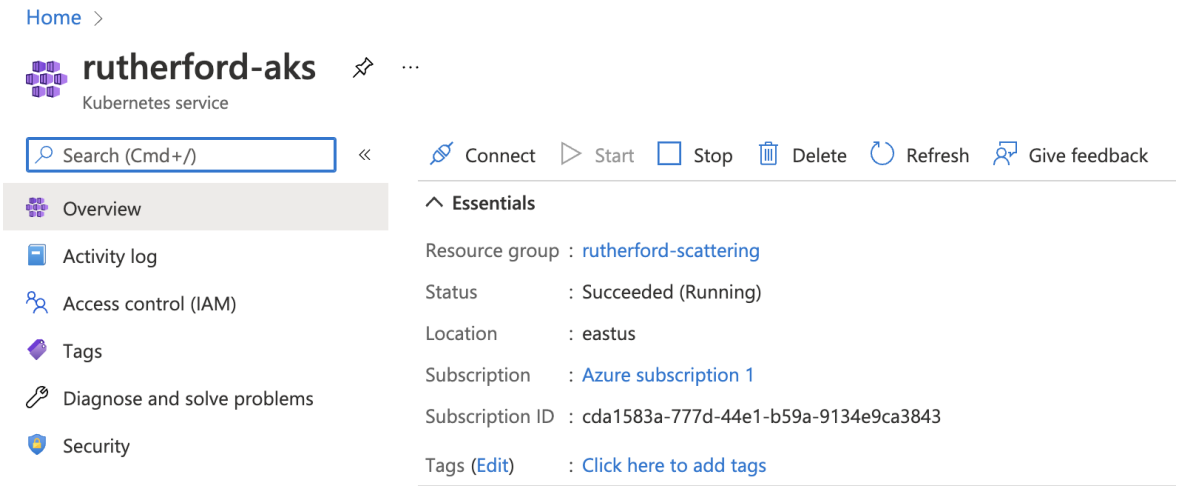
Aby stworzyć Azure Kubernetes Service z poziomu Cloud Shell, należy w powstałym wierszu poleceń wykorzystać komendę `RG=rutherford-scattering` (przypisując zmiennej `RG` nazwę stworzonej wcześniej Resource Group), a następnie komendę `az aks create` z odpowiednimi flagami, jak poniżej:

```
gabriela@Azure:~$ RG=rutherford-scattering
gabriela@Azure:~$ az aks create --resource-group $RG --name rutherford-aks --node-count 3 --generate-ssh-keys --node-vm-size Standard_B2s --enable-ma
naged-identity
```

Rysunek 5.22: Tworzenie AKS z poziomu wiersza poleceń.

Flaga `--resource-group` definiuje istniejącą Resource Group, w której powstać ma AKS, `--name` określa jego nazwę, `--node-count` określa ilość węzłów, z jakich składać ma się klaster, `--generate-ssh-keys` generuje klucze ssh (umożliwiające uwierzytelnianie z węzłami klastra), `--node-vm-size` definiuje rodzaj maszyn wirtualnych, które zostaną wykorzystane jako węzły (ich nazewnictwo określone jest przez Microsoft), `--enable-managed-identity` jest konieczne, aby klaster mógł zostać połączony z Load Balancer.

Po sukcesywnym stworzeniu klastra, można zweryfikować jego istnienie wyszukując go wewnątrz Resource Group `rutherford-scattering` lub bezpośrednio w oknie wyszukiwania na platformie Azure. Jak widać na załączonym rysunku, AKS powstał i jest w stanie *running*:



Rysunek 5.23: Weryfikacja istnienia stworzonego AKS.

5.2.2.3 Wdrożenie aplikacji w Azure Kubernetes Service

Aby na stworzonym klastrze AKS wdrożyć powstałą w rozdziale 4 aplikację, należy wykorzystać plik konfiguracyjny (manifest). Jego struktura została opisana w rozdziale 2.3.3.2. Manifest wykorzystany w niniejszej pracy znajduje się w repozytorium [106] pod nazwą `deployment.yaml`. Jego kod wygląda następująco:

```
1  apiVersion: apps/v1
2  # The type of workload we are creating
3  kind: Deployment
4  metadata:
5    # Name of deployment - Required
6    name: rutherford-scattering-deployment
7  spec:
8    replicas: 2
9    selector:
10     matchLabels:
11       app: rutherford-scattering
12   # Pod template which describes the pod you want to deploy
13   template:
14     metadata:
15       # Used to logically group pods together
16       labels:
17         app: rutherford-scattering
18     # Specific details about the containers in the Pod
19     spec:
20       containers:
21         - name: rutherford-scattering-container
22           # Docker Hub image to use
23           image: gabibia/rutherford-scattering
24           # Define ports to expose
25           ports:
26             - containerPort: 5000
27               # Reference name of port
28               name: http
29           resources:
30             # Minimum amount of resources we want
31             requests:
32               cpu: 100m
33               memory: 128Mi
34             # Maximum amount of resources we want
35             limits:
36               cpu: 250m
37               memory: 256Mi
```

