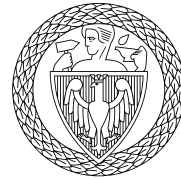


Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Telekomunikacji

Praca dyplomowa magisterska

na kierunku Telekomunikacja
w specjalności Teleinformatyka i Zarządzanie w Telekomunikacji

Propozycja architektury platformy do modelowania i uruchamiania
zamkniętych pętli sterowania w Kubernetes

Andrzej Gawor

Numer albumu 300528

promotor

dr inż. Dariusz Bursztynowski

WARSZAWA 2025

Propozycja architektury platformy do modelowania i uruchamiania zamkniętych pętli sterowania w Kubernetes

Streszczenie. Projekt opisany w niniejszej pracy skupia się na zaproponowaniu oraz przedstawieniu implementacji architektury platformy, która pozwala na modelowanie oraz uruchamianie zamkniętych pętli sterowania w Kubernetes. Genezą projektu jest praca jednego z komitetów ETSI o nazwie "ENI - Experiential Networked Intelligence", która skupia się na ułatwieniu pracy operatora sieci telekomunikacyjnych wykorzystując mechanizmy sztucznej inteligencji w zamkniętych pętlach sterowania. ENI w jednym ze swoich dokumentów dokonuje przeglądu zamkniętych pętli sterowania znanych ludzkości z innych dziedzin.

Naturalnym następnym krokiem jest zaproponowanie platformy, na której operator mógłby takowe pętle projektować oraz uruchamiać. W tym celu zdefiniowano zestaw wymagań oraz założeń dla takiego systemu. Jako środowisko uruchomieniowe wybrano Kubernetes z racji, że jest to system dobrze znany w społeczności oraz sam natywnie używa zamkniętych pętli sterowania. Następnie przeprowadzono obszerną analizę jak za pomocą mechanizmów rozszerzania Kubernetes takich jak "Custom Resources" oraz "Operator" pattern można stworzyć framework umożliwiający modelowanie zamkniętych pętli sterowania. Praca opisuje powstałą platformę, jej architekturę, semantykę składni w definiowanych obiektach, zasady działania, integracje z zewnętrznymi systemami oraz instrukcję jej użytkowania. Omówiona została również implementacja platformy, technologie za nią stojące oraz decyzje podjęte podczas jej powstawania. Finalnie przedstawiono również test działania platformy w praktyce wykorzystując do tego emulator systemu 5G jakim jest Open5GS w połączeniu z UERANSIM. Pracę podsumuje lista wniosków oraz potencjalnych dróg rozwoju platformy.

Słowa kluczowe: Zamknięte pętle sterowania, Kubernetes, Zarządzanie sieciami telekomunikacyjnymi, Automatyzacja, Go, Open5GS, Mikroserwisy

Creation of a platform for designing and running closed control loops in Kubernetes

Abstract. The project described in this thesis focuses on proposing and implementing a reusable architecture (hereafter referred to as "the Platform") that enables the modeling and execution of closed control loops in Kubernetes. The genesis of the project lies in the work of one of the ETSI committees called "ENI - Experiential Networked Intelligence," which aims to simplify the work of telecommunications network operators by leveraging artificial intelligence mechanisms in closed control loops based on metadata-driven and context-aware policies. In one of its documents, ENI reviews closed control loops known to humanity from other fields. A natural next step is to propose a platform on which operators could design and execute such loops.

To achieve this, a set of requirements and assumptions for such a system was defined. Kubernetes was chosen as the runtime environment due to its widespread adoption in the community and its inherent use of closed control loops. An extensive analysis was conducted on how Kubernetes extension mechanisms, such as "Custom Resources" and the "Operator" pattern, could be used to create a framework enabling the modeling of closed control loops.

This thesis describes the developed platform, its architecture, the semantics of syntax in the defined objects, operational principles, integrations with external systems, and a user guide. It also discusses the platform's implementation, the technologies behind it, and the decisions made during its development. Finally, the thesis presents a practical test of the platform's functionality using the Open5GS 5G system emulator in combination with UERANSIM. The work concludes with a list of findings and potential extensions or improvements to the platform.

Keywords: Closed Control Loops, Kubernetes, Managing Telco-Networks, Automation, Go, Open5GS, Microservices

Spis treści

1. Wstęp	7
1.1. Przedmowa	7
1.2. Cel i zakres pracy	7
1.3. Struktura dokumentu	8
2. Stan wiedzy	8
2.1. Wstęp	8
2.2. Historia	8
2.3. ENI	10
2.4. Przegląd literatury	11
2.4.1. Wstęp	11
2.4.2. ONAP/CLAMP	15
2.4.3. Architektura referencyjna ETSI ZSM	16
2.4.4. Architektura referencyjna ETSI ENI	19
2.4.5. Architektura referencyjna CLADRA (TM Forum)	19
2.4.6. Podsumowanie	19
3. Architektura	19
3.1. Wstęp	19
3.2. Wymagania i założenia	19
3.3. Pojęcia i zasady	22
3.3.1. Wstęp	22
3.3.2. Problem zarządzania	22
3.3.3. System Sterowania	22
3.3.4. Pętla Sterowania	23
3.3.5. Zamknięta Pętla Sterowania	23
3.3.6. Agenci Translacyjni	24
3.3.7. Workflow pętli	24
3.3.8. Dane i Akcje	25
3.3.9. Open Policy Agent	26
3.4. Instrukcja Użycia	26
3.4.1. Instalacja Lupus	27
3.4.2. Implementacja Agentów Translacji	27
3.4.3. Planowanie Logiki Pętli	27
3.4.4. Przygotowanie Elementów Zewnętrznych	27
3.4.5. Wyrażenie workflow pętli	28
3.4.6. Aplikacja plików manifestacyjnych	28
4. Implementacja	28
4.1. Wstęp	28
4.2. Mechanizmy Kubernetes stojące za Lupus	28
4.2.1. Kontroler	28
4.2.2. Zasoby własne	29

4.2.3. Operatory	30
4.2.4. Kubebuilder	30
4.3. Decyzje podjęte podczas developmentu	30
4.3.1. Wstęp	30
4.3.2. Komunikacja pomiędzy Elementami Lupus	30
4.3.3. Dane	32
4.3.4. Polimorfizm w Go	33
4.3.5. Dwa rodzaje workflow	37
4.3.6. Funkcje użytkownika	37
5. Test platformy na emulatorze sieci 5G	40
5.1. Wstęp	40
5.2. Emulator sieci 5G	40
5.2.1. Open5GS	40
5.2.2. UERANSIM	40
5.2.3. Wdrożenie na Kubernetes	41
5.3. Wdrożenie Lupus	42
5.3.1. Problem Zarządzania	42
5.3.2. Implementacja Agentów Translacji	43
5.3.3. Planowanie Logiki Pętli	45
5.3.4. Przygotowanie Elementów Zewnętrznych	46
5.3.5. Wyrażenie workflow pętli w notacji LupN	46
5.3.6. Prezentacja działania jednej iteracji pętli	47
6. Podsumowanie	48
6.1. Wstęp	48
Bibliografia	51
Wykaz symboli i skrótów	52
Spis rysunków	54
Spis tabel	54
Spis załączników	54

1. Wstęp

1.1. Przedmowa

Wraz z rozwojem telekomunikacji stopień jej skomplikowania jak i mnogość podłączonych urządzeń stale rośnie. Sieci 5G zwiastują obsługę miliardów urządzeń, co sprawia, że tradycyjne podejście do zarządzania sieciami staje się niewystarczające. W pewnym momencie manualne operowanie sieciami (ang. *human-driven networks*) stanie się wręcz niemożliwe. Dlatego obserwujemy obecnie zwrot w stronę wirtualizacji oraz automatyzacji sieci. Jednocześnie dynamiczny rozwój sztucznej inteligencji otwiera nowe możliwości. Te dwa czynniki stanowią wspólnie świetny fundament do tego, aby branża sieci telekomunikacyjnych postawiła sobie za cel budowę "inteligentnych" sieci - takich, które są w pełni autonomiczne, samowystarczalne oraz niewymagają nadzoru ludzkiego.

W tym celu ETSI (European Telecommunications Standards Institute) powołało komitet o nazwie ENI - Experiential Networked Intelligence, który ma na celu wypracowanie specyfikacji dla Kognitywnych Systemów Zarządzania Siecią (Cognitive Network Management system). Kognitywny system oznacza taki, który jest w stanie uczyć się i podejmować decyzje bazujące na zebranej wiedzy w sposób przypominający ludzki umysł. ENI opiera swoją architekturę na zamkniętych pętlach sterowania.

Pętlą sterowania, ENI nazywa mechanizm, który monitoruje wydajność systemu lub procesu poddanego kontroli w celu osiągnięcia pożądanego zachowania. Innymi słowy, pętla sterowania reguluje działanie zarządzanego obiektu. Pętle sterowania można podzielić na zamknięte lub otwarte w zależności od tego czy działanie sterujące zależy od sprzężenia zwrotnego z kontrolowanym obiektem. Jeśli tak, pętle nazywamy zamkniętą, jeśli nie - otwartą.

W przypadku architektury ENI, zamknięta pętla sterowania służy jako model organizacji pracy (ang. *workflow*) elementów odpowiedzialnych za sztuczną inteligencję. W dokumencie [1], ENI dokonało przeglądu obiecujących architektur zamkniętych pętli sterowania znanych ludzkości. Naturalnym następnym krokiem jest zaproponowanie platformy, na której można takowe pętle zamodelować oraz uruchomić.

1.2. Cel i zakres pracy

Celem pracy jest kontynuacja badań ENI, polegająca na zaproponowaniu architektury platformy, na której możliwe będzie modelowanie oraz uruchamianie zamkniętych pętli sterowania. Niniejsza praca nie skupia się na aspektach związanych ze sztuczną inteligencją. Platforma ma jedynie służyć do zamodelowania oraz uruchomienia *workflow* pętli, ale sama nie stanowi środowiska wykonawczego dla jej komponentów.

W zakres pracy wchodzi: sformułowanie wymagań na platformę, opracowanie proponowanej architektury wsparte licznymi badaniami podczas jej powstawania, implementacja PoC (ang. *Proof of Concept*) według proponowanej architektury, przeprowadzenie testów platformy oraz analiza jej potencjału w kontekście dalszego rozwoju. Po publikacji praca

może stanowić podstawy do konkretnych implementacji Kognitywnych Systemów Zarządzania Siecią, specyfikowanych przez ENI.

1.3. Struktura dokumentu

Dokument został podzielony na 6 sekcji. Druga sekcja przedstawia bardziej szczegółowo niż we wstępie badania podjęte przez ENI. Stanowi ona nieco wprowadzenie teoretyczne oraz pojęciowe, aby ułatwić przekaz w dalszej części pracy. Sekcja trzecia zawiera opis proponowanej architektury. Sekcja czwarta opisuje implementację PoC platformy oraz napotkane wyzwania podczas formułowania architektury, które celowo zostały zebrane w jedno miejsce i umieszczone oddzielnie w celu łatwiejszej lektury. Sekcja numer pięć przedstawia użycie platformy w praktyce przy okazji stanowiąc jej test. Na koniec, w sekcji szóstej przeprowadzono analizę zaproponowanej architektury, jej możliwości oraz ograniczenia oraz wskazano kierunki potencjalnego rozwoju.

2. Stan wiedzy

2.1. Wstęp

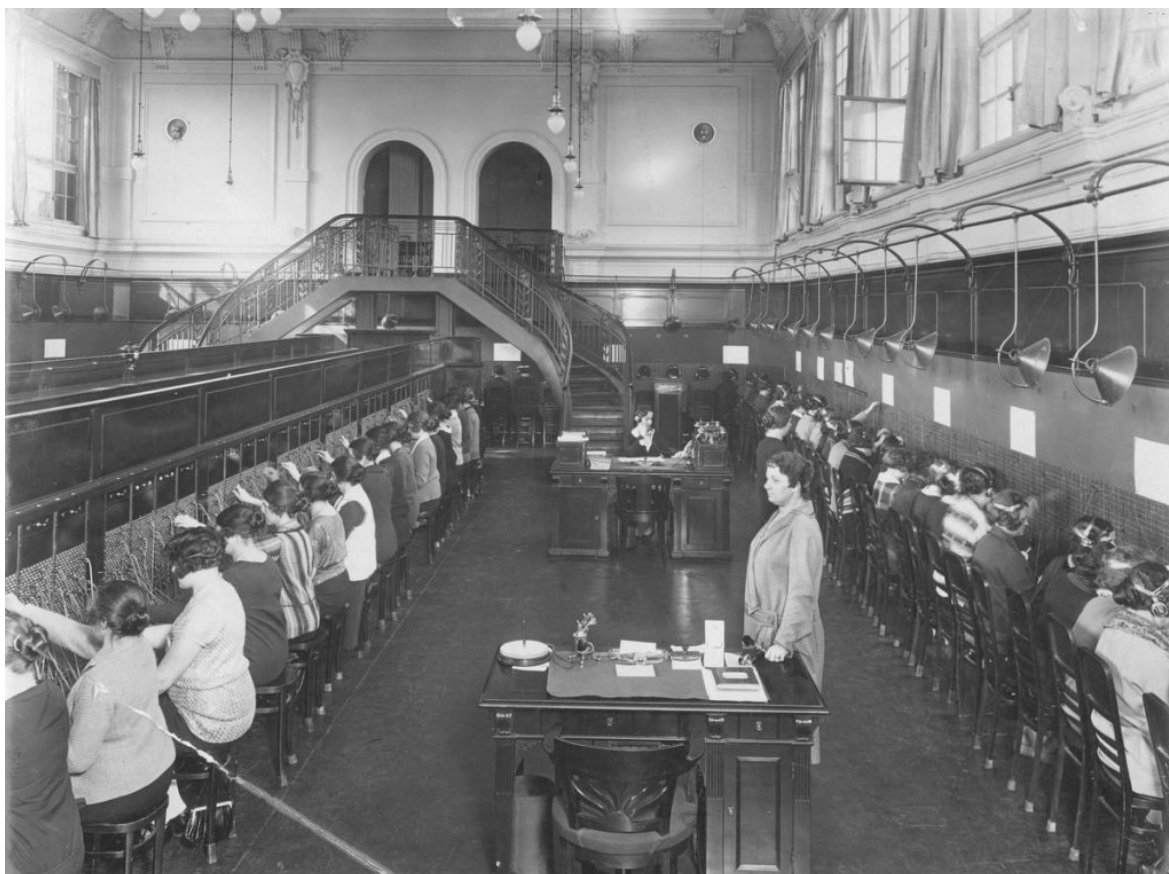
Sekcja ta stanowi krótki wstęp historyczny problemu jakiego podejmuje się praca oraz przytacza kluczowe znaleziska z powiązanej literatury w celu identyfikacji luki badawczej.

2.2. Historia

Na początku XX wieku, gdy sieci telekomunikacyjne dopiero się rozwijały, wszystkie połączenia zestawiane były ręcznie. W momencie, gdy abonent podnosił słuchawkę telefonu, jego aparat wysyłał sygnał do lokalnej centrali telefonicznej. Na tablicy świetlnej zapalała się lampka informująca telefonistkę o próbie połączenia. Telefonistka odbierała, pytając, z kim abonent chce się połączyć. Następnie wprowadzała odpowiednią wtyczkę do odpowiedniego gniazda na tablicy rozdzielczej, zestawiając fizyczne połączenie między dwoma liniami. Jeśli rozmowa miała się odbyć na większą odległość (np. między miastami) połączenie przekazywane było przez kolejne centrale. Każda centrala po drodze wymagała ręcznej obsługi przez pracujące w nich telefonistki.

Dziś taki scenariusz wydaje się wręcz absurdalny, a oferty pracy dla telefonistek dawno już zniknęły z tablic ogłoszeń. Praca wykonywana przez telefonistki (czyli komutacja łączy) nadal jest potrzebna do prawidłowego funkcjonowania sieci telekomunikacyjnych, lecz wykonywana jest przez programy komputerowe w sposób w pełni zautomatyzowany. Ręczna komutacja była pierwszym krokiem w kierunku rozwoju globalnych sieci telekomunikacyjnych i choć z dzisiejszej perspektywy wydaje się być bardzo pracochłonna i ograniczająca, bez niej nie powstałyby fundamenty, na których oparto późniejsze systemy automatyczne. Jest to przykład tego, jak technologia stopniowo uwalniała człowieka od bezpośredniej obsługi różnych systemów dając mu przestrzeń na rozwój w innych obszarach.

Na zasadzie indukcji możemy przyjąć, iż dziś znajdujemy się w podobnym położeniu - sieci telekomunikacyjne wciąż wymagają bezpośredniego zarządzania przez człowieka.



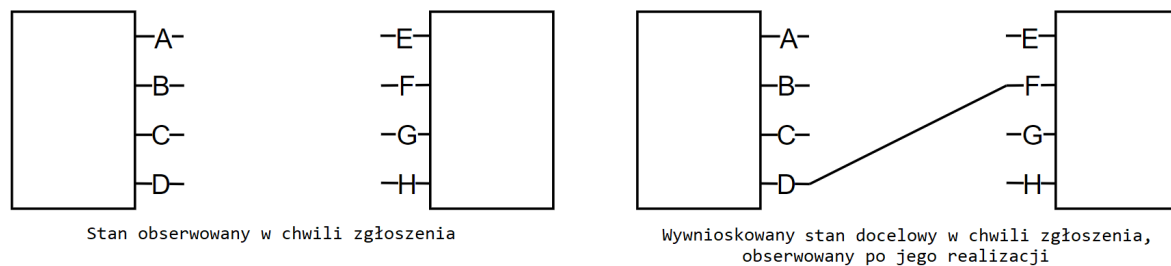
Rysunek 2.1. Pracowniczki warszawskiej centrali telefonicznej z końca lat 20. XX wieku

Współcześnie granica styku - system-człowiek - jest mocno przesunięta, a interakcja zachodzi na dużo wyższym poziomie abstrakcji – zamiast fizycznie łączyć linie, operatorzy zarządzają skomplikowanymi systemami sterowania ruchem telekomunikacyjnym.. Możliwe, że nie istnieje ostateczny punkt styku i systemy telekomunikacyjne zawsze będą wymagały nadzoru ludzkiego. Niezależnie od tej kwestii przesunięcie wspomnianej granicy w czasach telefonistek, a dziś wymaga automatyzacji innego rodzaju.

O telefonistkach możemy powiedzieć, że zarządzały one pracą systemu telekomunikacyjnego ¹ regulując jego działanie. W momencie zgłoszenia zastawały system w pewnym stanie i w ramach realizacji zgłoszenia musiały doprowadzić system do nowego (docelowego) stanu. Cała ich praca tak naprawdę polegała na "wnioskowaniu" (ang. *reason*) jak ma wyglądać stan docelowy, a następnie na wykonaniu akcji sterującej, która doprowadzała system do tego stanu.

Pomówmy trochę o *wnioskowaniu*. Jak ono przebiega? Po pierwsze, telefonistka otrzymuje *dane*. **Dane** to fakty lub statystyki zebrane w celu analizy. Telefonistka gromadzi dane w formie faktów dotyczących zaistniałych zdarzeń. Na przykład: dzwoni aparat telefoniczny, telefonistka podnosi słuchawkę i słyszy, że abonent chce połączyć się z abonentem mieszkającym w kamienicy pod numerem 35 na ulicy Polnej. Następnie dostępne dane zostają przekształcone w *informację*. **Informacja** to dane pozbawione formy przekazu. Równie

¹ t.j. siecią PSTN



Rysunek 2.2. Dwa stany systemu: sprzed, oraz po realizacji zgłoszenia o treści: "D" chce zadzwonić do "F"

dobrze rozmowa z abonentem mogłaby przebiec zupełnie inaczej, a telefonistka mogłaby otrzymać listowną prośbę o realizację połączenia. Różne dane mogą dostarczyć tę samą informację, czyli że abonent z określonego łącza chce dodzwonić się do abonenta z innego, konkretnego łącza. Gdy telefonistka uzyska informację o zgłoszeniu, łączy ją z posiadanymi już informacjami, np. dotyczącymi rozmieszczenia łączy na tablicy komutacyjnej. Następnie wykorzystuje swoją *wiedzę*, czyli zestaw wzorców umożliwiających interpretację oraz przewidywanie tego, co się wydarzyło, co dzieje się obecnie i co może się wydarzyć w przyszłości. **Wiedza** opiera się na informacji oraz umiejętnościach zdobytych dzięki *doświadczeniu* lub *edukacji*. Telefonistka obserwuje aktualny stan systemu i *wnioskując*, na podstawie wiedzy, wyobraża sobie stan docelowy (pożądany), a następnie definiuje akcje sterujące, które należy przeprowadzić na systemie, aby osiągnął on ten stan.