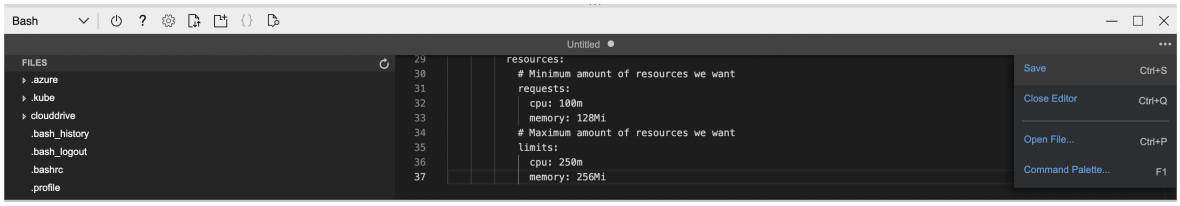
Definiuje on, że jest to wdrożenie aplikacji o dwóch replikach, na podstawie obrazu znajdującego się w repozytorium Docker Hub `gabibia/rutherford-scattering`, udostępnionego na porcie `5000` oraz określa limity zasobów (maszyn wirtualnych, będących węzłami klastra), jakie wdrożona aplikacja może wykorzystywać.

Aby powyższy manifest wdrożyć w klastrze, należy się z nim połączyć komendą `az aks get credentials ––resource-group $RG ––name rutherford-aks`. Połączenie z klastrem można sprawdzić stosując `kubectl get nodes` - jeśli się powiodło, wyświetlone zostaną jego działające węzły, jak poniżej:

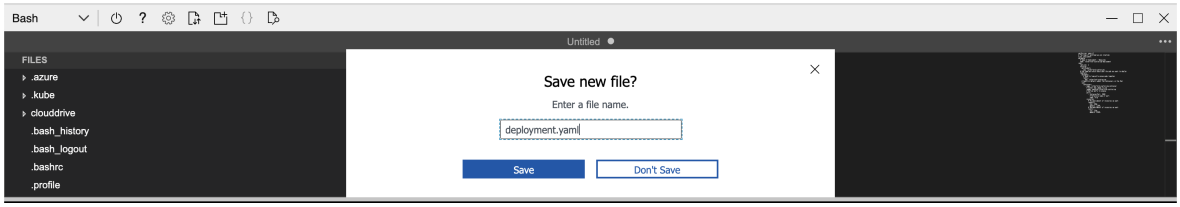
```
gabriela@Azure:~$ az aks get-credentials --resource-group rutherford-scattering --name rutherford-aks
Merged "rutherford-aks" as current context in /home/gabriela/.kube/config
gabriela@Azure:~$ kubectl get nodes
NAME                                STATUS   ROLES   AGE     VERSION
aks-agentpool-32645555-vmss000000   Ready    agent   2m11s   v1.21.7
aks-agentpool-32645555-vmss000001   Ready    agent   2m2s    v1.21.7
aks-agentpool-32645555-vmss000002   Ready    agent   2m12s   v1.21.7
gabriela@Azure:~$
```

Rysunek 5.24: Połączenie z klastrem z poziomu wiersza poleceń.

Po połączeniu z klastrem można przystąpić do wdrożenia manifestu. W tym celu wykorzystuje się komendę `touch NAZWA PLIKU` (`touch deployment.yaml`) lub symbol `{}` na pasku Cloud Shell. Dzięki temu otwarte zostanie okno edycji pliku, w którym należy wkleić powyższy kod manifestu i zapisać go (z odpowiednią nazwą, jeśli nie został on stworzony komendą `touch`):



Rysunek 5.25: Wdrożenie manifestu.



Rysunek 5.26: Zapisywanie manifestu z odpowiednią nazwą.

Wdrożenie zapisanego manifestu możliwe jest dzięki wykorzystaniu komendy `kubectl apply -f deployment.yaml`. Jeśli wdrożenie powiodło się, w wierszu poleceń wyświetlony zostanie komunikat zwrotny: `deployment/rutherford-scattering-deployment created`.