W przypadku komutacji proces wnioskowania jest stosunkowo prosty i możliwy do zapisania w języku programowania. Dlatego wraz z rozwojem technologicznym dokonano przejścia na automatyczne centrale telefoniczne. Jednak opisany powyżej schemat pozostaje aktualny. Do centrali trafiają dane w postaci sygnału na jednym z łączy. Sygnał zawiera numer MSISDN abonenta końcowego. Centrala przekształca te dane w informację, a następnie program w jej pamięci dokonuje wnioskowania. Wiedza potrzebna do wnioskowania zakodowana jest w programie. Dzięki programowalnym przełącznikom komutacyjnym również wykonanie akcji sterującej (czyli połączenie dwóch łączy) jest możliwe bez udziału człowieka.

W dzisiejszych czasach proces wnioskowania wykonywany przez ludzi nadzorujących systemy telekomunikacyjne nie jest już tak prosty. Dzięki wirtualizacji i programowalności funkcji sieciowych każdy rodzaj *wnioskowania*, który można sprowadzić do programu komputerowego, został już zautomatyzowany. Obecnie akcje wykonywane przez człowieka rzeczywiście wymagają jego inteligencji². Dlatego kolejnym etapem rozwoju telekomunikacji jest stworzenie inteligentnych sieci.

2.3. ENI

ENI rozwija specyfikacje w kierunku Kognitywnego Systemu Zarządzania, który będzie regulował działanie sieci poprzez wykorzystanie technik sztucznej inteligencji, takich jak uczenie maszynowe (ang. *machine learning*) oraz wnioskowanie (ang. *reasoning*).

² lub sprawczości w świecie fizycznym, której komputerom brakuje

Z poprzedniej sekcji wiemy, że aby *wnioskować* potrzebna jest *wiedza*. Skąd taki system pozyskiwałby wiedzę? Otóż, dzieje się to poprzez proces *Uczenia Maszynowego*. Pierwsza faza czyli tzw. "trening", odbywa się jeszcze przed wdrożeniem systemu. Ale system również może uczyć się "z doświadczenia" (ang. *experience*), kiedy już jest w pełni operacyjny. Stąd mówimy o empirycznej inteligencji sieciowej (ang. *experiential networked intelligence*).

Czyli z jednej strony *wiedza* potrzebna do wnioskowania, a w ostateczności podejmowania decyzji płynęłaby z samej sieci, co w przypadku telefonistki równoważne jest zapalającej się lampce na tablicy świetlnej. Z drugiej strony telefonistka ma również niejako wbudowaną w siebie wiedzę o tym, jak zorganizowane są łącza na przełącznicy komutacyjnej. Podobnie sieć musi mieć pewną części wiedzy nieco narzuconą z góry. Ta część wiedzy stanowi jako zestaw reguł używanych do zarządzania i kontrolowania stanu sieci. Takie reguły nazywamy *politykami*.

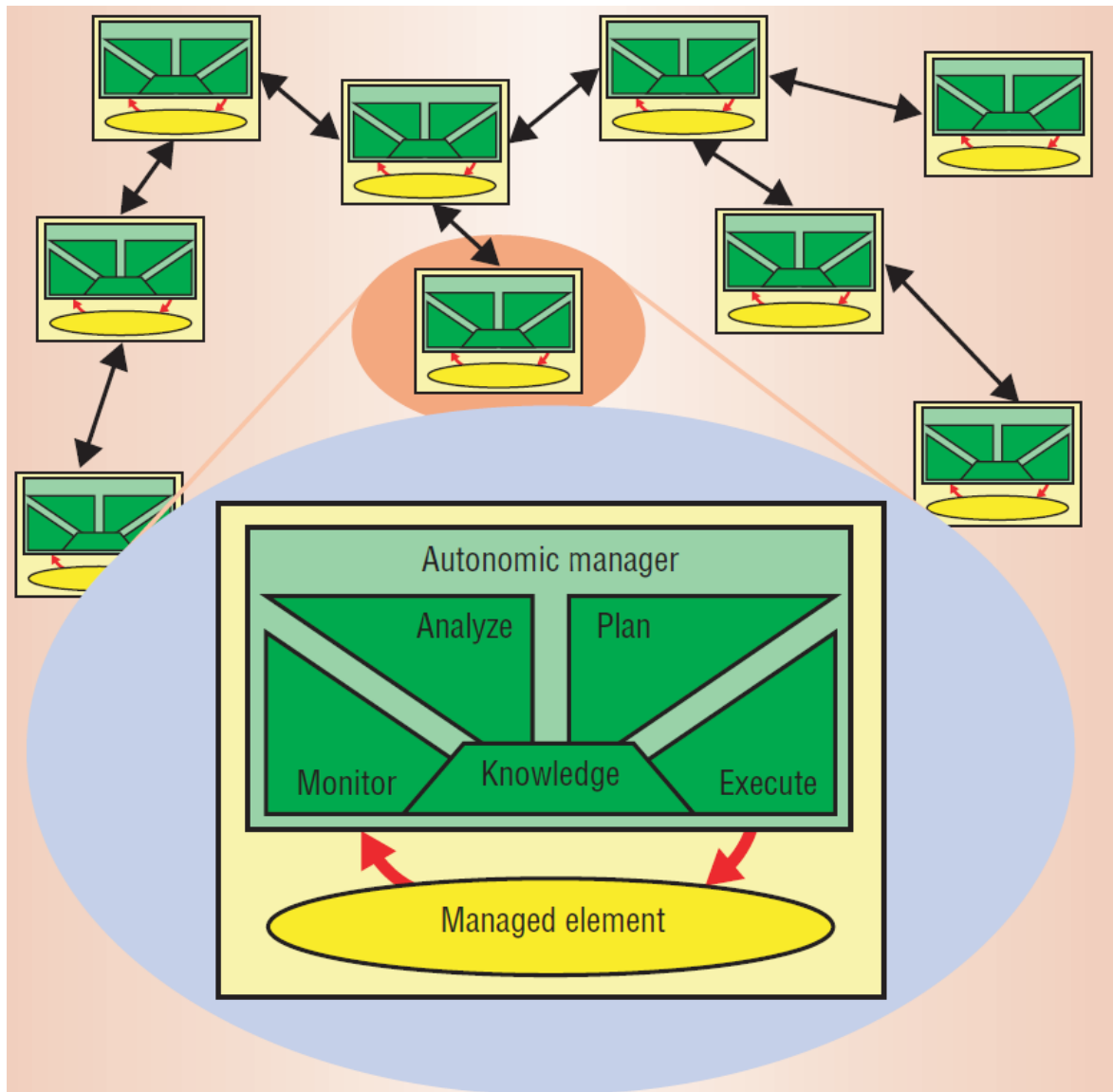
Polityki mogą płynąć od: aplikacji zarządzających siecią, użytkowników sieci, systemów OSS/BSS lub Orkiestratora. Ważne jest to, aby polityki były *świadome kontekstu*. Pozwoli to na stworzenie systemu *kognitywnego*, czyli takiego, który uczy się, wnioskuje oraz podejmuje decyzje w sposób przypominający ludzki umysł. Taki system z kolei jest w stanie w dużym stopniu odciążyć operatora i zautomatyzować zadania jak:

- dynamiczne przydzielanie zasobów (ang. *dynamic resource allocation*),
- równoważenie obciążenia (ang. *load balancing*),
- zarządzanie wydajnością energetyczną (ang. *energy efficiency management*,
- samo naprawiające się sieci (ang. *self-healing networks*),
- optymalizacja jakości doświadczeń użytkowników (ang. *QoE optimization*),
- egzekwowanie polityk (ang. *policy enforcement*),
- zapewnienie zgodności z regulacjami (ang. *regulatory compliance*)
- i wiele innych.

2.4. Przegląd literatury

2.4.1. Wstęp

Rozważania na temat inteligentnych systemów w kontekście niniejszej pracy należy zacząć od artykułu opublikowanego przez IBM w 2003 roku [2]. Wskazuje on, iż ówczesne systemy informatyczne stają się coraz bardziej skomplikowane, co prowadzi do kryzysu zarządzania. Ręczna administracja przestaje być skalowalna, ponieważ systemy wymagają coraz większej liczby specjalistów do instalacji, konfiguracji oraz optymalizacji. IBM wskazuje, że jedyną opcją poradzenia sobie z tym problemem jest Autonomiczne Przetwarzanie (ang. *Autonomic Computing*). Nawiązuje ono do autonomicznego układu nerwowego człowieka, który zarządza pracą naszego serca lub utrzymaniem temperatury ciała, zwalniając świadomą część mózgu z tych obowiązków. W kontekście systemów informatycznych człowiek zwolniony by był z konfiguracji, optymalizacji, naprawy oraz ochrony, gdyż powstałe systemy byłyby "Self-CHOP" (ang. *Self-Configuration, Self-Optimization, Self-Healing, Self-Protection*). Jest to ważne pojęcie stanowiące podwaliny wielu dalszych badań. Architektura autonomicznych systemów przedstawiona jest na rysunku ??.



Rysunek 2.3. Architektura autonomicznych systemów IBM

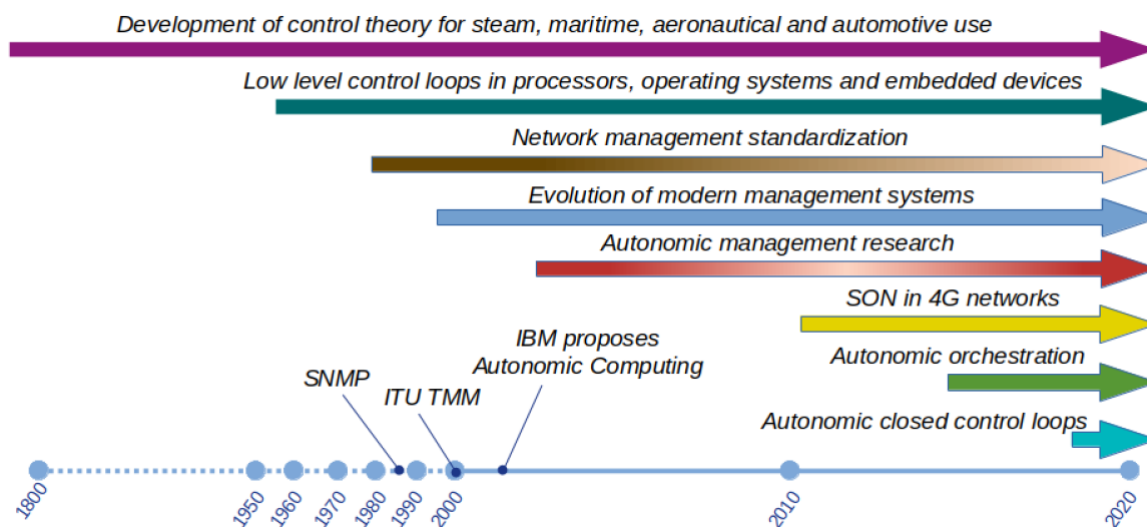
Podstawową jej jednostką jest element autonomiczny (ang. *Autonomic Element*). Składa się on z zarządzanego komponentu (np. serwera, bazy danych, aplikacji czy urządzenia sieciowego) oraz autonomicznego menadżera (ang. *Autonomic Manager*), który steruje jego działaniem. Menedżer autonomiczny działa w cyklu MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge). Faza "Monitor" zbiera dane o zarządzanym obiekcie, które to są analizowane i oceniane pod względem potencjalnych problemów w fazie "Analyze". "Plan" odpowiada za rozpisanie działań naprawczych lub optymalizacyjnych, które są przekazywane do "Execute" w celu wdrożenia w systemie zarządzanym. "Knowledge" nie jest fazą a repozytorium wiedzy, do której każdy z elementów cyklu ma dostęp. Tu gromadzona jest *wiedza*. Elementy autonomiczne mogą ze sobą współpracować i wymieniać się wiedzą. Inteligencja społeczna (ang. *social intelligence*) całego rozproszonego systemu rośnie wraz z liczbą interakcji pomiędzy elementami, podobnie jak w kolonii mrówek. Gdy przyjdzie nowy element, to zdobywa wiedzę od reszty populacji. Widzimy tu lekką inspirację [3].

Następnie [4] opisuje ewolucję architektury sieci telekomunikacyjnych w kierunku większej automatyzacji wymieniając czynniki umożliwiające (ang. *enablers*) taki kierunek jako technologie: SDN (ang. *Software Defined Networks*) oraz NFV (ang. *Network Function Virtualization*).

Należy w tym momencie rozróżnić pojęcia Automatyzacji i Autonomiczności. Pierwsze odnosi się do predefiniowanego i zaprogramowanego procesu, podczas gdy drugie do aspektów związanych z samozarządzaniem. Zazwyczaj proces automatyczny nie jest w stanie zaadaptować się do zmian środowiska bez interwencji człowieka, co z kolei potrafi proces autonomiczny [5]. Czynnikiem technologicznym (ang. *enabler*) pozwalającym na przejście z automatyzacji na autonomiczność jest sztuczna inteligencja. [6] dokonuje przeglądu kierunków rozwoju badań na temat systemów ZSM. Czyli bezobsługowych systemów zarządzania siecią i usługami (ang. *Zero-touch network and Service Management*), gdzie sztuczna inteligencja odgrywa kluczową rolę. Artykuł wymienia projekty organizacji standaryzujących w tym kierunku:

- **ETSI GS ZSM** - specyfikuje referencyjną architekturę zarządzania siecią i usługami end-to-end. Framework ZSM jest postrzegany jako system zarządzania nowej generacji, który ma na celu pełną autonomiczność wszystkich procesów - od planowania, projektowania, dostarczania i wdrażania, po udostępnianie, monitorowanie i optymalizację. Docelowo, w idealnym przypadku, powinien on działać w 100% samodzielnie bez ingerencji człowieka.
- **ETSI GR ENI** - specyfikuje referencyjną architekturę kognitywnych systemów zarządzania siecią używając zamkniętych pętli sterowania oraz świadomych kontekstu polityk. O ile ETSI GS ZSM skupia się na technikach automatyzacji zarządzania oraz udostępniania usług end-to-end, to ENI koncentruje się na technikach AI, zarządzaniu poprzez polityki oraz zamkniętych pętlach sterowania.
- **TM FORUM** - specyfikuje referencyjną architekturę CLADRA (Closed Loop Anomaly Detection and Resolution Automation) opartej o AI, która umożliwia dostawcom usług komunikacyjnych (CSP - ang. *Communication Service Providers*) na szybkie wykrywanie i rozwiązywanie problemów sieciowych.

Powyższe architektury zostaną dokładniej opisane w dalszej części pracy (zwłaszcza w części badawczej). Istotne na ten moment jest to, że każda z tych architektur opiera swoje działanie o zamknięte pętle sterowania, co przewidział z resztą artykuł [7]. Wskazuje on, iż badanie oraz stosowanie zamkniętych pętli sterowania jest dosyć starą dyscypliną sięgającą aż od ery pary³ (Rysunek 2.4). Stosowane są one z powodzeniem w systemach morskich, lotniczych, motoryzacyjnych, przemysłowych ale też mikroprocesorach, systemach wbudowanych czy sterownikach urządzeń. Mimo że tego typu systemy są często złożone, cechują się determinizmem i dobrze określonymi granicami, co sprawia, że świetnie nadają się do zastosowania teorii sterowania⁴. Systemy telekomunikacyjne natomiast są słabo określonymi, stochastycznymi systemami, co sprawia, że zastosowanie teorii sterowania do ich zarządzania okazuje się wyjątkowo trudne. Stąd duże opóźnienie w ich adaptacji w branży telekomunikacyjnej. Od czasu pojawienia się propozycji Autonomicznego Zarządzania [2] w 00s oraz enablerów w postaci SDN&NFV w 10s zaczęły się pojawiać implementacje zamkniętych pętli sterowania w telekomunikacji. Niestety są one bardzo pragramtyczne i sztywne, skupione jedynie na dowieżeniu danej funkcjonalności. Dodatkową problematyczność zagadnienia sprawia konieczność integracji systemów od wielu różnorodnych dostawców.



Rysunek 2.4. Zestawienie lini czasu rozwoju systemów zarządzania siecią oraz pętli sterowania

[7] wskazuje wyzwania jakim należy się przeciwstawić, aby umożliwić autonomiczność poprzez pętle sterowania w sieciach. Są to następujące kroki:

- Potrzebny jest model opisujący pętle sterowania w standardowy sposób. Pozwoli to uchronić się przed pragramtycznością i niesystematycznością.
- Potrzebny jest framework do wdrażania pętli sterowania umożliwiający ich zarządzanie oraz działanie (ang. *runtime*) w jednym miejscu.

³ https://en.wikipedia.org/wiki/Age_of_Steam

⁴ https://en.wikipedia.org/wiki/Control_theory

- Potrzebne są (najlepiej) graficzne narzędzia do projektowania i symulacji pętli sterowania, tak aby ich wdrażanie nie wymagało specjalistycznych umiejętności technicznych.

2.4.2. ONAP/CLAMP

Następnym punktem [7] jest omówienie systemu zarządzania ONAP⁵ (Open Network Automation Platform) [8]. ONAP to open-source'owa platforma do orkiestracji, zarządzania i automatyzacji usług sieciowych, rozwijana przez Linux Foundation. ONAP składa się z **wielu modułów i API**, które umożliwiają pełne **zarządzanie cyklem życia usług sieciowych**, od ich projektowania, poprzez wdrażanie, aż po operacje eksploatacyjne. Jego kluczowe elementy to:

- **Master Service Orchestrator (MSO)** – centralny komponent odpowiedzialny za orkiestrację.
- **SDN Controllers (APPC, SDNC, VFC)** – kontrolery zarządzające różnymi warstwami sieci.
- **A&AI (Active and Available Inventory)** – komponent do zarządzania inwentaryzacją zasobów sieciowych.
- **DCAE (Data Collection, Analytics and Events)** – moduł do analizy danych i obsługi pętli sterowania.
- **CDS (Controller Design Studio)** – moduł do modelowania inteligentnych konfiguracji komponentów sieciowych
- **CLAMP (Control Loop Automation Management Platform)** – zarządza zamkniętymi pętlami sterowania.

Komponentem odpowiedzialnym za pętle sterowania w ONAP jak i zarówno kandydatem na przeciwstawienie wyzwaniom postawionym powyżej jest CLAMP⁶. Ma on jednak swoje ograniczenia:

- Wymaga stosowania architektury pętli zgodnej z MAPE-K, co ogranicza możliwość wdrożenia nietypowych pętli sterowania.
- Wymaga ścisłej integracji z innymi modułami ONAP takimi jak DCAE, CDS, czy silniki polityk, co ogranicza możliwości tworzenia pętli niezależnych od ekosystemu ONAP.
- Jest zoptymalizowany do prostych pętli sterowania. Brak natywnego wsparcia dla bardziej złożonych architektur jak pętle hierarchiczne, rozproszone lub współzależne.
- Brak wsparcia AI/ML w swoich komponentów.
- Ścisłe połączenie z wybranymi silnikami polityk takimi jak XACML⁷, Drools⁸ czy APEX⁹
- Ścisła specjalizacja w orkiestracji sieci SDN&NFV. Ciężko jest rozwijać pętle, które nie są związane z tym zagadnieniem.

⁵ <https://www.onap.org>

⁶ <https://docs.onap.org/projects/onap-policy-parent/en/istanbul/clamp/clamp/clamp-architecture.html>

⁷ <https://docs.onap.org/projects/onap-policy-parent/en/istanbul/xacml/xacml.html>

⁸ <https://docs.onap.org/projects/onap-policy-parent/en/istanbul/drools/drools.html>

⁹ <https://docs.onap.org/projects/onap-policy-parent/en/istanbul/apex/apex.html>

2.4.3. Architektura referencyjna ETSI ZSM

Nadrzędnym celem projektowym ZSM jest umożliwienie nie wymagającego udziału człowieka (ang. *zero touch*) zarządzania siecią i usługami w środowisku wielu dostawców (ang. *multi-vendor environment*) [9]. Architektura przyjmuje pewien zestaw pryncypiów (zasad):

1. Modularność
2. Rozszerzalność
3. Skalowalność
4. Sterowane modelem, otwarte interfejsy
5. **Automatyzacja zarządzania w pętli zamkniętej**
6. Obsługa funkcji zarządzania bezstanowego
7. Odporność
8. Rozdzielenie obszarów zarządzania
9. Kompozycyjność usług
10. Interfejsy oparte na intencjach
11. Abstrakcja funkcjonalna
12. Prostota
13. Zaprojektowane do automatyzacji

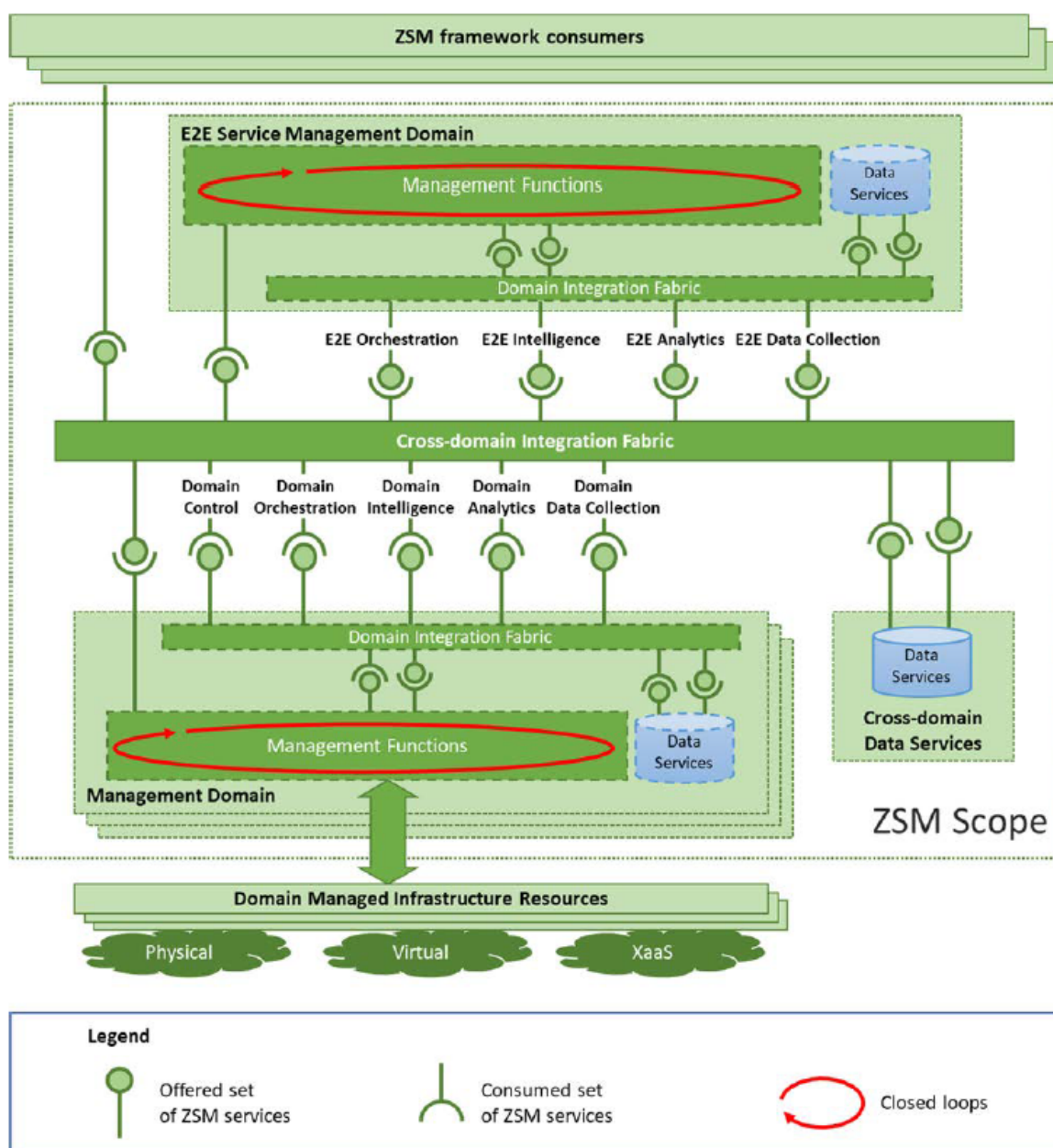
Framework ZSM podąża za trendem w branży polegającym na odejściu od monolitycznych ściśle powiązanych systemów na rzecz bardziej elastycznych zestawów usług zarządzania (ang. *management services*). Referencyjna architektura ZSM określa zestaw elementów składowych (ang. *building blocks*), które wspólnie umożliwiają konstruowanie bardziej złożonych usług i funkcji zarządzania. Konstrukcja ta odbywa się zgodnie ze wzorcami kompozycji i współdziałania (ang. *composition and interoperation patterns*).

Logicznie, framework ZSM składa się z rozproszonych serwisów zarządzania (ang. *management services*) i serwisów danych (ang. *data services*), które zorganizowane są w domeny zarządzania (ang. *management domains*) oraz zintegrowane poprzez powłokę integracyjną (ang. *integration fabric*). Powłoka integracyjna również jest używana do komunikacji serwisów zarządzania z systemami zewnętrznymi.

Architektura referencyjna zapewnia środki do budowania i komponowania luźno powiązanych funkcji zarządzania (ang. *management functions*), które oferują serwisy zarządzania i wspólnie zapewniają kompleksowe zarządzanie danej domeny w sposób "zero-touch". Aby oferować swoje serwisy, funkcje udostępniają możliwość ich wywoływania i komunikacji z nimi za pomocą end-point'ów.

Na rysunku 2.5 przedstawiono architekturę referencyjną. Poniżej krótkie omówienie jej elementów:

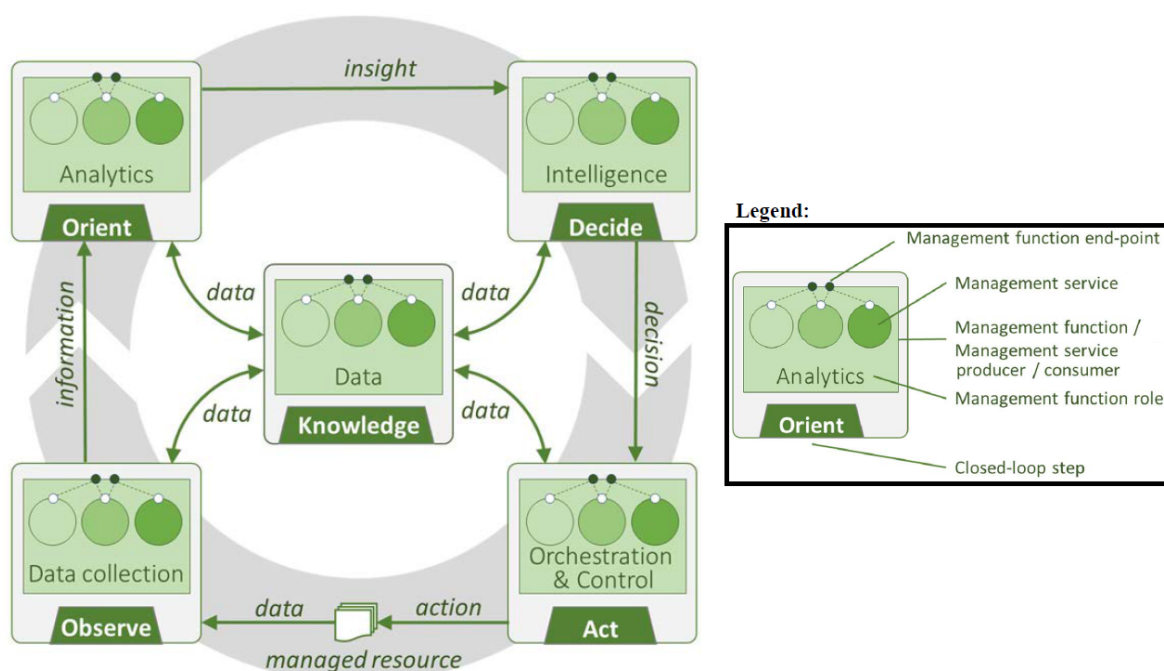
- Serwis zarządzania (ang. *Management Service*) - niewidoczna na rysunku, najmniej jedna jednostka usługowa architektury, odpowiedzialna za niepodzielny aspekt składowy zarządzania
- Serwis danych (ang. *Data Service*) - jednostka odpowiedzialna za gromadzenie danych, informacji oraz wiedzy



Rysunek 2.5. Architektura referencyjna ZSM

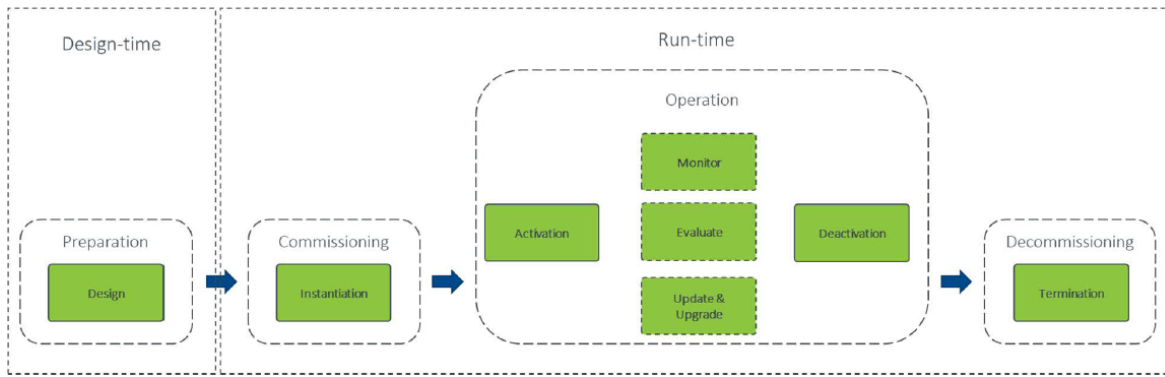
- Funkcja zarządzania (ang. Management Function) - połączenie wielu serwisów zarządzania, odpowiedzialna za dany aspekt składowy zamkniętej pętli zarządzania
- Domena Zarządzania (ang. Management Domain) - odpowiedzialna za konkretną funkcjonalność zarządzania (np. analityka, orkiestracja, bezpieczeństwo)
- Domena Zarządzania end-to-end (ang. E2E Management Domain) - odpowiedzialna za zarządzanie usługami wymagającymi wielu funkcjonalności
- Powłoka integracji (ang. Integration Fabric) - odpowiedzialna za komunikacje pomiędzy wszystkimi komponentami

Pętle zarządzania komponowane są z funkcji zarządzania w sposób pokazany na rysunku 2.6. Jako model przyjęto strukturę pętli OODA [10] z wprowadzeniem komponentu wiedzy znanego z pętli MAPE-K [2].



Rysunek 2.6. Mapowanie pomiędzy elementami składowymi architektury ZSM a zamkniętą pętlą sterowania

Architektura ZSM również specyfikuje cykl życia zamkniętej pętli sterowania oraz koordynację między pętlami. Cykl życia pętli podzielony jest na dwie fazy: "Design-Time" oraz "Run-Time" a ich poszczególne kroki obserwujemy na rysunku 2.7. Jeśli chodzi o koordynację między pętlami to zdefiniowano dwa przypadki: pętles hierarchiczne oraz pętles peer-to-peer. Pętles mogą komunikować się w obrębie jednej domeny zarządzania jak i również między domenami.



Rysunek 2.7. Fazy cyklu życia oraz aktywności zamkniętej pętli sterowania

2.4.4. Architektura referencyjna ETSI ENI

2.4.5. Architektura referencyjna CLADRA (TM Forum)

2.4.6. Podsumowanie

Niniejsza praca znajduje lukę, którą chce zapłacić w postaci platformy służącej do modelowania oraz uruchamiania zamkniętych pętli sterowania. Luka ta umotywowana jest wyzwaniem zdefiniowanymi przez [7], brakami w ONAP/CLAMP oraz zidentyfikowanie potrzeby takiej platformy w architekturach referencyjnych definiowanych przez ETSI GS ZSM oraz CLADRA, które to zostaną omówione dokładniej w dalszej części pracy w celu sformułowania wymagań na platformę.

Odnosząc się do architektury z [2] proponowana w niniejszej pracy platforma polega na wyniesieniu autonomicznych menadżerów do jednego wspólnego miejsca (Rysunek 2.8).

3. Architektura

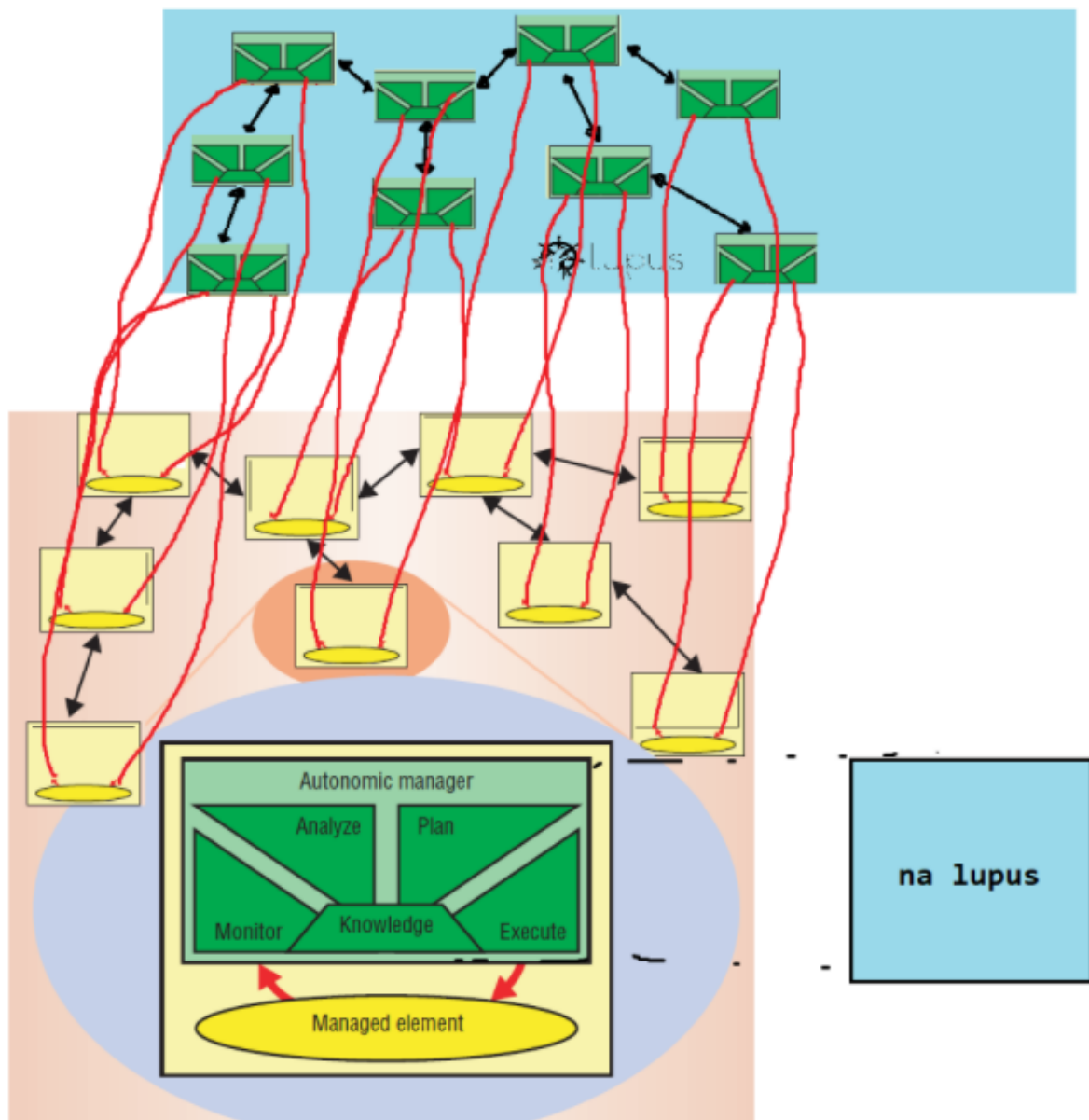
3.1. Wstęp

Niniejsza sekcja przedstawia zaproponowaną w pracy architekturę *Systemu ENI*, t.j. platformy, na której jest możliwe modelowanie oraz uruchamianie workflow zamkniętych pętli sterowania. Platformie nadano nazwę w celu ułatwienia jej opisu. Nazwa brzmi "Lupus". Powstała od przekształcenia angielskiego słowa "loops" oznaczającego pętle, oraz od zakotwiczenia o wyraz mający znaczenie nadające się na "maskotkę" projektu. "Lupus" po łacinie oznacza wilka, a po angielsku "toczeń rumieniowaty", stąd w logo projektu zarumieniony wilk.

Architektura w rozumieniu ENI jest to "zbiór reguł i metod opisujących funkcjonalność, organizację oraz implementację systemu". Niniejsza sekcja pomija aspekt implementacji, która jest omówiona w następnej sekcji.

3.2. Wymagania i założenia

Po pierwsze Lupus ma przyjąć rolę *regulatora* (ang. *Control System*) znanego z teorii sterowania. Regulowanymi systemami mają być w tym przypadku systemy telekomuni-



Rysunek 2.8. Koncepcja wyniesienia autonomicznych menadżerów na wspólną platformę (//TODO)

kacyjne. Aby odegrać rolę *Systemu ENI*, platforma musi być w stanie zamodelować oraz uruchomić dowolną zamkniętą pętlę sterowania, zwłaszcza te zawarte w [1].

Z racji ogólności systemu oraz zwiększenia szansy na pozytywne przyjęcie w społeczności wybrano Kubernetes jako infrastrukturę dla platformy. Kubernetes natywnie używa zamkniętych pętli sterowania w swojej *warstwie sterowania* (ang. *control plane*), z których użytkownik jest w stanie skorzystać za pomocą mechanizmów rozszerzeń Kubernetes takich jak *Definicje Zasobów Własnych* (ang. *Custom Resource Definitions (CRD)*) oraz *Wzorzec Operatora* (ang. *Operator Pattern*). Są to mechanizmy dobrze znane w branży.

Wybór Kubernetes narzuca jednakże pewne ograniczenie. Regulowane procesy nie mogą odbywać się w czasie rzeczywistym. Warstwa sterowania Kubernetes działa nieco wolniej.

Lupus musi być "data-driven" co na polski można przetłumaczyć jako "sterowany danymi". Oznacza to, że platforma nie może narzucać żadnej postaci logiki pętli. Warstwa sterowania Kubernetes musi być w stanie interpretować zamiary użytkownika platformy, który może wyrazić dowolną pętlę. Zamiary te wyrażone są w danych.

Logikę pętli możemy podzielić na dwie części: workflow pętli oraz części obliczeniową. Workflow jest to zdefiniowane elementów oraz relacji między nimi. Częścią obliczeniową za to nazywamy procesowanie wykonywane przez konkretne elementy. Z racji podejścia "data-driven" nie możemy zaszyć części obliczeniowej w warstwie sterowania Kubernetes. Dlatego elementy odpowiedzialne za część obliczeniową są "na zewnątrz" pętli Lupus, przykładowo są to serwery HTTP, do których Lupus wykonując pętlę może się odwołać.