5.2.2.4 Tworzenie Load Balancer

Tworzenie Load Balancer przebiega w sposób analogiczny do wdrożenia aplikacji. Gotowy plik konfiguracyjny Load Balancer znajduje się w repozytorum pracy [106] pod nazwą `loadbalancer.yaml`. Wygląda on następująco:

```
apiVersion: v1
# The type of workload we are creating
kind: Service
metadata:
  # Name of Service - Required
  name: rutherford-scattering-loadbalancer
# Specific details about the Service
spec:
  # Type of Service to be deployed
  type: LoadBalancer
  ports:
    - port: 5000
  # Used to tell the Service which Pods to associate with
  selector:
    app: rutherford-scattering
```

W tym przypadku tworzony jest nowy serwis, nie wdrożenie aplikacji na istniejącym klastrze, stąd różnica w polu *kind*. Analogicznie powyższy kod należy wkleić do pliku `loadbalancer.yaml`, stworzonego komendą `touch` i zapisać, a następnie uruchomić komendą `kubectl apply -f loadbalancer.yaml`. Po otrzymaniu komunikatu `service/rutherford-scattering-loadbalancer created`, należy wykorzystać komendę `kubectl get service rutherford-scattering-loadbalancer --watch`, która wyświetli adres strony (`EXTERNAL IP`), na którym działa wdrożona na klastrze aplikacja:

```
gabriela@Azure:~$ kubectl apply -f loadbalancer.yaml
service/rutherford-scattering-loadbalancer created
gabriela@Azure:~$ kubectl get service rutherford-scattering-loadbalancer --watch
NAME                                TYPE           CLUSTER-IP     EXTERNAL-IP
rutherford-scattering-loadbalancer  LoadBalancer   10.0.208.141   20.84.16.219
```

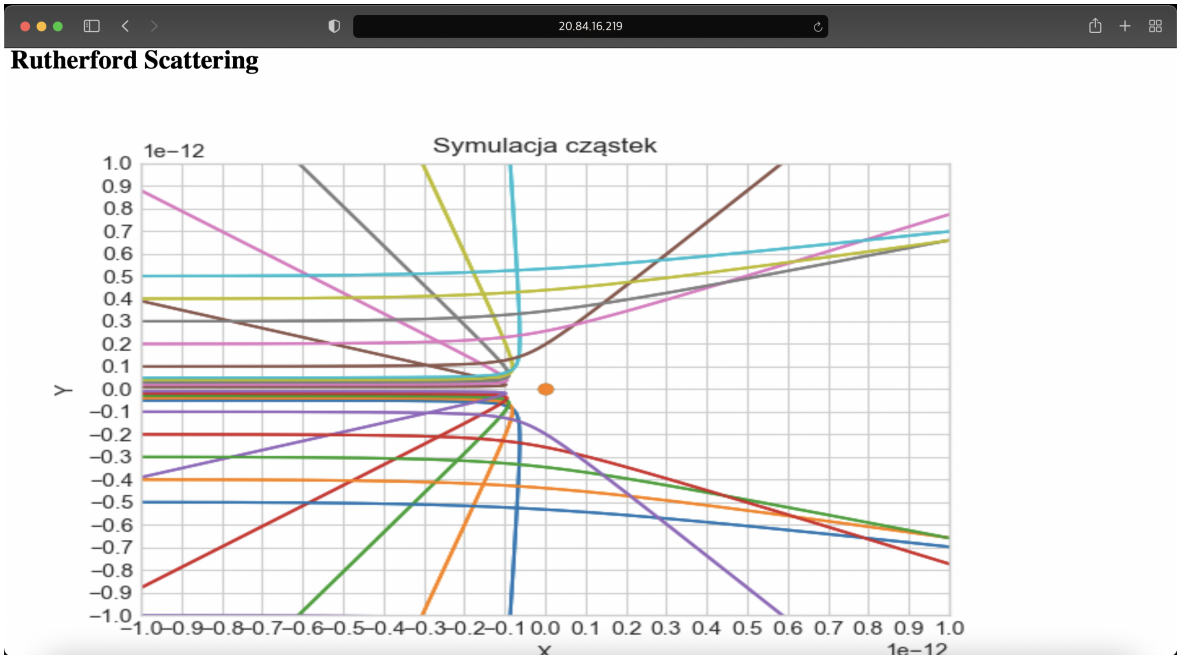
Rysunek 5.27: Tworzenie Load Balancer i wyświetlanie adresu strony, na którym działa wdrożona aplikacja.

Wklejenie powyższego adresu strony w oknie przeglądarki skutkuje uruchomieniem w nim aplikacji.