Z tego opisu powstaje garść wymagań oraz założeń, które nie są zbiorami rozłącznymi i wielu miejscach się zacierają, jednakże warto wyszczególnić je w sposób wylistowany poniżej, aby móc łatwiej się do nich odwoływać w dalszej części pracy:

1. Lupus jest skierowany do branży telekomunikacyjnej.
2. Lupus umożliwia modelowanie i uruchamianie dowolnych architektur zamkniętych pętli sterowania, w szczególności tych zaproponowanych w [1].
3. Lupus zarządza procesami, które nie wymagają regulacji w czasie rzeczywistym (są "non-realtime").
4. Lupus jest zaimplementowany na bazie Kubernetes, wykorzystując jego *Controller Pattern*.
5. Lupus jest oparty na danych (*data-driven*), co oznacza, że nie narzuca i nie ma wbudowanej żadnej domyślnej logiki pętli.
6. Faktyczne komponenty przetwarzające w pętli (część obliczeniowa) Lupus są zewnętrzne względem niego (np. serwery HTTP, szczególnie Open Policy Agent).
7. Lupus powinien być w stanie regulować pracą dowolnego systemu teleinformatycznego bez żadnych jego modyfikacji.
8. Zamodelowanie pętli oraz wyrażenie jej **Workflow** w Lupus nie powinno wymagać umiejętności technicznych.

Powyższa lista nazwana jest Wymaganiami i referowana w dalszej części dokumentu.

3.3. Pojęcia i zasady

3.3.1. Wstęp

Ta podsekcja dokumentuje **Lupus** w formie wyjaśniania zdefiniowanych na jego potrzeby pojęć, konceptów i zasad. Są one punktem wyjścia do dokładniejszych specyfikacji i zarazem zrozumienia architektury systemu. Sekcja ta zawiera wiele odnośników do definicji znajdujących się w załącznikach i nie wszystkie pojęcia tłumaczone są w tej sekcji.

3.3.2. Problem zarządzania

W rzeczywistym świecie często napotkamy sytuacje, w których byłoby dobrze, gdyby praca jakiegoś systemu mogła być stale regulowana. Na przykład:

- chcielibyśmy, aby samochody miały funkcję regulującą pracę silnika w celu utrzymania stałej prędkości,
- przydałoby się, gdyby lodówka mogła utrzymywać chłodną, ale nie ujemną temperaturę, niezależnie od tego, jak często otwierane są drzwi lub jaka jest temperatura na zewnątrz,
- byłoby korzystne, gdyby serwer w chmurze mógł zagwarantować, że aplikacja z wystarczającymi zasobami do pokrycia potrzeb użytkowników będzie uruchomiona i działała.

Problemy wymienione powyżej można traktować jako **problemy zarządzania** (ang. *management problems*). Nazwa bierze się stąd, że nie ma żadnych technicznych ograniczeń uniemożliwiających osiągnięcie tych celów. Wszystkie wymienione systemy są odpowiednio wyposażone; np. możemy dodać więcej paliwa do silnika lub dostarczyć więcej mocy do sprężarki w lodówce. Problem leży w faktycznym wykonaniu tych czynności w odpowiednich momentach np. dodanie więcej paliwa, gdy auto zwalnia lub dostarczenie większej mocy, gdy temperatura w lodówce wzrasta. Dlatego jest to problem czystego zarządzania.

Systemem, którym chcemy zarządzać, nazywamy **System Zarządzany** (ang. *managed system*).

3.3.3. System Sterowania

System Sterowania (ang. *Control System*¹⁰) to system, który reguluje pracę **Systemu Zarządzanego**. Przykładowo jest to:

- tempomat, który reguluje pracę silnika w celu utrzymania stałej prędkości,
- lodówka, która reguluje pracę sprężarki w celu utrzymania stałej, chłodnej temperatury,
- Kubernetes, który reguluje liczbę działających Podów, aby utrzymać pożądaną dostępność aplikacji.

Innymi słowy mówiąc, **System Sterowania** rozwiązuje **Problem Zarządzania**.

¹⁰innym tłumaczeniem na polski jest "regulator"

3.3.4. Pętla Sterowania

Ogólną architekturą **Systemów Sterowania** używaną do rozwiązywania **Problemów Zarządzania** jest **Pętla Sterowania**.

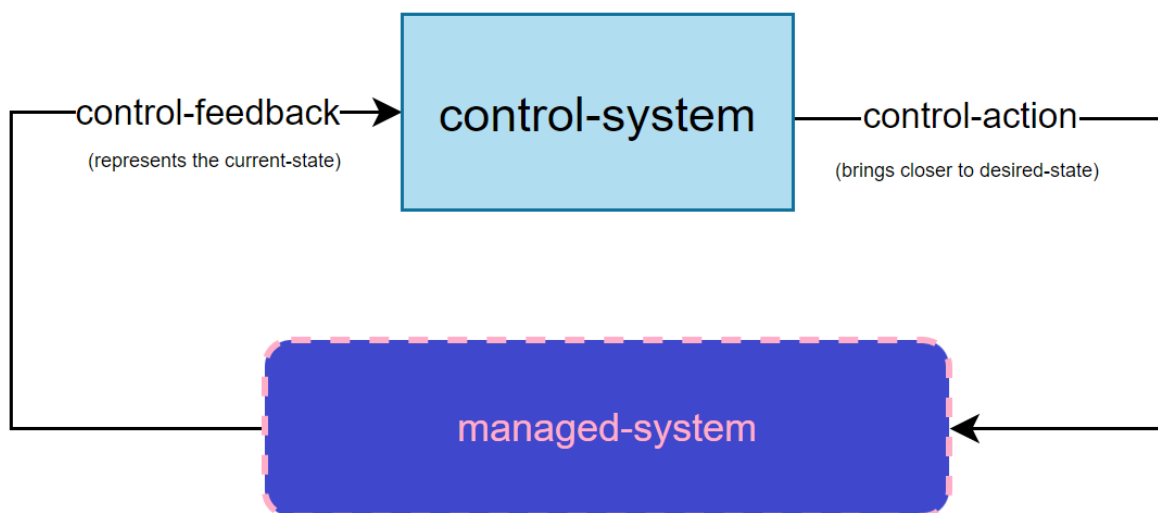
Pętle sterowania są klasyfikowane w zależności od tego, czy wykorzystują mechanizmy sprzężenia zwrotnego (ang. *feedback mechanism*):

- **Otwarte Pętla Sterowania: Akcja Sterująca** (ang. *Control Action*) (czyli wejście do **Systemu Zarządzanego**) jest niezależne od wyjścia **Systemu Zarządzanego**.
- **Zamknięte Pętla Sterowania:** Wyjście **Systemu Zarządzanego** jest sprzężane do wejścia **Systemu Sterowania** i wpływa na **Akcję Sterującą**.

W **Lupus** bierzemy pod uwagę wyłącznie **Zamknięte Pętle Sterowania**.

3.3.5. Zamknięta Pętla Sterowania

Jest to punkt wyjściowy dla naszej architektury referencyjnej (rys. 3.1).



Rysunek 3.1. Architektura referencyjna dla Lupus

Architekturę (rys. 3.1) należy czytać, mając na uwadze następujące definicje:

- **System Sterowania** (ang. *Control System*) - system, który rozwiązuje **Problem Zarządzania** występujący w **Systemie Zarządzanym** za pomocą **Zamkniętej Pętli Sterowania**. W każdej iteracji **System Sterowania** analizuje **Sprzężenie Zwrotne** (ang. *Control Feedback*) i wnioskuje **Akcję Sterującą**.
- **Zamknięta Pętla Sterowania** - nieskończona pętla, która reguluje stan **Systemu Zarządzanego**, iteracyjnie zbliżając jego **Stan Aktualny** do **Stanu Pożądanego**.
- **Akcja Sterująca** (ang. *Control Action*) - akcja wykonywana na **Systemie Zarządzanym**, która ma na celu przybliżenie go do **Stanu Pożądanego**.
- **Sprzężenie Zwrotne** (ang. *Control Feedback*) - reprezentacja **Stanu Aktualnego** wysyłana z (odbierana od) **Systemu Zarządzanego**.

W powyższej architekturze **Lupus** pełni rolę **Systemu Sterowania**.

3.3.6. Agenci Translacyjni

Z racji, że każdy system, bez żadnych modyfikacji (Wymaganie 7) może wejść w rolę **Systemu Zarządzanego**, potrzebujemy warstwy integracji między **Systemem Zarządzanym** a **Lupus**, podobnej do koncepcji "API Broker" w architekturze ENI (rys. ??). Tak narodziła się koncepcja **Agentów Translacji** (ang. *Translation Agents*).

W każdym **wdrożeniu Lupus**, **Użytkownik** musi stworzyć **Agentów Translacyjnych**.

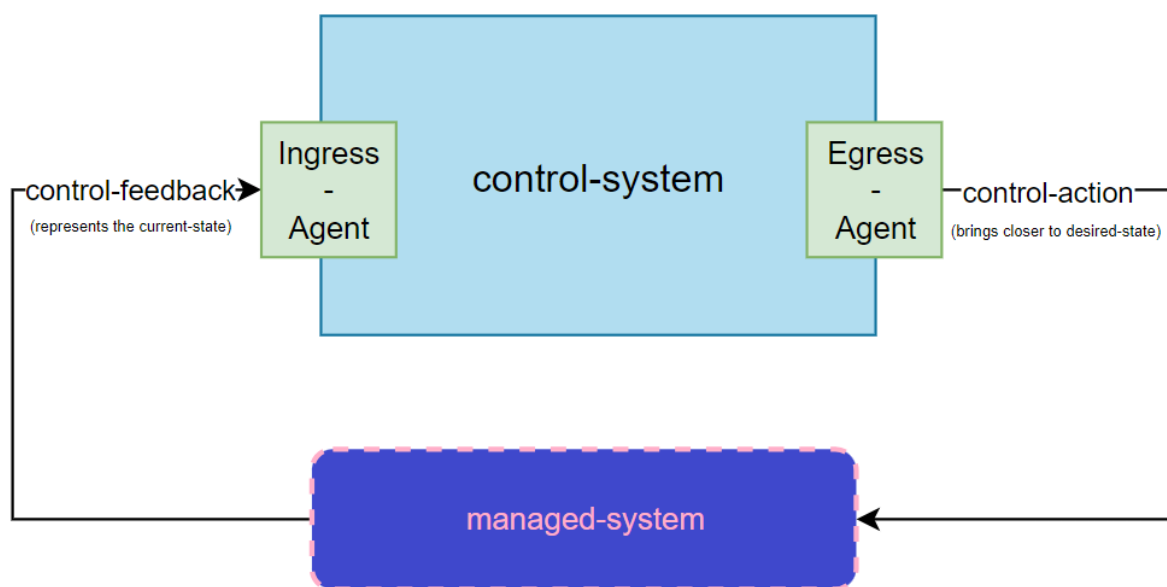
Z punktu widzenia komunikacji, każdego **Agenta Translacyjnego** możemy podzielić na dwie części:

- Część komunikująca się z **Systemem Zarządzanym**. Jest zewnętrzna względem **Lupus** i nie podlega żadnej specyfikacji.
- Część komunikująca się z **Lupus**. Musi być zgodna z jednym z **Interfejsów Lupus**: **Lupin** lub **Lupout**.

Mamy dwóch **Agentów Translacji**, jednego do komunikacji przychodzącej (ang. *Ingress*) i jednego do wychodzącej (ang. *Egress*).

W każdej iteracji pętli zadaniem **Agentu Ingress** jest odbieranie/zbieranie **Sprzężenia Zwrotnego** z **Systemu Zarządzanego** i translacja go na format zrozumiały przez **Lupus** za pomocą **Interfejsu Lupin**. Z kolei zadaniem **Agentu Egress** jest odbieranie **Finalnych Danych** i translacja ich na **Akcję Sterującą**, która później zostaje wysłana do (lub przeprowadzona na) **Systemie Zarządzanym**.

Architektura wraz z wprowadzeniem **Agentów Translacji** ukazana jest na rysunku 3.2.



Rysunek 3.2. Architektura referencyjna z agentami translacji

Specyfikacja interfejsów **Lupin** oraz **Lupout** zawarta jest w Załączniku 4.

3.3.7. Workflow pętli

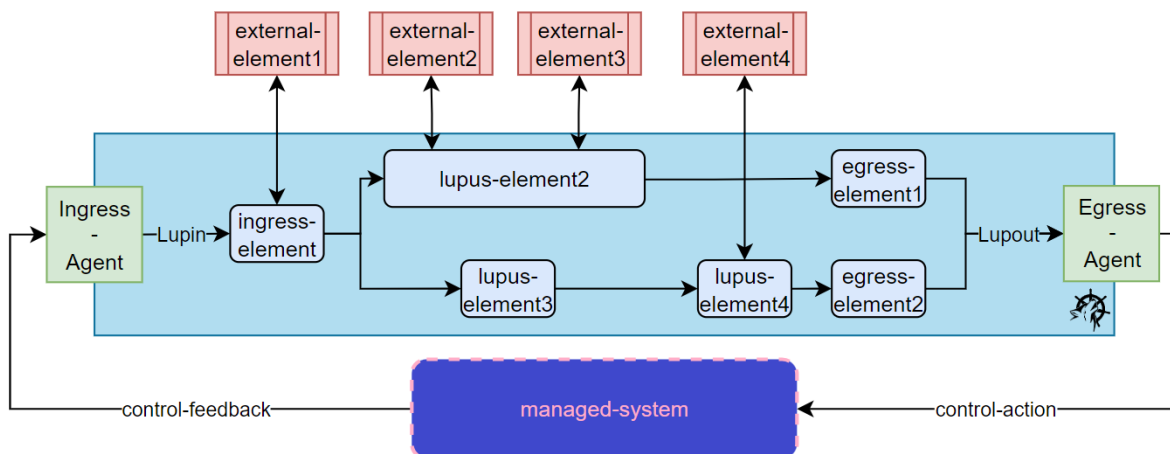
Kiedy **Lupus** otrzyma **Stan Aktualny** poprzez interfejs **Lupin**, rozpoczyna się **Workflow Pętli**, które ma na celu dostarczać **Logikę Pętli** (ang. *Loop Logic*) i składa się z **Elementów Pętli** (ang. *Loop Elements*).

Elementem pętli może być zarówno:

- **Element Lupus**, który działa w warstwie sterowania Kubernetes, a jego misją jest wykonywać **Workflow Pętli**.
- referencja do **Elementu Zewnętrznego**, który działa poza warstwą sterowania Kubernetes, a jego misją jest wykonywać **Część Obliczeniową Logiki Pętli**.

Workflow Pętli jest wyrażany w **LupN**, specjalnej notacji do opisywania workflow pętli, której dokładna specyfikacja znajduje się w Załączniku 3.

Przykładowe **Workflow Pętli** pokazano na rysunku 3.3.



Rysunek 3.3. Przykładowe workflow pętli

- Niebieskie, zaokrąglone prostokąty reprezentują **Elementy Lupus**.
- Czerwone prostokąty oznaczają **Elementy Zewnętrzne**.
- Niebieski obszar wyznaczony linią przerywaną wskazuje elementy działające w warstwie sterowania Kubernetes.
- Wyróżniono **Elementy Lupus** odpowiedzialne za **Ingress** i **Egress**.

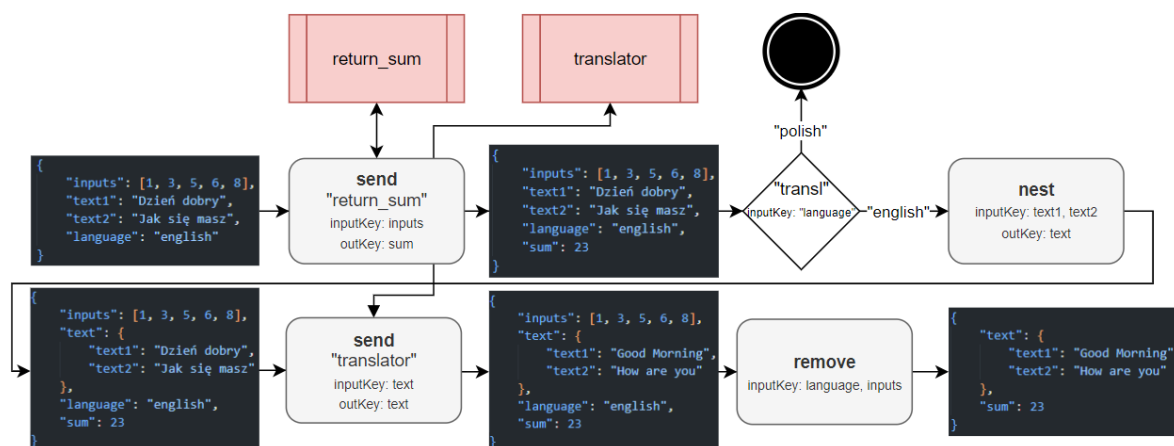
Elementy Zewnętrzne to zazwyczaj serwery HTTP (w szczególności serwery **Open Policy Agent**).

Jeden **Element Lupus** może komunikować się z żadnym, jednym lub wieloma **Elementami Zewnętrznymi**, a liczba ta może się różnić w każdej iteracji pętli.

3.3.8. Dane i Akcje

Dane (ang. *data*) to nośnik informacji w ramach jednej iteracji pętli w formacie JSON. Na wejściu **Elementu Ingress** reprezentują one **Stan Aktualny Systemu Zarządzanego**. Następnie, w trakcie iteracji pętli, **Projektant** decyduje, jakie informacje będą przynosić. Zazwyczaj są to informacje związane z **Logiką Pętli**, takie jak wejścia (ang. *inputs*) do **Elementów Zewnętrznych** oraz ich odpowiedzi. Na końcu iteracji, gdy **Dane** trafiają do **Agenta Egress**, muszą reprezentować **Akcje Sterowania**.

Działanie pojedynczego **Elementu Lupus** jest opisane poprzez **Workflow Akcji**. **Akcje** wykonują różne operacje na **Danych**. Istnieje wiele rodzajów akcji, a kluczowym typem akcji jest "Send", która pozwala komunikować się **Elementowi Lupus** z **Elementem Zewnętrznym**. Inne typy akcji służą organizacji danych.



Rysunek 3.4. Przykładowe workflow akcji

Pełna specyfikacja **Danych** znajduje się w Załączniku 5, zaś pełna specyfikacja **Akcji** w Załączniku 6.

3.3.9. Open Policy Agent

Open Policy Agent (OPA) to otwartoźródłowe narzędzie przeznaczone do definiowania, egzekwowania i zarządzania politykami w systemach oprogramowania. Systemy informatyczne odpytują serwery **OPA** podczas wykonywania operacji, które wymagają decyzji (np. decyzji dostępu, konfiguracji czy zachowania aplikacji). Dzięki takiemu podejściu można oddzielić logikę podejmowania decyzji od kodu aplikacji, co zwiększa modularność, wprowadza centralny punkt zarządzania politykami oraz ułatwia utrzymanie systemów.

Polityki w **OPA** definiowane są w języku Rego. Jest to deklaratywny język stworzony specjalnie na potrzeby **OPA**. Pozwala na tworzenie złożonych reguł i logiki decyzji. Oficjalna strona projektu dostępna jest pod linkiem: <https://www.openpolicyagent.org>.

Listing 1. Przykładowy kod rego

```

allow {
    input.user.role == "admin"
}

allow {
    input.user.role == "user"
    input.action == "read"
}
  
```

Open Policy Agent jest rekomendowanym **Elementem Zewnętrznym** dla **Lupus**.

3.4. Instrukcja Użycia

Niniejsza podsekcja prezentuje krótką instrukcję użycia platformy **Lupus**. **Użytkownikiem Lupus** może stać się dowolna organizacja bądź pojedyncza osoba. Ważne jest, aby zespół **Użytkownika** posiadał kompetencje z zakresu tworzenia oprogramowania. W

zespole **Użytkownika** wyróżniamy rolę **Projektanta**, który jest odpowiedzialny jedynie za projektowanie i wyrażanie **Workflow Pętli** oraz niekoniecznie – zgodnie z Wymaganiem 8 – musi posiadać umiejętności techniczne.