Rozdział 6

Wyniki

Wynikiem niniejszej pracy jest skonteneryzowana symulacja eksperymentu Rutherforda, napisana w języku Python, dostępna w przeglądarce internetowej pod adresem `20.84.16.219`. Aplikacja ta została uruchomiona w chmurze Microsoft Azure, dzięki wykorzystaniu usługi Azure Kubernetes Service oraz udostępniona pod określonym adresem IP dzięki Load Balancer. Wygląda ona następująco:



Rysunek 6.1: Wynik pracy - skonteneryzowana aplikacja działająca w przeglądarce.

Rozdział 7

Podsumowanie

Celem niniejszej pracy było stworzenie systemu webowego, użytecznego w tworzeniu symulacji układów fizycznych oraz ich modelowaniu. Cel ten został spełniony dzięki napisaniu aplikacji w języku Python, w oparciu o biblioteki Matplotlib, NumPy oraz Flask. Gotowa aplikacja została skonteneryzowana, a następnie wdrożona w infrastrukturze chmurowej, dzięki wykorzystaniu usług chmury publicznej Microsoft Azure.

Stworzone na potrzeby pracy repozytorium, zawierające wszystkie wykorzystane w niej pliki konfiguracyjne, może zostać wykorzystane również w przypadku innych symulacji - szczególnie tych wymagających wykonywania skomplikowanych obliczeń, a zatem dużej mocy obliczeniowej. Konteneryzacja umożliwia uruchomienie ich na dowolnym środowisku, bez potrzeby zwiększania mocy obliczeniowej danego komputera. Jak zostało sprawdzone po ukończeniu części praktycznej, skonteneryzowana symulacja wykorzystana w niniejszej pracy działa również na klastrze, którego węzłami są urządzenia Raspberry Pi. Funkcjonuje na nich lekka wersja oprogramowania Kubernetes, zwana *K3S* [47].

Technologie informatyczne są obecnie jednym z narzędzi pracy współczesnych fizyków. Niniejsza praca ukazuje możliwości ich wykorzystania na małą skalę, jednak podobne rozwiązania są skutecznie implementowane także w ogromnych ośrodkach badawczych. CERN stosuje chmurę prywatną, usługi chmury publicznej oraz automatyzację procesów w celu zapewnienia wystarczającej mocy obliczeniowej podczas przesyłu danych z Wielkiego Zderzacza Hadronów [46]. Infrastruktury informatyczne oparte na konteneryzacji oraz chmurze obliczeniowej udostępniają badaczom ogromne możliwości i w znacznym stopniu usprawniają ich pracę, jednocześnie oferując bezpieczeństwo, niskie koszty utrzymania zasobów sprzętowych oraz opcję uruchamiania symulacji w dowolnych środowisku.

Bibliografia

[1] Akademia j-labs - Jakub Bujny. *Kubernetes bez tajemnic*. 2019. URL: https://www.youtube.com/watch?v=ysk_2w_diKY&t=1218s. (dostęp: 23.11.2021).

[2] Adam i Przemysław Ćwik. *Mikrousługi zamiast monolitu. Dlaczego warto uwolnić się od aplikacji monolitycznej (i jak to zrobić)*. 2021. URL: https://kissdigital.com/pl/blog/mikrouslugi-zamiast-monolitu. (dostęp: 02.12.2021).

[3] Bethel Afework i Jason Donev. *Energy Education - Electric charge*. 2018. URL: https://energyeducation.ca/encyclopedia/Electric_charge. (dostęp: 20.09.2021).

[4] John Arundel i Justin Domingus. *Kubernetes - rozwiązania chmurowe w świecie DevOps*. Gliwice: Helion, 2020.

[5] *Azure Kubernetes Service*. URL: https://azure.microsoft.com/pl-pl/services/kubernetes-service/#getting-started. (dostęp: 23.12.2021).

[6] *Azure Kubernetes Service Logo*. URL: https://github.com/Azure/AKS. (dostęp: 23.12.2021).

[7] *Azure Load Balancer*. URL: https://azure.microsoft.com/pl-pl/services/load-balancer/#overview. (dostęp: 23.12.2021).

[8] *Azure Pay As You Go Subscription*. URL: https://azure.microsoft.com/en-in/pricing/purchase-options/pay-as-you-go/. (dostęp: 29.12.2021).

[9] *Azure Portal*. URL: https://portal.azure.com. (dostęp: 29.12.2021).

[10] *Azure Resource Group*. URL: https://docs.microsoft.com/pl-pl/azure/azure-resource-manager/management/manage-resource-groups-portal. (dostęp: 23.12.2021).

[11] Dast Blog. *Co to jest CI/CD?* 2020. URL: http://dast.webd.pl/co-to-jest-ci-cd/. (dostęp: 30.11.2021).

[12] Dast Blog. *Podstawy Dockera*. 2018. URL: http://dast.webd.pl/podstawy-dockera/. (dostęp: 23.11.2021).