Kiedy dany **Użytkownik** planuje użyć **Lupus** jako **Systemu Sterowania** do rozwiązywania **Problemu Zarządzania** w swoim **Systemie Zarządzanym**, powinien:

1. Zainstalować **Lupus** w swoim klastrze Kubernetes.
2. Zintegrować **System Zarządzany** z **Lupusem** poprzez implementację **Agentów Translacji**.
3. Zaplanować **Workflow Pętli**.
4. Przygotować **Elementy Zewnętrzne**.
5. Wyrazić **Workflow Pętli** w **LupN**.
6. Zaaplikować pliki manifestacyjne zawierające kod **LupN** w klastrze.

Podjęcie przez użytkownika takich akcji nazywamy pojedynczym **Wdrożeniem Lupus**.

3.4.1. Instalacja Lupus

Lupus jest zaimplementowany jako *Zasoby Własne* (ang. *Custom Resources*) w Kubernetes. Instalacja polega na zainstalowaniu tych zasobów w swoim klastrze.

Pełna dokumentacja tego procesu znajduje się w Załączniku 2.

3.4.2. Implementacja Agentów Translacji

To **Użytkownik** podczas **Wdrożenia** jest odpowiedzialny za implementację **Agentów Translacji**. Tylko **Użytkownik** zna specyfikę swojego **Systemu Zarządzanego**. Stąd w zespole **Użytkownika** potrzebne są umiejętności programistyczne.

Podczas implementacji należy kierować się specyfikacją **Interfejsów Lupus** zawartą w Załączniku 4.

3.4.3. Planowanie Logiki Pętli

Workflow Pętli powinno wyrazić **Logikę Pętli**, czyli w każdej iteracji doprowadzić **System Zarządzany** do **Stanu Pożądanego**. **Użytkownik** w tym kroku jedynie modeluje wysokopoziomowo **Workflow Pętli**, rysując np. jego diagram. Dopiero takie spojrzenie podpowie **Użytkownikowi**, jakich **Elementów Zewnętrznych** potrzebuje.

3.4.4. Przygotowanie Elementów Zewnętrznych

Elementem Zewnętrznym może być dowolne oprogramowanie, które **Projektant** ma chęć włączyć w **Workflow Pętli**. Mogą to być już gotowe systemy (np. sztucznej inteligencji), ale równie dobrze **Użytkownik** może stworzyć oprogramowanie samodzielnie. Ważne jest to, aby wystawiały one jakiś sposób komunikacji. Na razie jedynym wspieranym przez **Lupus** sposobem jest komunikacja HTTP. **Użytkownik** musi umożliwić więc komunikację tego typu.

Rekomendowanym przez **Lupus** typem **Elementu Zewnętrznego** jest serwer **Open Policy Agent**. Przygotowanie w tym wypadku polega na uruchomieniu takiego serwera oraz wgraniu mu odpowiednich polityk. **Użytkownik** jednakże może wydewelopować dowolny serwer HTTP.

Ostatnim możliwym **Elementem Zewnętrznym** są **Funkcje Użytkownika**. Są to funkcje w kodzie *kontrolera* zasobu **Lupus Element**, które są definiowane przez **Użytkownika**. Stanowią one alternatywę dla serwerów HTTP, gdy specjalne wdrażanie takowych może się okazać zbyt kosztowne. Przykładowo, jeśli logika wykonywana przez dany **Element Zewnętrzny** miałaby mieć tylko kilka linijek kodu, dużo łatwiej użyć **Funkcji Użytkownika**. Ich dokładniejszy opis znajduje się w Sekcji 4.3.6.

3.4.5. Wyrażenie workflow pętli

Gdy już całe **Workflow Pętli** oraz **Elementy Zewnętrzne** są gotowe, czas wyrazić je w notacji **LupN**. Za jej pomocą wyraża się **Workflow Pętli** jako zbiór **Elementów Lupus**, połączenia między nimi oraz specyfikacja każdego z nich.

3.4.6. Aplikacja plików manifestacyjnych

Aby stworzyć pętlę opisaną przez **Kod LupN** w pliku manifestacyjnym zasobu **Master**, należy wykonać komendę z listingu 2, która tworzy instancję *obiektu API* typu **Master**. Kontroler tego zasobu stworzy odpowiednie obiekty API typu **Element** według specyfikacji w pliku **LupN**. Podczas iteracji pętli, kontroler każdego **Elementu** interpretuje ich specyfikację, wykonując odpowiednie **Akcje** na **Danych**. **Workflow Akcji** również specyfikowane jest w notacji **LupN** (Załącznik 3).

Listing 2. Stworzenie zasobu *Master*

```
kubectl apply -f <nazwa_pliku>
```

4. Implementacja

4.1. Wstęp

Sekcja ta opisuje wydzieloną z architektury implementację Lupus. Pierwsza jej części opisuje kluczowe mechanizmy Kubernetes, za pomocą których zaimplementowano Lupus. Druga część opowiada o decyzjach podjętych podczas delopmentu platformy. To tu objawia się badawcza natura pracy.

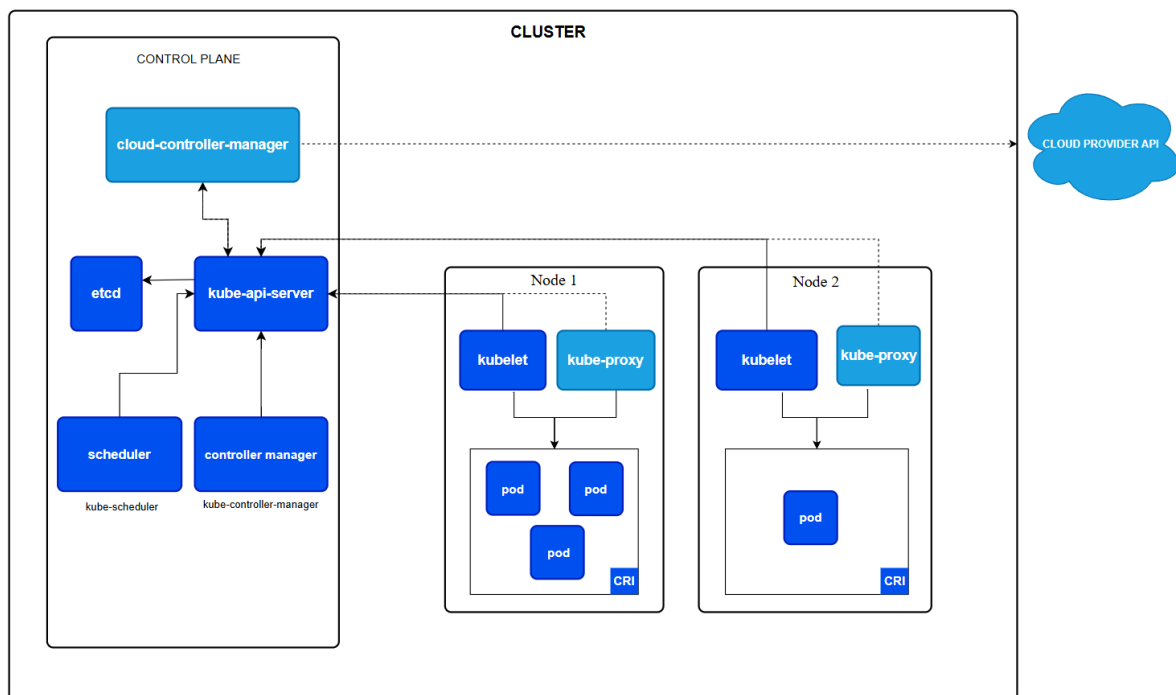
4.2. Mechanizmy Kubernetes stojące za Lupus

Niniejszy dokument zakłada znajomość czytelnika platformy Kubernetes na podstawowym poziomie.

4.2.1. Kontroler

Działanie Kubernetes opiera się na zamkniętej pętli sterowania [odnośnik do artykułu]. **Aktualny Stan** systemu zapisany jest w bazie etcd. **Stan Pożądany** wyrażony jest poprzez pliki manifestacyjne. Każdy *Obiekt API* posiada swoją sekcję *'spec'*, to ona definiuje pożądany stan danego obiektu. Każdy obiekt ma swój kontroler, który rekuncyluje jego aktualny stan do stan pożądanego. Kontroler jest to proces działający w warstwie sterowania Kubernetes. Każdy typ (ang. *Kind*) wbudowanych zasobów (ang. *buil-in resources*) ma swój

kontroler stworzony przez zespół Kubernetes. Kontrolery każdego typu zasobu działają w podzie 'kube-controller-manager'.



Rysunek 4.1. Architektura Kubernetes

Flow pracy kontrolera pokazano na rysunku 4.2. Gdy 'kube-api-server' otrzyma żądanie zmiany danego obiektu API zanim 'kube-api-server' zleci utrwalenie tych zmian wykona tzw. *webhooki* do kontrolera danego obiektu. Webhooki nie są jednakże istotne z punktu widzenia niniejszej pracy dyplomowej¹¹. Następnie, gdy doszło do zmian w obiekcie, kontroler zostaje o tym poinformowany. Jego misją jest rekoncylacja, czyli porównanie aktualnego stanu obiektu ze stanem pożądanym. Kontroler zawiera w sobie logikę rekoncylacji, która przybliży oba stany do siebie. Wykonanie logiki wiąże się często z przeprowadzeniem różnych akcji przez kontroler w innych częściach klastra.

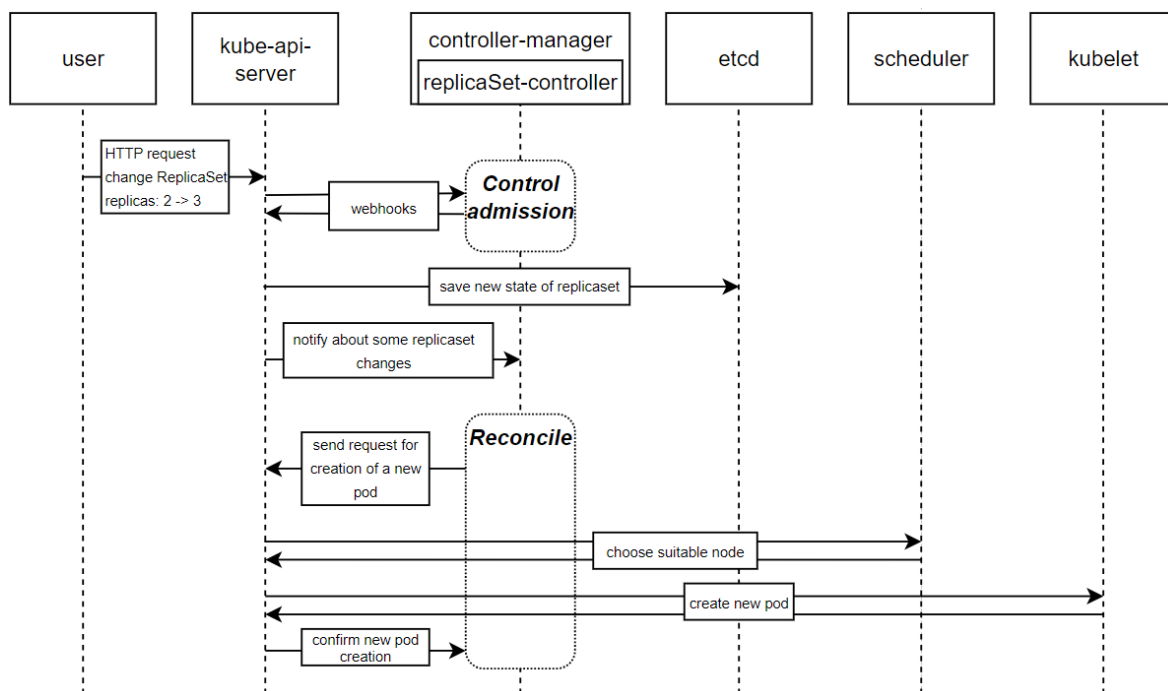
4.2.2. Zasoby własne

Zasoby własne (ang. *Custom Resources*) rozszerzają wbudowane zasoby (ang. *built-in resources*) o niestandardowe, zdefiniowane przez użytkownika. Najczęściej tworzone są w celu zarządzania konfiguracją skomplikowanych lub stanowych aplikacji.

Czasami wbudowane typy zasobów (jak Pod'y, Wdrożenia (ang. *Deployments*), Serwisy (ang. *Services*)) nie są dla nas wystarczające, dlatego Kubernetes udostępnia możliwość rejestrowania nowych typów. Aby tego dokonać należy stworzyć plik manifestacyjny YAML typu *Custom Resource Definition* a następnie zaaplikować go w klastrze. Po pomyślnej operacji, będzie można tworzyć obiekty nowego typu.

¹¹Dla zainteresowanych tematem: link

4. Implementacja



Rysunek 4.2. Flow pracy kontrolera

4.2.3. Operatory

Mechanizm ten służy do rozszerzania możliwości Kubernetes. Kiedy tworzymy Zasoby Własne (ang. *Custom Resources*), możemy również implementować dla nich kontrolery. Takie podejście nazywane jest "wzorem operatora" ("operator-pattern"). Często takie kontrolery nazywamy po prostu "operatorami". Nazwa "operator" wywodzi się z idei, że taki kontroler zazwyczaj zastępuje rzeczywistego operatora (człowieka), który zarządzałby aplikacją (dla której wdrożenie wymagało zdefiniowania Zasobu Własnego).

4.2.4. Kubebuilder

Kubebuilder jest frameworkiem programistycznym do tworzenia *textit*Zasobów Własnych oraz ich *Operatorów*. Pozwala na to, aby zaprogramować sekcje zaznaczone na rysunku 4.3.

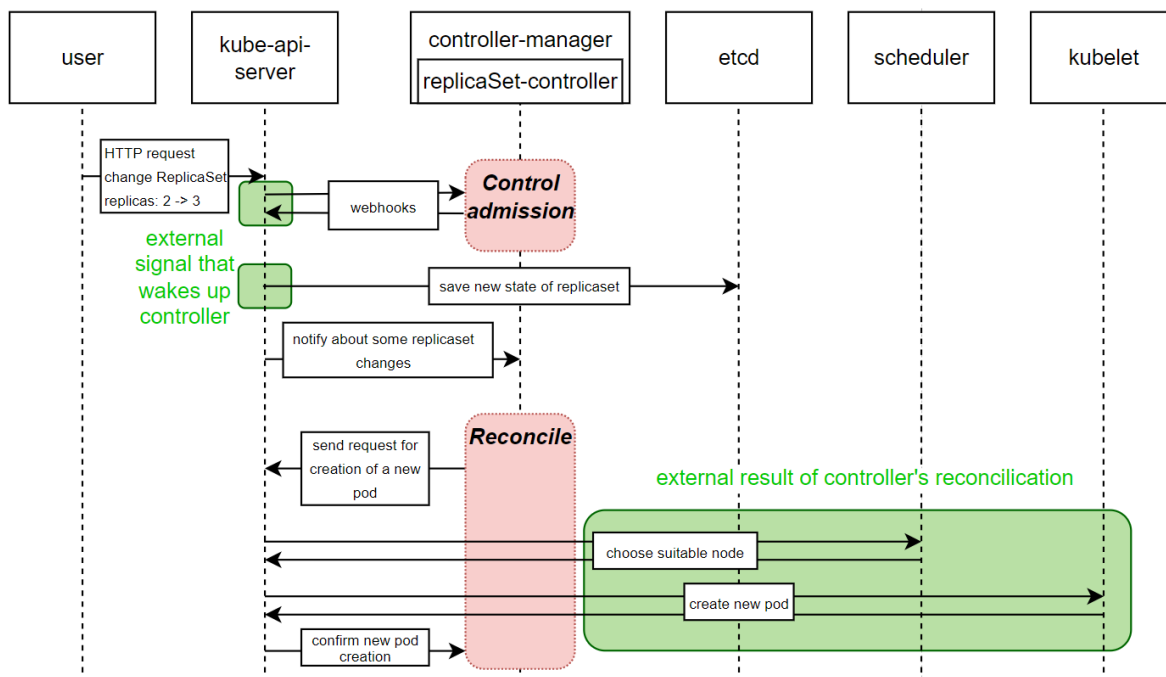
4.3. Decyzje podjęte podczas developmentu

4.3.1. Wstęp

W tym rozdziale objawia się badawcza natura pracy. Podczas realizacji projektu zbadano wiele podejść oraz rozwiązano wiele problemów implementacyjnych, aby dojść do postawionej w sekcji 3 architektury. Każda podsekcja omawia po jednym z aspektów – od wymagań do implementacji.

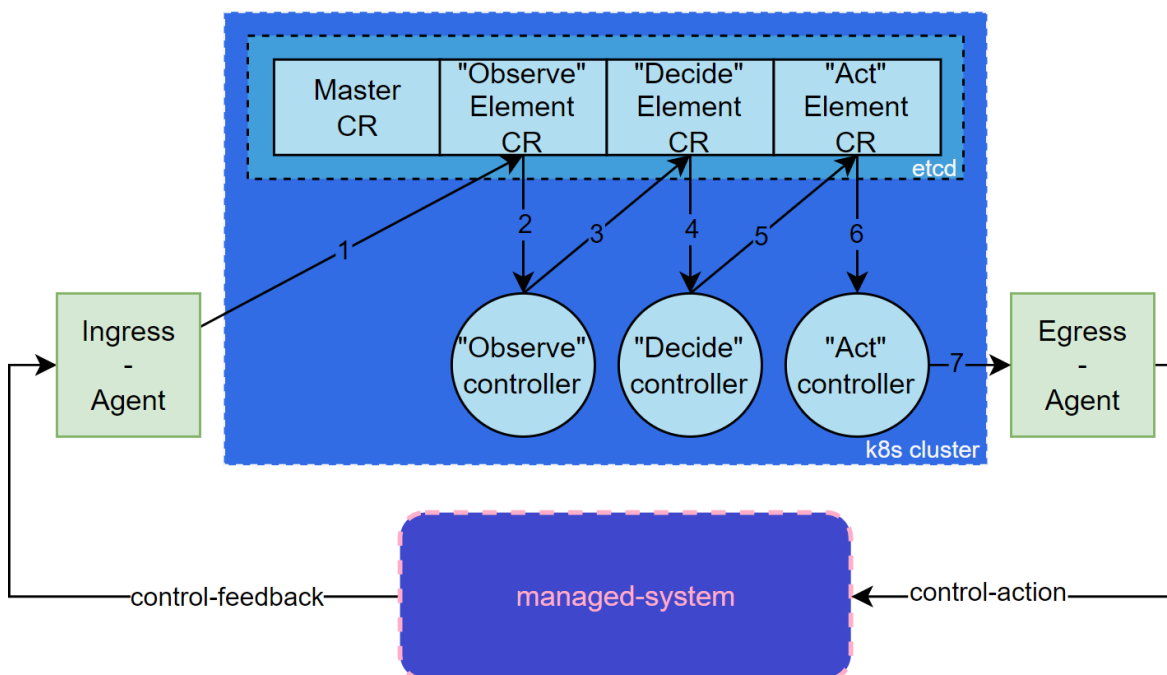
4.3.2. Komunikacja pomiędzy Elementami Lupus

Pierwszym krokiem w implementacji było wymyślenie sposobu na komunikację pomiędzy **Elementami Lupus**. Z racji, że **Elementy Lupus** są *Zasobami Własnymi*, wykorzystano



Rysunek 4.3. Zaznaczenie elementów dających się zaprogramować podczas flow pracy kontrolera

tu natywne mechanizmy Kubernetes opisane w podsekcji 4.2 i 4.3. Ideą było to, że *Operator* jednego zasobu dokonuje zmian w obiekcie API innego elementu, co z kolei wywoła ponownie Operator z innym obiektem wejściowym. **Elementy Lupus** modyfikują nawzajem swój Status, a dokładniej jego pole input, przekazując tam swoją **Finalną Postać Danych**.



Rysunek 4.4. Komunikacja pomiędzy Elementami Lupus

Działa to w sposób ukazany na rysunku 4.4. Początkowo **Elementy Lupus** mogły przyjąć

postać jednego z 4 typów: Observe, Learn, Decide lub Execute¹². **Agent Ingress** modyfikował Status obiektu Observe, natomiast Operator Observe modyfikował Statusy obiektów Learn oraz Decide. Na koniec modyfikacje otrzymywał obiekt Execute, którego Operator przekazywał swoje **Dane** do **Agentu Egress**.

Z czasem jednak, aby nie narzucać konkretnej struktury pętli, zrezygnowano z 4 typów i stworzono jeden uniwersalny typ, którego Operator jest w stanie wykonać **Workflow Akcji**, co pozwalało na wyrażenie logiki dowolnego z 4 poprzednich typów. W ten sposób spełniono Wymaganie 5.

4.3.3. Dane

Wymaganie 5 nakazuje, aby **Elementy Lupus** były sterowane danymi (*data-driven*). **Dane** oraz **Workflow Akcji** spełniają to wymaganie. Dzięki nim **Elementy Lupus** mogą wykonać dowolny zestaw **Akcji** na **Danych**, dając **Projektantowi** pewną elastyczność. To właśnie **Dane** przekazywane są jako pole input w statusie obiektu API **Elementu Lupus**.

Z racji uniwersalności **Danych**, należało wybrać format, który pozwoli na ich możliwie dużą dowolność. Wybór padł na JSON, ze względu na jego ogólnie przyjęty standard i możliwości reprezentacji dowolnej struktury danych.

Kubernetes z kolei wymaga, aby pola umieszczane w Statusie Zasobu miały konkretny typ. Dokonano więc analizy, jaki typ nadaje się do reprezentacji obiektu JSON. Wybór padł na `RawExtension`. Jest to typ zdefiniowany przez zespół Kubernetes, używany do obsługi dowolnych surowych danych w formacie JSON lub YAML. Należy do pakietu `k8s.io/apimachinery/pkg/runtime` i jest często stosowany, gdy zasób musi osadzić lub pracować ze strukturą danych, która jest elastyczna, ale jednocześnie uporządkowana. `RawExtension` spełnia te wymagania.

Listing 3. Definicja struktury Go reprezentującej status Elementu Lupus

```
// ElementStatus defines the observed state of Element
type ElementStatus struct {
    // Input contains operational data
    Input runtime.RawExtension `json:"input" `
    // Timestamp of the last update
    LastUpdated metav1.Time `json:"lastUpdated" `
}
```

`RawExtension` to typ, którego użyjemy do przenoszenia **Danych** między **Elementami Lupus**. Pozostaje kwestia reprezentacji tych **Danych** w Operatorach **Lupus**.

Listing 4. Definicja struktury Go `RawExtension` w paczce `k8s.io/apimachinery/pkg/runtime`

```
type RawExtension struct {
    Raw []byte `json:"- "` // Serialized JSON or YAML data
    Object Object // A runtime.Object representation
}
```

¹²W myśl pętli ODA

Zamierzonym celem `RawExtension` według deweloperów Kubernetes jest umożliwienie deserializacji do określonej, znanej struktury. Jednak ze względu na Wymaganie 5 taka struktura nie istnieje. Potrzebujemy więc natywnej struktury Go, która jest w stanie reprezentować dowolny obiekt JSON. Pierwszym pomysłem, który się nasuwa, jest użycie typu Go `interface`, ponieważ może on reprezentować dowolne dane. Problem z `interface` polega jednak na tym, że nie można na nim operować – nie udostępnia żadnego interfejsu do interakcji. Jest to typ podstawowy.

Drugim pomysłem na reprezentację JSON-a było użycie `map[string] interface`, ponieważ większość instancji JSON-a to faktycznie obiekty klucz-wartość. Klucze w tym przypadku są typu `string`, a wartości mogą być dowolne (stąd użycie `interface`) w Go. W większości przypadków obiekty JSON zawierają kilka głównych pól (ang. *top-level fields*), co idealnie pasuje do reprezentacji `map[string] interface`.

Tak właśnie narodziło się pojęcie **Danych**. **Dane** to w rzeczywistości struktura opakowująca i dająca odpowiedni interfejs (ang. *Wrapper*) dla wspomnianej wcześniej mapy.

Listing 5. Definicja struktury Go dla *Danych*

```
type Data struct {
    Body map[string] interface {}
}
```

Ta struktura posiada bogaty zestaw funkcji (metod), które pełnią rolę interfejsu do pracy z **Danymi**. Metody te są wywoływane podczas wykonywania **Akcji** i zazwyczaj – z wyjątkiem metod `Get` i `Set` – każda metoda odpowiada dokładnie jednej akcji. Kluczowym konceptem **Danych** jest **Pole Danych** (ang. *Data-field*). Jest ono odpowiednikiem pola w JSON-ie. Każde pole jest identyfikowane przez swój **Klucz** i przechowuje wartość. Za pomocą metody `Get` możemy pobrać wartość znajdującą się pod danym kluczem, a przy użyciu `Set` możemy ustawić nową wartość dla pola wskazanego określonym kluczem.

Nie obejmuje to jednak wszystkich obiektów JSON, jakie istnieją. JSON pozwala, aby element na najwyższym poziomie był tablicą. Nakłada to pewne ograniczenia na projektowanie pętli. Szczególnie JSON reprezentujący aktualny stan **Systemu Zarządzanego**, wysyłany przez **Interfejs Lupin**, musi być serializowalny do `map[string] interface`. Oznacza to, że nie może być:

- typem prymitywnym,
- tablicą,
- obiektem JSON z kluczami innymi niż `string`.

4.3.4. Polimorfizm w Go

Notacja **LupN** pozwala, aby wiele jej obiektów posiadało swój typ. Przykładowo akcje są różnorakiego typu. Mają pewną część wspólną, ale też pola szczególne dla każdego typu. Go jest statycznie typowanym językiem, który nie posiada dziedziczenia ani tradycyjnego obiektowego polimorfizmu. Dlatego dokonano analizy jak w Go osiągnąć tę wielopostaciowość.

Polimorfizm poprzez interfejsy

Natywnym sposobem na polimorfizm w Go jest ten osiągniany poprzez interfejsy. To zagadnienie najlepiej tłumaczy poniższy kod.

Listing 6. *Przykład polimorfizmu poprzez interfejsy*

```
package main

import (
    "fmt"
)

type Forwarder interface {
    Forward() string
}

type NextElement struct {
    Name string
}

func (e *NextElement) Forward() string {
    return fmt.Sprintf("Forwarding_to_element:_%s", e.Name)
}

type Destination struct {
    URL string
}

func (d *Destination) Forward() string {
    return fmt.Sprintf("Forwarding_to_destination:_%s", d.URL)
}

func ProcessForwarder(f Forwarder) {
    fmt.Println(f.Forward())
}

func main() {
    element := &NextElement{Name: "Element1"}
    destination := &Destination{URL: "https://example.com"}

    ProcessForwarder(element)
    ProcessForwarder(destination)
}
```

Jak to działa

- **Definicja interfejsu:** Forwarder definiuje metodę Forward(), którą muszą zaimplementować określone typy.
- **Konkretnie implementacje:** NextElement oraz Destination implementują metodę Forward().
- **Zastosowanie:** Każdy typ spełniający interfejs Forwarder może być przekazywany do funkcji oczekujących obiektu Forwarder.

Polimorfizm poprzez wskaźniki oraz pole dyskryminatora

Kolejnym potężnym i idiomatycznym wzorcem w Go jest **polimorfizm w stylu Go z użyciem wskaźników**, gdzie struktura posiada opcjonalne pola wskaźnikowe, a pole type (znacznik) określa, które z tych pól jest istotne w czasie działania programu.

Listing 7. Przykład polimorfizmu poprzez wskaźniki oraz pole dyskryminatora

```
type Next struct {
    // Type specifies the type of next loop-element
    Type string `json:"type"`
    // List of input keys (Data fields) that have to be forwarded
    Keys []string `json:"keys"`
    // One of the fields below is not null
    Element *NextElement `json:"element,omitempty"`
    Destination *Destination `json:"destination,omitempty"`
}

type NextElement struct {
    Name string `json:"name"`
}

type Destination struct {
    URL string `json:"url"`
}

func (n *Next) Validate() error {
    if n.Type == "element" && n.Element == nil {
        return fmt.Errorf("Element_must_be_set
        _____for_type_'element'")
    }
    if n.Type == "destination" && n.Destination == nil {
        return fmt.Errorf("Destination_must_be_set
        _____for_type_'destination'")
    }
    if n.Element != nil && n.Destination != nil {
        return fmt.Errorf("Only_one_of_Element
        _____or_Destination_can_be_set")
    }
}
```

```

    }
    return nil
}

```

Jak to działa

Unia tagowana to wzorzec, w którym pole tag określa, którą z kilku możliwych reprezentacji danych wykorzystuje dany obiekt. W Go jest to realizowane poprzez kombinację:

- Pole dyskryminujące typ (np. `Type string`).
- Pola wskaźnikowe dla możliwych wariantów. Jeśli dane pole nie występuje w aktualnej reprezentacji obiektu, jego wartość jest po prostu `nil`.
- Podczas działania programu możemy zweryfikować, której reprezentacji danych używa obiekt, i odpowiednio na tej podstawie podjąć działanie.

Podczas działania programu możemy zweryfikować, której reprezentacji danych używa obiekt, i odpowiednio na tej podstawie podjąć działanie.

Porównanie

Porównanie zostało przedstawione w tabeli 4.1

Funkcja	Polimorfizm przez wskaźniki	Polimorfizm przez interfejsy
Zachowanie w czasie działania	Używa dyskryminatora (<code>Type</code>) i pól wskaźnikowych	Używa implementacji metod do polimorfizmu
Bezpieczeństwo typów	Wymaga jawnej walidacji	Wymuszane podczas kompilacji za pomocą interfejsów
Serializacja	Bezproblemowa z JSON	Może wymagać niestandardowego marshaling
Rozszerzalność	Dodaj nowe pola wskaźnikowe i zaktualizuj enum <code>Type</code>	Dodaj nowe typy implementujące interfejs
Łatwość użycia	Prosta, ale wymaga ręcznej walidacji	Czysta i idiomatyczna w Go
Wspólne zachowanie	Wymaga zewnętrznej logiki	Enkapsulowane w metodach interfejsu

Tabela 4.1. Porównanie polimorfizmu przez wskaźniki i przez interfejsy w Go.

Polimorfizm przez wskaźniki oferuje przejrzystą reprezentację danych, która może być łatwo serializowana i deserializowana (np. do formatu JSON lub YAML). Dodatkowo wspiera włączenie pola `type` jako części modelu danych, dzięki czemu może być ono również przechowywane lub przesyłane. Z drugiej strony polimorfizm przez interfejsy jest natywny dla Go, bardziej przejrzysty i gwarantuje silne sprawdzanie typów podczas kompilacji.

Podsumowanie:

- Polimorfizm przez wskaźniki jest preferowany w aplikacjach skoncentrowanych na danych.

- Polimorfizm przez interfejsy jest preferowany w aplikacjach skoncentrowanych na zachowaniach.

W **Lupus** polimorfizm był potrzebny do reprezentacji różnych typów (odmian) niektórych **Obiektów LupN**, takich jak **Actions** lub **Next**, dlatego wybrano polimorfizm poprzez wskaźniki oraz pole dyskryminatora.

4.3.5. Dwa rodzaje workflow

W **Lupus** występują dwa rodzaje *workflow*: **Workflow Pętli**, które definiuje przepływ pracy **Elementów Lupus**, oraz **Workflow Akcji**, które definiuje przepływ **Akcji** we wnętrzu pojedynczego elementu. Występują między nimi pewne różnice w możliwościach, wynikające ze sposobu implementacji komunikacji między ich węzłami.

- Węzły w **Workflow Pętli** komunikują się ze sobą poprzez pobudzanie operatorów.
- Węzły w **Workflow Akcji** komunikują się poprzez pamięć RAM zaalokowaną przez pojedynczy operator **Elementu Lupus**.

Różnice są widoczne w specyfikacji LupN w Załączniku 3.

4.3.6. Funkcje użytkownika

Zgodnie z Wymaganiem 6, **Elementy Lupus** nie wykonują **Części Obliczeniowej Logiki Pętli**. Zamiast tego, jest ona delegowana do **Elementów Zewnętrznych**.

Pojawiają się tutaj dwa aspekty:

- Co w sytuacji, gdy ktoś potrzebuje wykonać małą i prostą operację na **Danych** (np. dodanie dwóch pól), a wdrażanie dedykowanego serwera HTTP jest nieekonomiczne?
- Każda platforma ramowa powinna być rozszerzalna. Powinniśmy zapewnić mechanizm umożliwiający rozszerzanie naszej platformy.

Z tych dwóch powodów wdrożono funkcję o nazwie „Definiowane przez użytkownika, wewnętrzne funkcje Go” (lub w skrócie – **Funkcje Użytkownika**).

Użytkownik może definiować własne fragmenty kodu Go jako funkcje i wywoływać je jako jedną z **Destynacji** w akcji **Send**.

W repozytorium kodu z projektem Kubebuilder znajduję plik o nazwie `user-functions.go`¹³. To w nim **Użytkownik** może definiować swoje funkcje. Plik posiada już jedną przykładową funkcję.

Listing 8. Przykładowa funkcja użytkownika

```
// Exemplary user-function. It just returns the input
func (UserFunctions) Echo(input interface{}) (interface{}, error) {
    return input, nil
}
```

¹³<https://github.com/0x41gawor/lupus/blob/master/lupus/internal/controller/user-functions.go>

Funkcja jako argument przyjmuje interface, który będzie reprezentował **pole danych**.¹⁴ Zwracanym typem jest również interface, który zostanie przez akcję send wpisany jako **pole danych**.

Nazwa `UserFunctions` jest strukturą, która gromadzi funkcje użytkownika.

Listing 9. *Struktura składująca funkcje użytkownika jako swoje metody.*

```
// UserFunctions struct for user-defined, internal functions
type UserFunctions struct {}
```

Powstaje teraz pytanie jak wywołać takową funkcję w notacji **LupN**. Na szczęście funkcje w Go mogą być używane jako typy i przechowywane jako wartości. Dzięki temu mogą zostać zapisane jako wartości w mapie klucz-wartość, gdzie nazwy funkcje typu `string` pełnią rolę kluczy.

Listing 10. *Mapa przechowująca funkcje użytkownika*

```
// A global map to store function references
var FunctionRegistry = map[string]func(input interface{}) (interface{}, error){}
```

Następnie, z pomocą biblioteki `reflect`¹⁵ można zaimplementować funkcję, która iteruje po metodach struktury `UserFunctions` i zapisuje je do powyższej mapy jako wartości, dla których kluczami są nazwy funkcji.

Listing 11. *Funkcja zaplaniająca mapę funkcji użytkownika*

```
// RegisterFunctions dynamically registers user-defined functions
func RegisterFunctions(target interface{}) {
    t := reflect.TypeOf(target)
    v := reflect.ValueOf(target)

    for i := 0; i < t.NumMethod(); i++ {
        method := t.Method(i)

        // Ensure the method matches the required signature
        if method.Type.NumIn() == 2 && // Receiver + input
            method.Type.NumOut() == 2 && // Output + error
            method.Type.In(1).Kind() == reflect.Interface && // Input:
            method.Type.Out(0).Kind() == reflect.Interface && // Output
            method.Type.Out(1).Implements(reflect.TypeOf((*error)(nil)))

        funcName := method.Name
        FunctionRegistry[funcName] = func(input interface{}) (interface{}, error) {
            // Call the user-defined function
            result := method.Func.Call([]reflect.Value{v, reflect.ValueOf(input)})
            return result[0].Interface(), result[1].Interface().(error)
        }
    }
}
```

¹⁴ Pola lub pola, gdyż pod tym pojęciem może też się kryć użycie `"**"`

¹⁵<https://pkg.go.dev/reflect>

```

        // Handle result[1] (error) being nil
        var err error
        if !result[1].IsNil() {
            err = result[1].Interface().(error)
        }

        return result[0].Interface(), err
    }
}
}
}

```

Funkcja widoczna na listingu 11 jest wywoływana podczas inicjalizacji paczki kontrolera z strukturą `UserFunctions` jako argument.

Listing 12. Inicjaliza mapy

```

func init() {
    // Fill in the FunctionRegistry map with functions defined as a method of User
    RegisterFunctions(UserFunctions{})
}

```

Od tego momentu funkcja może zostać wywołana po nazwie w funkcji obsługującej akcję `Send` w interpreterze Operatora Elementu `Lupus`.

Listing 13. Wywołanie funkcji użytkownika w akcji `send`

```

func sendToGoFunc(funcName string, body interface{}) (interface{}, error) {
    if fn, exists := FunctionRegistry[funcName]; exists {
        return fn(body)
    } else {
        return nil, fmt.Errorf("no_such_UserFunction_defined")
    }
}

```

W notacji **LupN** wystarczy zapis:

Listing 14. Użycie funkcji użytkownika w `LupN`

```

- name: "bounce"
  type: send
  send:
    inputKey: "field1"
    destination:
      type: gofunc
      gofunc:

```

name: **echo**
outputKey: "field2"

5. Test platformy na emulatorze sieci 5G

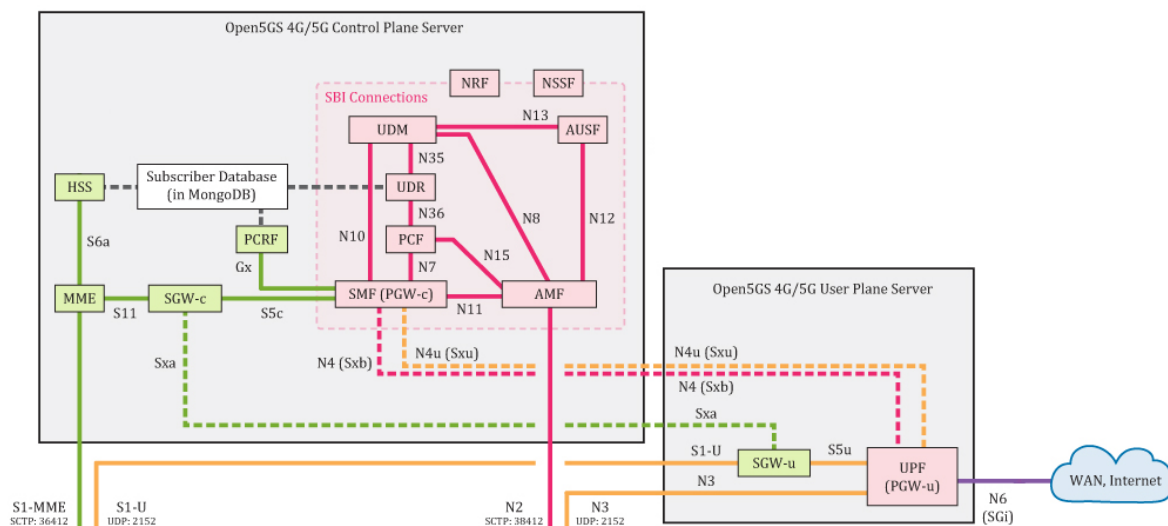
5.1. Wstęp

Niniejsza sekcja omawia przypadek użycia (ang. *use case*) proponowanej architektury na emulatorze sieci 5G stworzonej za pomocą Open5GS oraz UERANSIM. Przedstawia emulator sieci 5G sam w sobie. Następnie prezentuje przykładowy **Problem Zarządzania** oraz zaaplikowanie **Lupus** w celu rozwiązywania tego problemu. Sekcja omawia proces pojedynczego **Wdrożenia Lupus**.

5.2. Emulator sieci 5G

5.2.1. Open5GS

Pierwszym komponentem emulatora jest otwartoźródłowe oprogramowanie Open5GS. Oficjalna strona projektu znajduje się tutaj: <https://open5gs.org/open5gs/>. Open5GS emuluje sieć rdzeniową (ang. *core network*) czwartej oraz piątej generacji sieci standaryzowanych przez 3GPP. Architektura oprogramowania przedstawiona jest na rysunku 5.1.