[13] Brendan Burns i David Oppenheimer. „Design patterns for container-based distributed systems". W: (2005). DOI: https://static.googleusercontent.com/media/research.google.com/pl//pubs/archive/45406.pdf. (dostęp: 25.11.2021).

[14] CERN. *The Large Hadron Collider*. 2021. URL: https://home.cern/science/accelerators/large-hadron-collider. (dostęp: 27.09.2021).

[15] Alan Chodos, Jennifer Ouellette i Ernie Tretkoff. „May, 1911: Rutherford and the Discovery of the Atomic Nucleus". W: *APS News* 15.5 (2006). DOI: https://www.aps.org/publications/apsnews/200605/history.cfm. (dostęp: 03.10.2021).

[16] Computerworld. *AWS vs Microsoft Azure vs Google Cloud. Najlepsze IaaS dla biznesu*. 2019. URL: https://www.computerworld.pl/news/AWS-vs-Microsoft-Azure-vs-Google-Cloud-Najlepsze-IaaS-dla-biznesu,414888.html. (dostęp: 16.11.2021).

[17] Computerworld. *DevOps – a co to takiego?* URL: https://www.computerworld.pl/news/DevOps-a-co-to-takiego,403374.html. (dostęp: 30.11.2021).

[18] Robert P. Crease. *The Prism and the Pendulum: The Ten Most Beautiful Experiments in Science*. Random House, 2004.

[19] *Docker Hub*. URL: https://hub.docker.com. (dostęp: 28.12.2021).

[20] Docker Docs. *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/. (dostęp: 28.12.2021).

[21] Marcin Dolecki. „Nie-kwantowe teorie budowy atomu dyskutowane na łamach tygodnika "Wszechświat"w latach 1882-1914". W: *Analecta* 27.1-2(27-28) (2005), s. 167–184. DOI: https://bazhum.muzhp.pl/media/files/Analecta_studia_i_materialy_z_dziejow_nauki/Analecta_studia_i_materialy_z_dziejow_nauki-r2005-t14-n1_2_(27_28)/Analecta_studia_i_materialy_z_dziejow_nauki-r2005-t14-n1_2_(27_28)-s167-184/Analecta_studia_i_materialy_z_dziejow_nauki-r2005-t14-n1_2_(27_28)-s167-184.pdf. (dostęp: 11.10.2021).

[22] Domenomania.pl. *Co to jest port protokołu?* URL: https://domenomania.pl/centrum-wiedzy/co-to-jest-port-protokolu. (dostęp: 28.12.2021).

[23] Robert Martin Eisberg. *Fundamentals of Modern Physics*. John Wiley & Sons, Inc (Wiley), 1961.

[24] Wikipedia The Free Encyclopedia. *Amazon Web Services*. 2021. URL: https://en.wikipedia.org/wiki/Amazon_Web_Services. (dostęp: 16.11.2021).

[25] Wikipedia The Free Encyclopedia. *Docker (container engine) logo*. 2021. URL: https://pl.m.wikipedia.org/wiki/Plik:Docker_(container_engine)_logo.png. (dostęp: 19.11.2021).

[26] *Encyklopedia PWN - Gluony*. URL: https://encyklopedia.pwn.pl/haslo/;3905959. (dostęp: 12.10.2021).

[27] *Encyklopedia PWN - Synchrotron*. URL: https://encyklopedia.pwn.pl/haslo/;3982062. (dostęp: 14.10.2021).

[28] *Encyklopedia PWN - Wzór Rutherforda*. URL: https://encyklopedia.pwn.pl/haslo/Rutherforda-wzor;3970206.html. (dostęp: 04.10.2021).

[29] „First M87 Event Horizon Telescope Results. III. Data Processing and Calibration". W: (2019). DOI: https://iopscience.iop.org/article/10.3847/2041-8213/ab0c57. (dostęp: 19.12.2021).

[30] *Flask*. URL: https://flask.palletsprojects.com/en/2.0.x/. (dostęp: 19.12.2021).

[31] Cloud Native Computing Foundation. URL: https://www.cncf.io. (dostęp: 03.12.2021).

[32] Gartner. *Magic Quadrant for Cloud Infrastructure and Platform Services*. 2021. URL: https://www.gartner.com/doc/reprints?id=1-271OE4VR&ct=210802&st=sb. (dostęp: 15.11.2021).

[33] Cassie Gates. *Shapes of Atomic Orbitals — Overview & Examples*. 2020. URL: https://www.expii.com/t/shapes-of-atomic-orbitals-overview-examples-8323. (dostęp: 22.09.2021).

[34] *Get Docker*. URL: https://docs.docker.com/get-docker/. (dostęp: 28.12.2021).