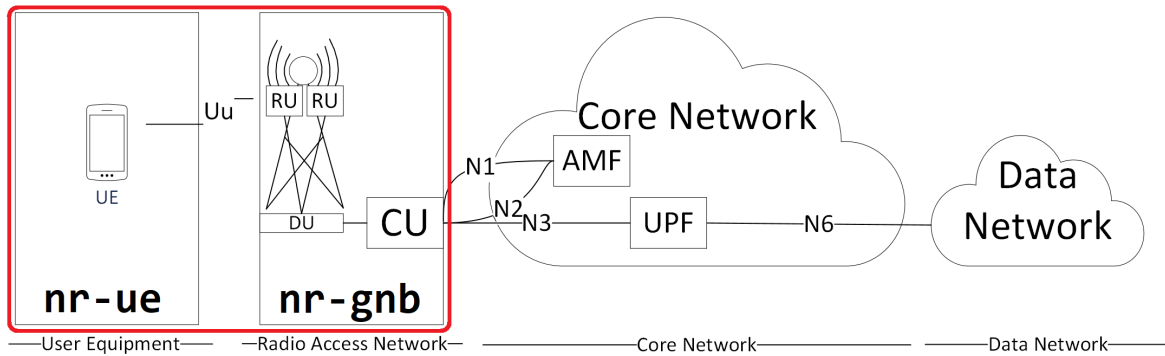


Rysunek 5.1. Architektura Open5GS

5.2.2. UERANSIM

Drugim komponentem emulatora jest otwartoźródłowe oprogramowanie UERANSIM. Oficjalna strona projektu: <https://github.com/aligungr/UERANSIM>. UERANSIM symuluje Radiową Sieć Dostępową (ang. *Radio Access Network (RAN)*) wraz z Terminalami Abonenta (ang. *User Equipment (UE)*). UERANSIM składa się z dwóch programów nr-ue

oraz nr-gnb, które można uruchamiać w wielu instancjach. Ich rolę w architekturze sieci 5G przedstawiono na rysunku 5.2 w czerwonej obwódce.



Rysunek 5.2. Architektura UERANSIM

5.2.3. Wdrożenie na Kubernetes

Emulator sieci 5G złożony z Open5GS oraz UERANSIM wdrożono na klastrze Kubernetes według repozytorium: <https://github.com/niloysh/open5gs-k8s>. Przeprowadzenie kroków wskazanych w repozytorium skutkuje stanem podów w klastrze przedstawionym na rysunku 5.3.

```
ejek@open5gs-vm:~$ kubectl get pods -A
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
cluster-network-addons	cluster-network-addons-operator-5ddc679dbc-z4bpq	2/2	Running	48 (9m28s ago)	51d
cluster-network-addons	ovs-cni-amd64-f48tz	1/1	Running	26 (9m28s ago)	51d
kube-flannel	kube-flannel-ds-rgzcc	1/1	Running	25 (9m28s ago)	51d
kube-system	coredns-5dd5756b68-77dc9	1/1	Running	24 (9m28s ago)	51d
kube-system	coredns-5dd5756b68-wtspf	1/1	Running	24 (9m28s ago)	51d
kube-system	etcd-open5gs-vm	1/1	Running	24 (9m28s ago)	51d
kube-system	kube-apiserver-open5gs-vm	1/1	Running	24 (9m28s ago)	51d
kube-system	kube-controller-manager-open5gs-vm	1/1	Running	29 (9m28s ago)	51d
kube-system	kube-multus-ds-92ndr	1/1	Running	24 (9m28s ago)	51d
kube-system	kube-proxy-25kj5	1/1	Running	24 (9m28s ago)	51d
kube-system	kube-scheduler-open5gs-vm	1/1	Running	30 (9m28s ago)	51d
kube-system	metrics-server-596474b58-x5l28	1/1	Running	19 (9m28s ago)	43d
open5gs	mongodb-0	1/1	Running	23 (9m28s ago)	45d
open5gs	open5gs-amf-57f756b4cb-f4hv8	1/1	Running	31 (9m28s ago)	43d
open5gs	open5gs-ausf-cc8ccb99-qsjvb	1/1	Running	36 (9m28s ago)	45d
open5gs	open5gs-bsf-57d59cd58f-bhm6p	1/1	Running	36 (9m28s ago)	45d
open5gs	open5gs-nrf-c85cfcdd4-c8mqg	1/1	Running	25 (9m28s ago)	45d
open5gs	open5gs-nssf-748c6b66d8-2m84p	1/1	Running	35 (9m28s ago)	45d
open5gs	open5gs-pcf-5999669bdd-6ds12	1/1	Running	36 (9m28s ago)	45d
open5gs	open5gs-scp-6b7d795854-j8d74	1/1	Running	36 (9m28s ago)	45d
open5gs	open5gs-smf1-5b4df46b97-5dt47	1/1	Running	37 (9m28s ago)	45d
open5gs	open5gs-smf2-c96c8c9c5-2rq2p	1/1	Running	37 (9m28s ago)	45d
open5gs	open5gs-udm-6bbdbb86c-nxcgl	1/1	Running	40 (9m28s ago)	45d
open5gs	open5gs-udr-64d75654f5-x6zt4	1/1	Running	37 (9m28s ago)	45d
open5gs	open5gs-upf1-65c4f9d55d-2gbl6	1/1	Running	16 (9m28s ago)	39d
open5gs	open5gs-upf2-594d6bc47f-tmqkb	1/1	Running	13 (9m28s ago)	36d
open5gs	open5gs-webui-669b694b5c-hfrtf	1/1	Running	22 (9m28s ago)	45d
open5gs	ueransim-gnb-64679ddbb7-nj6lr	1/1	Running	21 (9m28s ago)	43d
openebs	openebs-localpv-provisioner-56d6489bbc-2kd77	1/1	Running	32 (9m28s ago)	51d
openebs	openebs-ndm-fd6fp	1/1	Running	48 (8m19s ago)	51d
openebs	openebs-ndm-operator-5d7944c94d-8f8cn	1/1	Running	24 (9m28s ago)	51d

```
ejek@open5gs-vm:~$
```

Rysunek 5.3. Stan podów w klastrze

Programy UERANSIM typu nr-ue będą uruchamiane poza klastrem. Tej decyzji dokonano z prostego powodu. System operacyjny hostujący klastrowy (Ubuntu 22.04 LTS) posiada dużo większe możliwości, jeśli chodzi o narzędzia generowania ruchu, niż okrojony system operacyjny użyty w podach UERANSIM.

5.3. Wdrożenie Lupus

Sekcja opisuje użycie **Lupus** jako **Systemu Sterowania** dla **Problemu Zarządzania** opisanego w następnej podsekcji dla emulatora sieci 5G, który przyjmie z kolei rolę **Systemu Zarządzanego**.

5.3.1. Problem Zarządzania

Celem jest utrzymanie żądanych wartości zasobów dla funkcji płaszczyzny użytkownika (ang. *User Plane Function*, *UPF*) na optymalnym poziomie. Optymalność oznacza zapewnienie wystarczających zasobów do obsługi obciążenia, ale jednocześnie unikanie ich nadmiernego przydziału, co pozwala na ograniczenie niepotrzebnych kosztów związanych z wynajmem zasobów chmurowych.

W ramach wdrożenia Open5GS, w tym dla komponentu UPF, istnieje kilka instancji uruchomieniowych (ang. *Deployments*), jak pokazano na rysunku 5.4.



NAME	READY	UP-TO-DATE	AVAILABLE	AGE
open5gs-amf	1/1	1	1	45d
open5gs-ausf	1/1	1	1	45d
open5gs-bsf	1/1	1	1	45d
open5gs-nrf	1/1	1	1	45d
open5gs-nssf	1/1	1	1	45d
open5gs-pcf	1/1	1	1	45d
open5gs-scp	1/1	1	1	45d
open5gs-smf1	1/1	1	1	45d
open5gs-smf2	1/1	1	1	45d
open5gs-udm	1/1	1	1	45d
open5gs-udr	1/1	1	1	45d
open5gs-upf1	1/1	1	1	45d
open5gs-upf2	1/1	1	1	45d
open5gs-webui	1/1	1	1	45d
ueransim-gnb	1/1	1	1	43d

Rysunek 5.4. Wdrożenia Open5GS, w tym instancje UPF

Każdy *pod* w klastrze Kubernetes posiada zdefiniowane wartości dotyczące przydziału zasobów:

- *requests* – określa ilość zasobów CPU oraz pamięci operacyjnej (RAM), jaką dany pod żąda od środowiska chmurowego. Kubernetes zapewnia, że *węzeł* posiada wystarczające zasoby, aby spełnić te wymagania przed przydzieleniem poda do *węzła*.
- *limits* – maksymalna ilość zasobów, jaką pod może zużyć. Przekroczenie tych wartości skutkuje zakończeniem procesu (np. jego ubiciem w celu ochrony środowiska).

Dodatkowo możliwe jest monitorowanie rzeczywistego (aktualnego, chwilowego) wykorzystania procesora i pamięci operacyjnej.

Mechanizmy dostępne w Kubernetes

Kubernetes umożliwia:

- monitorowanie bieżącego zużycia CPU oraz pamięci dla konkretnego poda,
- monitorowanie zdefiniowanych wartości `requests` i `limits` dla konkretnego poda,
- aktualizację wartości `requests` i `limits` dla konkretnego poda.

Wdrożenie komponentu UPF w repozytorium `open5gs-k8s` wykorzystuje następujące wartości zasobów:

Listing 15. Konfiguracja zasobów dla UPF w Open5GS

```
resources:
  requests:
    memory: "256Mi"
    cpu: "200m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

Niniejsza konfiguracja jest traktowana jako punkt operacyjny (punkt pracy), który powinien być wystarczający w normalnych warunkach (np. przez 80% czasu). Jednak w sytuacjach, gdy pojedynczy pod UPF sporadycznie przekracza te wartości, konieczne jest zwiększenie dostępnych zasobów oraz proporcjonalne podniesienie limitów.

Zakłada się, że każdorazowe przekroczenie wartości operacyjnych CPU lub pamięci skutkuje przydzieleniem dodatkowych 20% zasobów dla `requests` oraz ustawieniem `limits` na poziomie dwukrotności nowej wartości `requests`.

Przykładowa adaptacja zasobów

- Gdy zużycie CPU osiąga wartość 270m:
 - `requests` zostaje zwiększone do $270m \times 1.2 = 324m$,
 - `limits` zostaje zwiększone do $324m \times 2 = 648m$.
- Analogiczne zasady dotyczą przydziału pamięci operacyjnej.

Jeśli rzeczywiste wykorzystanie zasobów jest znacząco niższe od wartości operacyjnej, nie jest podejmowana żadna akcja korygująca. Nadmierne skalowanie w dół skutkowałoby zbyt częstymi restartami podów, co w konsekwencji generowałoby większe koszty niż wynikające z niedostatecznego wykorzystania zasobów.

5.3.2. Implementacja Agentów Translacji

Agent Ingress

Agent Ingress okresowo pobiera aktualne zużycie zasobów oraz zdefiniowane wartości `requests` oraz `limits` dla podów UPF w obu wdrożeniach (ang. *deployments*). Następnie, zgodnie ze specyfikacją **Interfejsu Lupin**, będzie aktualizował pole `Input` statusu obiektu API **Elementu Ingress** obiektem JSON, jak pokazano na listingu 16 z przykładowymi wartościami. Jednocześnie taki obiekt JSON będzie stanowił początkową postać **Danych**.

Listing 16.

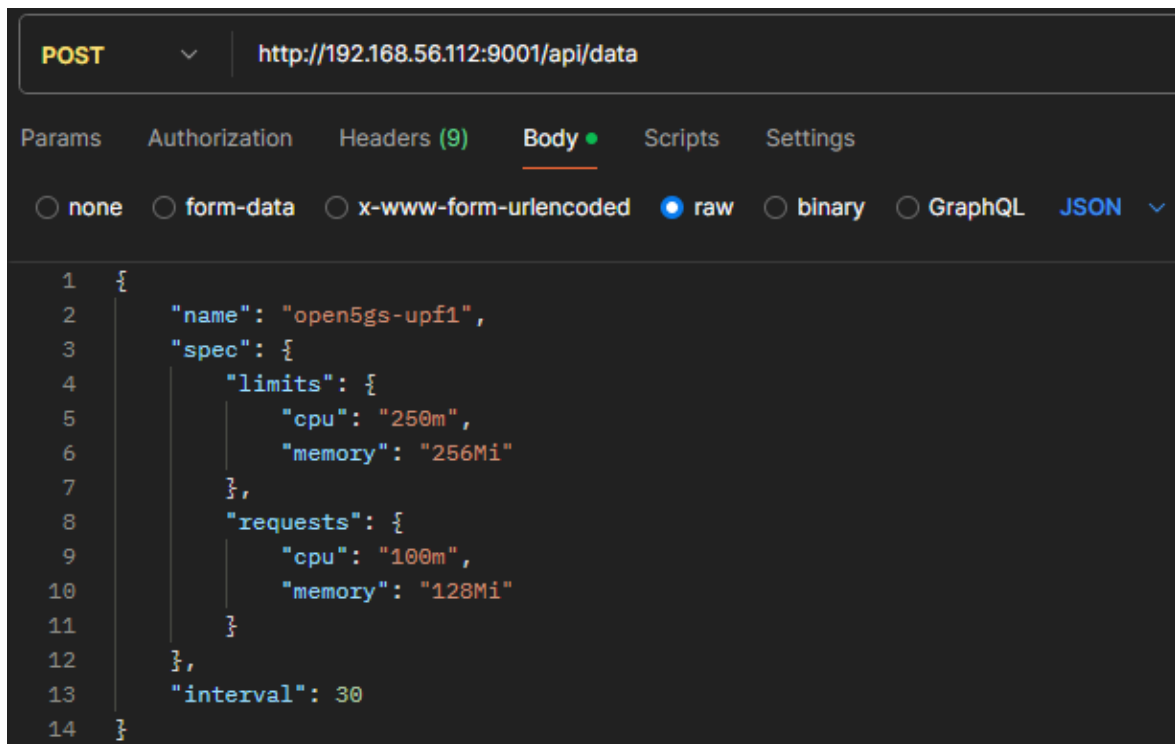
```
{
  "open5gs-upf1": {
    "actual": {
      "cpu": "1m",
      "memory": "18Mi"
    },
    "requests": {
      "cpu": "100m",
      "memory": "128Mi"
    },
    "limits": {
      "cpu": "250m",
      "memory": "256Mi"
    }
  },
  "open5gs-upf2": {
    "actual": {
      "cpu": "1m",
      "memory": "25Mi"
    },
    "requests": {
      "cpu": "200m",
      "memory": "256Mi"
    },
    "limits": {
      "cpu": "500m",
      "memory": "512Mi"
    }
  }
}
```

Kod agenta Ingress znajduje się w Załączniku 7.

Agent Egress

Agent Egress udostępnia endpoint HTTP, który przyjmuje dane w postaci przedstawionej na rysunku 5.5

Następnie dla wdrożenia (ang. *deployment*) zadanego polem name zmodyfikuje zdefiniowane wartości `requests` oraz `limits` według pola `spec`. Opcjonalne jest pole `interval`, które określa z jakim interwałem czasowym (w sekundach) raportować aktualne zużycie zasobów. Agent Egress odpowiednio wysteruje tym polem Agentu Ingress. Kod Agentu Egress znajduje się w Załączniku 8.



Rysunek 5.5. Endpoint Egress Agent przyjmujący finalne dane

5.3.3. Planowanie Logiki Pętli

Analizując dane wejściowe z listingu 16 można wyróżnić cztery możliwe stany pojedynczego wdrożenia UPF:

- **NORMAL** – wartości `requests` i `limits` ustawione na wartości domyślne, rzeczywiste zużycie (`actual`) pozostaje poniżej wartości domyślnych.
- **NORMAL_TO_CRITICAL** – wartości `requests` i `limits` nadal są domyślne, ale rzeczywiste zużycie (`actual`) przekracza wartości domyślne.
- **CRITICAL** – wartości `requests` i `limits` przekraczają wartości domyślne, a rzeczywiste zużycie (`actual`) również pozostaje powyżej wartości domyślnych.
- **CRITICAL_TO_NORMAL** – wartości `requests` i `limits` nadal przekraczają wartości domyślne, ale rzeczywiste zużycie (`actual`) spadło poniżej wartości domyślnych.

W zależności od stanu systemu należy podjąć następujące działania:

- **NORMAL** – brak działań, system działa poprawnie.
- **NORMAL_TO_CRITICAL** – dostosowanie wartości `requests` i `limits`, ustawienie interwału obserwacji na wysoki (HIGH).
- **CRITICAL** – dostosowanie wartości `requests` i `limits` do aktualnego zapotrzebowania.
- **CRITICAL_TO_NORMAL** – przywrócenie wartości `requests` i `limits` do wartości domyślnych, ustawienie interwału obserwacji na niski (LOW).

W związku z powyższym opisem pętla powinna w każdej iteracji określić punkt pracy, czyli stan w jakim znajduje się UPF. Jeśli jest to stan **NORMAL**, nie ma potrzeby aby wykonywać jakiegokolwiek akcji sterujących. W przeciwnym wypadku, należy "dostroić" UPF

odpowiednimi wartościami `requests` oraz `limits`. Jeśli UPF jest lub zbliża się do stanu krytycznego należy je ustawić z odpowiednim zapasem, zaś jeżeli UPF wraca już do stanu normalnego należy przywrócić im wartości domyślne. Na sam koniec, jeżeli mamy jeden ze stanów przejściowych (`NORMAL_TO_CRITICAL` bądź `CRITICAL_TO_NORMAL`) należy zmienić interwał obserwacji odpowiednio na wysoki lub niski.

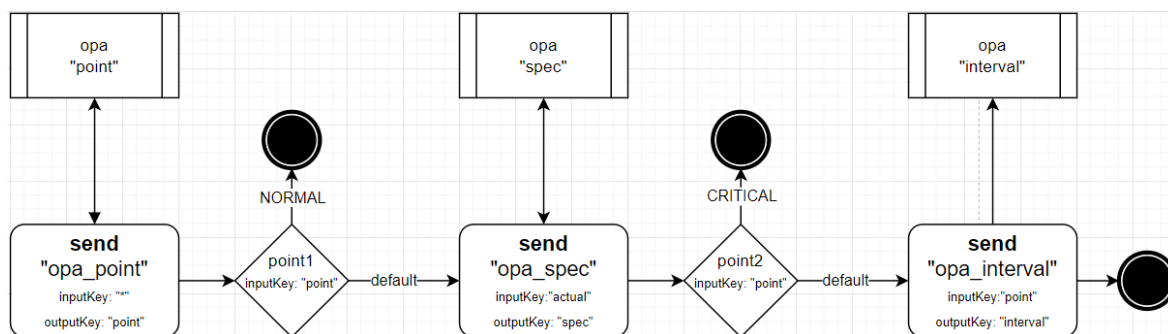
Z tego opisu widzimy, że **Część obliczeniowa** logiki pętli składa się z trzech części:

- Ustalenie punktu pracy
- Obliczenie wartości `requests` oraz `limits`
- Ustalenie wartości interwału na podstawie punkty pracy

Dlatego też powstaną trzy **Elementy Zewnętrzne**:

- **POINT** – akceptuje "*" (wszystkie **Pola Danych**), zwraca "point".
- **SPEC** – akceptuje "actual", zwraca "spec" zawierające wartości "requests" oraz "limits".
- **INTERVAL** – akceptuje "point", zwraca "interval".

A **Workflow Akcji** dla **Elementu Lupus** odpowiedzialnego za rekonyliację jednego wdrożenia będzie wyglądało jak na rysunku 5.6.



Rysunek 5.6. Workflow Akcji Elementu Lupus

5.3.4. Przygotowanie Elementów Zewnętrznych

Każdy z **Elementów Zewnętrznych** przygotowano jako aplikację Python emulującą serwer **Open Policy Agent**.