[35] Noah Gift, Kennedy Behrman, Alfredo Deza i Grig Gheorghiu. *Python dla DevOps - Naucz się bezlitośnie skutecznej automatyzacji*. Gliwice: Helion, 2021.

[36] *GitHub*. URL: https://github.com. (dostęp: 15.12.2021).

[37] *GitHub Actions*. URL: https://github.com/features/actions. (dostęp: 27.12.2021).

[38] *GitHub Logo*. URL: https://logos-world.net/github-logo/. (dostęp: 23.12.2021).

[39] *Glowscript*. URL: https://www.glowscript.org. (dostęp: 16.08.2021).

[40] Krzysztof Golec–Biernat. *Wstęp do mechaniki kwantowej*. Rzeszów: Uniwersytet Rzeszowski, Instytut Fizyki Jądrowej PAN, 2015. URL: https://annapurna.ifj.edu.pl/~golec/data/teaching/files/wyklady/mk.pdf. (dostęp: 11.09.2021).

[41] *Gunicorn*. URL: https://github.com/benoitc/gunicorn. (dostęp: 28.12.2021).

[42] Marcel Guzenda. *Co to jest wirtualizacja? Typy, rodzaje i korzyści wirtualizacji*. 2021. URL: https://www.youtube.com/watch?v=7C_m5xvafLM. (dostęp: 04.11.2021).

[43] David Halliday, Robert Resnick i Jearl Walker. *Podstawy fizyki*. Warszawa: Wydawnictwo Naukowe PWN, 2005. Tom 5.

[44] Hans Christoph Wolf Hermann Haken. *Atomy i kwanty - wprowadzenie do współczesnej spektroskopii atomowej*. 2 wyd. Warszawa: Wydawnictwo Naukowe PWN, 2002.

[45] IBM. *Docker*. 2021. URL: https://www.ibm.com/pl-pl/cloud/learn/docker. (dostęp: 23.11.2021).

[46] Adam Jadczak. *Jak CERN zarządza jedną z największych instalacji OpenStack*. 2015. URL: https://itwiz.pl/jak-cern-zarzadza-jedna-najwiekszych-instalacji-openstack/. (dostęp: 30.12.2021).

[47] *K3S - Lightweight Kubernetes*. URL: https://k3s.io. (dostęp: 10.01.2021).

[48] Zbigniew Kąkol i Jan Żukrowski. *e-Fizyka - Podstawy Fizyki*. 2002-2020. URL: http://home.agh.edu.pl/~kakol/efizyka/w22/extra22a.html. (dostęp: 14.10.2021).

[49] Anna Kerdan. *Microsoft unveils a clean logo for the Azure product*. 2021. URL: https://logos-world.net/microsoft-unveils-a-clean-logo-for-the-azure-product/s. (dostęp: 16.11.2021).

[50] Jan Kozubowski. *Encyklopedia PWN - Mikroskop Elektronowy Skaningowy*. URL: https://encyklopedia.pwn.pl/haslo/;3941325. (dostęp: 24.09.2021).

[51] Jan Kozubowski. *Encyklopedia PWN - Nanotechnologia*. URL: https://encyklopedia.pwn.pl/haslo/nanotechnologia;3945658.html. (dostęp: 24.09.2021).

[52] Jarosław Krochmalski. *Docker - Projektowanie i wdrażanie aplikacji*. Gliwice: Helion, 2017.

[53] Jan Królikowski. *VI.5 Zderzenia i rozpraszanie. Przekrój czynny. Wzór Rutherforda i odkrycie jądra atomowego*. 2005. URL: https://www.fuw.edu.pl/~krolikow/fizyka1/vi5_zderzenia_przekroj_czynny.pdf. (dostęp: 26.09.2021).

[54] Kubernetes. *Kubernetes - Naucz się podstaw*. 2021. URL: https://kubernetes.io/pl/docs/tutorials/kubernetes-basics/_print/. (dostęp: 25.11.2021).

[55] Kubernetes. *Kubernetes - Production-Grade Container Orchestration*. 2021. URL: https://kubernetes.io. (dostęp: 25.11.2021).

[56] Kubernetes. *Składniki Kubernetesa*. 2020. URL: https://v1-18.docs.kubernetes.io/pl/docs/concepts/overview/components/. (dostęp: 27.11.2021).

[57] Kubernetes. *Understanding Kubernetes Objects*. 2021. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. (dostęp: 25.11.2021).

[58] Marianna Nesterowicz Lech Chmurzyński. *Kompendium z Chemii*. Gdańsk: Wydawnictwo Uniwersytetu Gdańskiego, 2009.

[59] Łukasz Łada, Łukasz Adamowski, Ewa Droste i Ludwik Dobrzyński. *Modele atomu od Demokryta do Bohra*. 2013. URL: http://www.ncbj.edu.pl/modele-atomu-od-demokryta-do-bohra. (dostęp: 05.09.2021).

[60] Andrzej Łukasik. *Atom - od greckiej filozofii przyrody do nauki współczesnej*. Lublin: Wydawnictwo Uniwersytetu Marii Curie-Skłodowskiej, 2000.

[61] Andrzej Łukasik. „Ewolucja pojęcia atomu". W: *Otwarte Referarium Filozoficzne* 2.15 (2009). DOI: http://wujzboj.com/orf/. (dostęp: 09.09.2021).

[62] Magnifier. *Konteneryzacja – czym jest i dlaczego staje się tak popularna?* 2019. URL: https://magnifier.pl/konteneryzacja-docker-kubernetes/. (dostęp: 19.11.2021).

[63] Leszek Marcinkowski. *Numeryczne rozwiązywanie równań różniczkowych*. 2011. URL: https://mst.mimuw.edu.pl/wyklady/nrr/wyklad.pdf. (dostęp: 05.01.2022).

[64] *MATLAB*. URL: https://www.mathworks.com/products/matlab.html. (dostęp: 19.12.2021).

[65] *Matplotlib*. URL: https://matplotlib.org. (dostęp: 15.12.2021).

[66] Microsoft. *Co to jest chmura prywatna?* 2021. URL: https://azure.microsoft.com/pl-pl/overview/what-is-a-private-cloud/. (dostęp: 09.11.2021).

[67] Microsoft. *Co to jest chmura publiczna?* 2021. URL: https://azure.microsoft.com/pl-pl/overview/what-is-a-public-cloud/. (dostęp: 09.11.2021).

[68] Microsoft. *Co to jest dostawca usług w chmurze?* 2021. URL: https://azure.microsoft.com/pl-pl/overview/what-is-a-cloud-provider/. (dostęp: 15.11.2021).

[69] Microsoft. *Co to jest infrastruktura jako kod?* 2021. URL: https://docs.microsoft.com/pl-pl/devops/deliver/what-is-infrastructure-as-code. (dostęp: 30.11.2021).

[70] Microsoft. *Jakie są różne typy usług chmury obliczeniowej?* 2021. URL: https://azure.microsoft.com/pl-pl/overview/types-of-cloud-computing/. (dostęp: 09.11.2021).

[71] Damian Naprawa. *Dockerfile – COPY vs ADD*. URL: https://szkoladockera.pl/dockerfile-copy-vs-add/. (dostęp: 28.12.2021).

[72] Damian Naprawa. *Dockerfile – ENTRYPOINT vs CMD*. URL: https://szkoladockera.pl/dockerfile-entrypoint-vs-cmd/. (dostęp: 28.12.2021).

[73] NCODER. *Docker – Docker Hub*. URL: https://ncoder.pl/docker-docker-hub/. (dostęp: 28.12.2021).

[74] Sam Newman. *Budowanie mikrousług - Szybkie wprowadzenie*. Gliwice: Helion, 2015.

[75] Sam Newman. *Od monolitu do mikrousług - Ewolucyjne wzorce przekształcania systemów monolitycznych*. Gliwice: Helion, 2020.

[76] Nordcloud. *Migrating to the Cloud with GCP*. 2021. URL: https://nordcloud.com/migrating-to-cloud-with-gcp/. (dostęp: 16.11.2021).

[77] *NumPy*. URL: https://numpy.org. (dostęp: 19.12.2021).

[78] Agnieszka Obłąkowska-Mucha. *Cząstki elementarne i oddziaływania - II Relatywistyka, zderzenia, rozpady, świetlność akceleratora*. URL: http://home.agh.edu.pl/~amucha/czastki/zima_2019/wyklad_2_relatywistyka.pdf. (dostęp: 27.09.2021).

[79] Politechnika Opolska. *Fizyka Atomowa*. URL: https://b.klimesz.po.opole.pl/wyklady/W8_Fizyka2_ns.pdf. (dostęp: 03.10.2021).

[80] Let's Code Physics. *Rutherford Scattering (A Journey through Modern Physics 4)*. URL: https://www.youtube.com/watch?v=hO-jLfUbfyw&list=PLjeqbAlWDXq1-fwn7YyTvN-IpXrlknjLw&index=2. (dostęp: 15.08.2021).

[81] Jan Pluta. *Jądro atomowe*. URL: http://www.if.pw.edu.pl/~pluta/pl/dyd/mfj/wyklad/w1/segment3/main.htm. (dostęp: 30.09.2021).

[82] Jan Pluta. *Model atomu Thomsona*. URL: http://www.if.pw.edu.pl/~pluta/pl/dyd/lekcje/lekcja13/segment1/main.htm. (dostęp: 30.09.2021).

[83] Szymon Płodowski. *Czym jest Infrastruktura informatyczna? – Część 1*. URL: https://www.devire.pl/infrastruktura-informatyczna-czesc-1/. (dostęp: 30.11.2021).

[84] Kamil Porembiński. *Nowoczesne administrowanie, czyli Infrastruktura jako kod*. 2015. URL: https://thecamels.org/nowoczesne-administrowanie-czyli-infrastruktura-jako-kod/. (dostęp: 01.12.2021).

[85] *Python*. URL: https://www.python.org. (dostęp: 15.12.2021).

[86] *Python Logo*. URL: https://www.python.org/community/logos/. (dostęp: 15.12.2021).

[87] Jothy Rosenberg i Arthur Mateos. *Chmura obliczeniowa. Rozwiązania dla biznesu*. Gliwice: Helion, 2011.

[88] *Rutherford Model*. URL: https://github.com/lev4ek0/rutherford_model. (dostęp: 16.08.2021).

[89] SENDnet. *Migracje systemów informatycznych*. 2013. URL: http://sendnet.pl/?page_id=28. (dostęp: 03.12.2021).

[90] Amazon Web Services. *Our Data Centers*. 2021. URL: https://aws.amazon.com/compliance/data-center/data-centers/. (dostęp: 10.11.2021).

[91] Amazon Web Services. *What is AWS?* 2021. URL: https://aws.amazon.com/what-is-aws/. (dostęp: 16.11.2021).

[92] Jarosław Sobel. *Wirtualizacja - czy to początek końca znanych nam systemów?* 2016. URL: https://www.slideshare.net/yar0/careercon-wirtualizacja-pl. (dostęp: 04.11.2021).

[93] Tomasz Sowiński. „Atom Rutherforda nie może istnieć!" W: *Młody Technik* (2007). DOI: http://www.ifpan.edu.pl/~tomsow/popular/mlody_technik/mt0708.pdf. (dostęp: 10.09.2021).

[94] National Insitute of Standards, Technology (Peter Mell i Timothy Grance). *The NIST Definition of Cloud Computing*. 2011. URL: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf. (dostęp: 05.11.2021).

[95] The Story. *Architektura Oprogramowania*. URL: https://thestory.is/pl/proces/faza-wdrozenia/architektura-oprogramowania/. (dostęp: 02.12.2021).

[96] Prefetch Technologies. *The beginners guide to creating Kubernetes manifests*. 2019. URL: https://prefetch.net/blog/2019/10/16/the-beginners-guide-to-creating-kubernetes-manifests/. (dostęp: 25.11.2021).

[97] *VPython*. URL: https://vpython.org. (dostęp: 15.12.2021).

[98] Łukasz Wadowski i Piotr Masztafiak. *Historia wirtualizacji*. 2008. URL: http://www.virtual-it.pl/artykuly/147-historia-wirtualizacji-cz-1.html. (dostęp: 04.11.2021).

[99] Katarzyna Węgiel. *Git a GitHub. Czym się różnią?* 2021. URL: https://www.kei.pl/blog/git-a-github-czym-sie-roznia. (dostęp: 20.12.2021).

[100] *What's the difference between cloud and virtualization?* 2018. URL: https://www.redhat.com/en/topics/cloud-computing/cloud-vs-virtualization. (dostęp: 05.11.2021).

[101] Dorota Wierzuchowska. *Elementy fizyki jądrowej*. URL: https://slideplayer.pl/slide/812531/. (dostęp: 26.09.2021).

[102] Andreas Wittig i Michael Wittig. *Amazon Web Services w akcji*. 2 wyd. Gliwice: Helion, 2020.

[103] Andrzej Kajetan Wróblewski. *Historia fizyki*. Warszawa: Wydawnictwo Naukowe PWN, 2006.

[104] Peter Young. *The leapfrog method and other "symplectic" algorithms for integrating Newton's laws of motion*. 2014. URL: https://young.physics.ucsc.edu/115/leapfrog.pdf. (dostęp: 08.01.2022).

[105] Sebastian Zawadzki. *Co to jest API? Wszystko o interfejsie programowania aplikacji*. 2020. URL: https://smartbees.pl/blog/co-jest-api-wszystko-o-interfejsie-programowania-aplikacji. (dostęp: 27.11.2021).

[106] *Zdalne repozytorium kodu symulacji*. URL: https://github.com/GabiBia/Rutherford-Scattering.