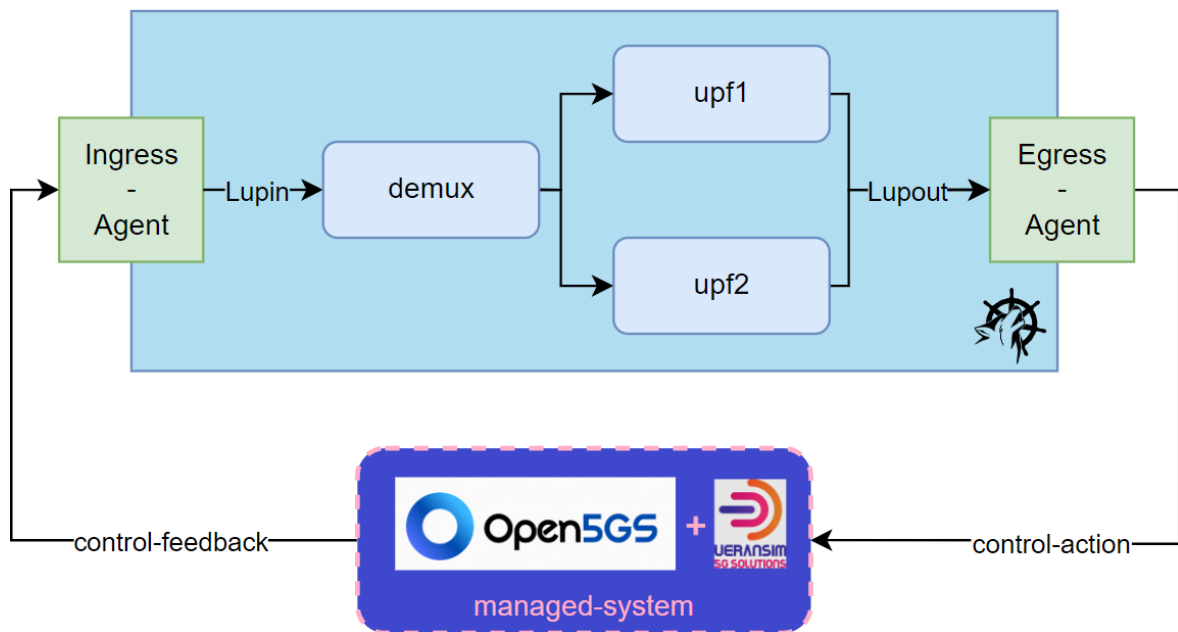
Implementacje można znaleźć w Załączniku 9.

5.3.5. Wyrażenie workflow pętli w notacji LupN

Workflow Pętli pokazano na rysunku 5.7.

Element demux służy do rozdzielenia **Danych** na dwie części. Każde wdrożenie UPF ma swój dedykowany element `upf1` lub `upf2`, który obsługuje je według **Workflow Akcji** przedstawionego w poprzedniej podsekcji. Elementy `upf1` oraz `upf2` niezależnie pobudzają endpoint `/api/data` **Agenta Egress**.

Kod **LupN** znajduje się w Załączniku 10. Ostatnim krokiem w celu uruchomienia pętli jest zaaplikowanie **Pliku LupN**.



Rysunek 5.7. Workflow Pętli

5.3.6. Prezentacja działania jednej iteracji pętli

Iterację rozpoczyna **Agent Ingress**, który pobiera **Dane** z Kubernetes (rys. 5.8). Następnie aktualizuje on Status obiektu API `lola-demux` (rys. 5.9), który wykonuje swoje **Workflow Akcji** (rys. 5.10), skutkujące aktualizacją Statusu obiektów API `upf1` i `upf2` (rys. 5.11). To skutkuje wywołaniem Operatora dla tych obiektów. Operator wykonuje dla obu **Workflow Akcji** (rys. 5.12). Na koniec oba elementy wywołują endpoint `api/data` serwowany przez **Agent Egress** (rys. 5.13).

```
ejek@open5gs-vm:~/lupus$ python3 examples/open5gs/sample-loop/ingress-agent.py --interval 30
* Serving Flask app 'ingress-agent'
* Debug mode: off
2025/01/30 17:02:02 Round: 1
{"open5gs-upf1": {"requests": {"cpu": "100m", "memory": "128Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"},
"actual": {"cpu": "1m", "memory": "19Mi"}}, "open5gs-upf2": {"requests": {"cpu": "100m", "memory": "128Mi"}, "li
mits": {"cpu": "250m", "memory": "256Mi"}, "actual": {"cpu": "1m", "memory": "19Mi"}}}
Updated Kubernetes custom resource status successfully.
```

Rysunek 5.8. Logi Agent Ingress

Gdy pobudzimy funkcję UPF do większej pracy ruchem z UE, widzimy raportowane większe zużycie zasobów przez odpowiednie wdrożenie (rys. 5.14). Aktualne zużycie przekraczające zdefiniowany punkt pracy powoduje otrzymanie od elementu zewnętrznego opa-point punkt `NORMAL_TO_CRITICAL`, co z kolei wpływa na workflow akcji (rys. 5.15). W kolejnych iteracjach w logach Agent Ingress widzimy błąd spowodowany tym, że Pod UPF1 chwilowo nie istnieje, stało się tak dlatego iż Agent Egress w poprzedniej iteracji zmienił jego wartość `requests` oraz `limits`, co poskutkowało restartem Poda. Po restarcie widzimy iż Agent Ingress raportuje nowe wartości `requests` i `limits` dla Poda UPF1.

Dokumentacja elektroniczna całego procesu wraz z instrukcją uruchomienia na własnym środowisku znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/tree/master/examples/open5gs>.

```

Status:
  Input:
    open5gs-upf1:
      Actual:
        Cpu:      1m
        Memory:   19Mi
      Limits:
        Cpu:      250m
        Memory:   256Mi
      Requests:
        Cpu:      100m
        Memory:   128Mi
    open5gs-upf2:
      Actual:
        Cpu:      1m
        Memory:   19Mi
      Limits:
        Cpu:      250m
        Memory:   256Mi
      Requests:
        Cpu:      100m
        Memory:   128Mi
  Last Updated: 2025-01-30T17:07:09.652335Z
  Events:      <none>

```

demux

Rysunek 5.9. Status obiektu API "lola-demux"

```

2025-01-30T17:09:12Z INFO ===== START OF lola-demux RECONCLIER:
{"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-demux", "namespace": "default"}, "namespace": "default", "name": "lola-demux", "reconcileID": "cf248c50-8c3b-43ae-97a0-4be35949d2c5"}
-----print-----Data:-----
{"open5gs-upf1":{"actual":{"cpu":"1m","memory":"19Mi"},"limits":{"cpu":"250m","memory":"256Mi"},"name":"open5gs-upf1","requests":{"cpu":"100m","memory":"128Mi"}}, "open5gs-upf2":{"actual":{"cpu":"1m","memory":"19Mi"},"limits":{"cpu":"250m","memory":"256Mi"},"name":"open5gs-upf2","requests":{"cpu":"100m","memory":"128Mi"}}}
2025-01-30T17:09:12Z INFO Success exit encountered {"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-demux", "namespace": "default"}, "namespace": "default", "name": "lola-demux", "reconcileID": "cf248c50-8c3b-43ae-97a0-4be35949d2c5"}
2025-01-30T17:09:12Z INFO Successfully sent final Data to Next element {"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-demux", "namespace": "default"}, "namespace": "default", "name": "lola-demux", "reconcileID": "cf248c50-8c3b-43ae-97a0-4be35949d2c5"}
2025-01-30T17:09:12Z INFO Element successfully reconciled {"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-demux", "namespace": "default"}, "namespace": "default", "name": "lola-demux", "reconcileID": "cf248c50-8c3b-43ae-97a0-4be35949d2c5", "name": "lola-demux"}
2025-01-30T17:09:12Z INFO ===== START OF lola-upf1 RECONCLIER:
{"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-upf1", "namespace": "default"}, "namespace": "default", "name": "lola-upf1", "reconcileID": "9bf0585c-8afb-4209-a3e8-8dc5795b43f4"}

```

Rysunek 5.10. Logi operatora element rekuncylującego obiekt API "lola-demux"

6. Podsumowanie

6.1. Wstęp

//TODO ze świeżą głową

- Wyzwania
- Analiza wyników w odniesieniu do założeń i celów pracy.
- Możliwe ograniczenia.
- Porównanie uzyskanych rezultatów z literaturą lub istniejącymi rozwiązaniami.
- Możliwości rozwoju i przyszłych badań.


```
Status:
Input:
  Actual:
    Cpu:      1m
    Memory:   19Mi
  Limits:
    Cpu:      250m
    Memory:   256Mi
  Name:      open5gs-upf1
  Requests:
    Cpu:      100m
    Memory:   128Mi
  Last Updated: 2025-01-30T17:07:40Z
Events:      <none>
```

upf1

```
Status:
Input:
  Actual:
    Cpu:      1m
    Memory:   19Mi
  Limits:
    Cpu:      250m
    Memory:   256Mi
  Name:      open5gs-upf2
  Requests:
    Cpu:      100m
    Memory:   128Mi
  Last Updated: 2025-01-30T17:07:40Z
Events:      <none>
```

upf2

Rysunek 5.11. Statusy obiektów API "lola-upf1" oraz "lola-upf2"

```
2025-01-30T17:09:12Z INFO ===== START OF lola-upf1 RECONCILER:
{"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-upf1", "namespace": "default", "namespace": "default", "name": "lola-upf1", "reconcileID": "9bf0585c-8afb-4209-a3e8-8dc5795b43f4"}}
-----print1-----Data:-----
{"actual":{"cpu":"1m","memory":"19Mi"},"limits":{"cpu":"250m","memory":"256Mi"},"name":"open5gs-upf1","requests":{"cpu":"100m","memory":"128Mi"}}
Sending {"input":{"actual":{"cpu":"1m","memory":"19Mi"},"limits":{"cpu":"250m","memory":"256Mi"},"name":"open5gs-upf1","requests":{"cpu":"100m","memory":"128Mi"}}} to POST http://192.168.56.112:9500/v1/data/policy/point
-----print2-----Data:-----
{"actual":{"cpu":"1m","memory":"19Mi"},"limits":{"cpu":"250m","memory":"256Mi"},"name":"open5gs-upf1","point":"NORMAL","requests":{"cpu":"100m","memory":"128Mi"}}
2025-01-30T17:09:12Z INFO Success exit encountered {"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-upf1", "namespace": "default", "namespace": "default", "name": "lola-upf1", "reconcileID": "9bf0585c-8afb-4209-a3e8-8dc5795b43f4"}}
2025-01-30T17:09:12Z INFO Element successfully reconciled {"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-upf1", "namespace": "default", "namespace": "default", "name": "lola-upf1", "reconcileID": "9bf0585c-8afb-4209-a3e8-8dc5795b43f4", "name": "lola-upf1"}}
2025-01-30T17:09:12Z INFO ===== START OF lola-upf2 RECONCILER:
{"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-upf2", "namespace": "default", "namespace": "default", "name": "lola-upf2", "reconcileID": "07d57cd1-3723-46e8-834c-b8be308237ca"}}
```

Rysunek 5.12. Logi operatora element rekoncyliującego obiekt API "lola-upf1"

```
192.168.56.112 - - [30/Jan/2025 17:10:14] "POST /api/data HTTP/1.1" 200 -
{"actual": {"cpu": "1m", "memory": "19Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5gs-upf1", "point": "NORMAL", "requests": {"cpu": "100m", "memory": "128Mi"}}
192.168.56.112 - - [30/Jan/2025 17:10:45] "POST /api/data HTTP/1.1" 200 -
{"actual": {"cpu": "1m", "memory": "19Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5gs-upf2", "point": "NORMAL", "requests": {"cpu": "100m", "memory": "128Mi"}}
192.168.56.112 - - [30/Jan/2025 17:10:45] "POST /api/data HTTP/1.1" 200 -
{"actual": {"cpu": "1m", "memory": "19Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5gs-upf1", "point": "NORMAL", "requests": {"cpu": "100m", "memory": "128Mi"}}
192.168.56.112 - - [30/Jan/2025 17:11:15] "POST /api/data HTTP/1.1" 200 -
{"actual": {"cpu": "1m", "memory": "19Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5gs-upf2", "point": "NORMAL", "requests": {"cpu": "100m", "memory": "128Mi"}}
192.168.56.112 - - [30/Jan/2025 17:11:15] "POST /api/data HTTP/1.1" 200 -
```

Rysunek 5.13. Logi Agenta Egress

```

ejek@open5gs-vm:~/lupus$ python3 examples/open5gs/sample-loop/ingress-agent.py --interval 30
* Serving Flask app 'ingress-agent'
* Debug mode: off
2025/01/30 18:36:46 Round: 1
{"open5gs-upf1": {"requests": {"cpu": "100m", "memory": "128Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"},
"actual": {"cpu": "0m", "memory": "19Mi"}, "open5gs-upf2": {"requests": {"cpu": "100m", "memory": "128Mi"}, "l
imits": {"cpu": "250m", "memory": "256Mi"}, "actual": {"cpu": "1m", "memory": "19Mi"}}}
Updated Kubernetes custom resource status successfully.
2025/01/30 18:37:19 Round: 2
{"open5gs-upf1": {"requests": {"cpu": "100m", "memory": "128Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"},
"actual": {"cpu": "51m", "memory": "19Mi"}, "open5gs-upf2": {"requests": {"cpu": "100m", "memory": "128Mi"}, "l
imits": {"cpu": "250m", "memory": "256Mi"}, "actual": {"cpu": "1m", "memory": "19Mi"}}}
Updated Kubernetes custom resource status successfully.
2025/01/30 18:37:52 Round: 3
{"open5gs-upf1": {"requests": {"cpu": "100m", "memory": "128Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"},
"actual": {"cpu": "108m", "memory": "19Mi"}, "open5gs-upf2": {"requests": {"cpu": "100m", "memory": "128Mi"}, "l
imits": {"cpu": "250m", "memory": "256Mi"}, "actual": {"cpu": "1m", "memory": "19Mi"}}}
Updated Kubernetes custom resource status successfully.
Error fetching actual usage for pod open5gs-upf1-855cf568bb-2np85: Error from server (NotFound): podmetrics.metr
ics.k8s.io "open5gs/open5gs-upf1-855cf568bb-2np85" not found
2025/01/30 18:38:24 Round: 4
{"open5gs-upf1": {"requests": {"cpu": "129m", "memory": "128Mi"}, "limits": {"cpu": "259m", "memory": "256Mi"},
"actual": {"cpu": "108m", "memory": "19Mi"}, "open5gs-upf2": {"requests": {"cpu": "100m", "memory": "128Mi"}, "l
imits": {"cpu": "250m", "memory": "256Mi"}, "actual": {"cpu": "1m", "memory": "19Mi"}}}
Updated Kubernetes custom resource status successfully.
2025/01/30 18:38:57 Round: 5
{"open5gs-upf1": {"requests": {"cpu": "129m", "memory": "128Mi"}, "limits": {"cpu": "259m", "memory": "256Mi"},
"actual": {"cpu": "114m", "memory": "17Mi"}, "open5gs-upf2": {"requests": {"cpu": "100m", "memory": "128Mi"}, "l
imits": {"cpu": "250m", "memory": "256Mi"}, "actual": {"cpu": "1m", "memory": "19Mi"}}}
Updated Kubernetes custom resource status successfully.

```

Rysunek 5.14. Logi Agenta Ingress przy pobudzeniu ruchem

```

2025-01-30T18:37:53Z INFO Element successfully reconciled {"controller": "element", "controllerGroup": "lup
us.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-demux", "namespace": "default", "namespace": "d
efault", "name": "lola-demux", "reconcileID": "8a78e89d-fd16-4f1a-87aa-f26ff67a9b5e", "name": "lola-demux"}
2025-01-30T18:37:53Z INFO ===== START OF lola-upf1 RECONCLIER:
{"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"
name": "lola-upf1", "namespace": "default", "namespace": "default", "name": "lola-upf1", "reconcileID": "6f3171e8-e
65e-4f0f-8b0c-cb8928fde1f0"}
-----print1-----Data:-----
{"actual": {"cpu": "108m", "memory": "19Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5gs-upf1", "requests
": {"cpu": "100m", "memory": "128Mi"}}
Sending {"input": {"actual": {"cpu": "108m", "memory": "19Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5g
s-upf1", "requests": {"cpu": "100m", "memory": "128Mi"}}} to POST http://192.168.56.112:9500/v1/data/policy/point
-----print2-----Data:-----
{"actual": {"cpu": "108m", "memory": "19Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5gs-upf1", "point": "
NORMAL_TO_CRITICAL", "requests": {"cpu": "100m", "memory": "128Mi"}, "spec": {"limits": {"cpu": "259m", "memory": "256Mi"}, "
requests": {"cpu": "129m", "memory": "128Mi"}}}
Sending {"input": {"cpu": "108m", "memory": "19Mi"}} to POST http://192.168.56.112:9500/v1/data/policy/spec
-----print3-----Data:-----
{"actual": {"cpu": "108m", "memory": "19Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5gs-upf1", "point": "
NORMAL_TO_CRITICAL", "requests": {"cpu": "100m", "memory": "128Mi"}, "spec": {"limits": {"cpu": "259m", "memory": "256Mi"}, "
requests": {"cpu": "129m", "memory": "128Mi"}}}
-----print4-----Data:-----
{"actual": {"cpu": "108m", "memory": "19Mi"}, "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5gs-upf1", "point": "
NORMAL_TO_CRITICAL", "requests": {"cpu": "100m", "memory": "128Mi"}, "spec": {"limits": {"cpu": "259m", "memory": "256Mi"}, "
requests": {"cpu": "129m", "memory": "128Mi"}}}
Sending {"input": {"NORMAL_TO_CRITICAL"}} to POST http://192.168.56.112:9500/v1/data/policy/interval
-----print5-----Data:-----
{"actual": {"cpu": "108m", "memory": "19Mi"}, "interval": "HIGH", "limits": {"cpu": "250m", "memory": "256Mi"}, "name": "open5
gs-upf1", "point": "NORMAL_TO_CRITICAL", "requests": {"cpu": "100m", "memory": "128Mi"}, "spec": {"limits": {"cpu": "259m", "
memory": "256Mi"}, "requests": {"cpu": "129m", "memory": "128Mi"}}}
2025-01-30T18:37:53Z INFO Success exit encountered {"controller": "element", "controllerGroup": "lup
us.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-upf1", "namespace": "default", "namespace": "d
efault", "name": "lola-upf1", "reconcileID": "6f3171e8-e65e-4f0f-8b0c-cb8928fde1f0"}
Sending {"actual": {"cpu": "108m", "memory": "19Mi"}, "interval": "HIGH", "limits": {"cpu": "250m", "memory": "256Mi"}, "name
": "open5gs-upf1", "point": "NORMAL_TO_CRITICAL", "requests": {"cpu": "100m", "memory": "128Mi"}, "spec": {"limits": {"cpu":
"259m", "memory": "256Mi"}, "requests": {"cpu": "129m", "memory": "128Mi"}}} to POST http://192.168.56.112:9001/api/data
2025-01-30T18:37:53Z INFO Successfully sent final Data to Next element {"controller": "element", "contro
llerGroup": "Lupus.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-upf1", "namespace": "default", "
namespace": "default", "name": "lola-upf1", "reconcileID": "6f3171e8-e65e-4f0f-8b0c-cb8928fde1f0"}
2025-01-30T18:37:53Z INFO Element successfully reconciled {"controller": "element", "controllerGroup": "lup
us.gawor.io", "controllerKind": "Element", "Element": {"name": "lola-upf1", "namespace": "default", "namespace": "d
efault", "name": "lola-upf1", "reconcileID": "6f3171e8-e65e-4f0f-8b0c-cb8928fde1f0", "name": "lola-upf1"}
2025-01-30T18:37:53Z INFO ===== START OF lola-upf2 RECONCLIER:
{"controller": "element", "controllerGroup": "lupus.gawor.io", "controllerKind": "Element", "Element": {"
name": "lola-upf2", "namespace": "default", "namespace": "default", "name": "lola-upf2", "reconcileID": "f4973e77-f

```

Rysunek 5.15. Logi z workflow akcji elementu "upf1" przy pobudzeniu ruchem

Bibliografia

- [1] ETSI GR ENI 017 - "Experiental Networked Intelligence (ENI); Overview of Prominent Closed Control Loops Architectures", Dostęp zdalny (24.01.2025): https://www.etsi.org/deliver/etsi_gr/ENI/001_099/017/02.02.01_60/gr_ENI017v020201p.pdf, 2024.
- [2] J. O. Kephart i D. M. Chess, „The Vision of Autonomic Computing”, *IEEE Computer*, t. 36, nr. 1, s. 41–50, 2003. DOI: 10.1109/MC.2003.1160055. adr.: https://www.researchgate.net/publication/2955831_The_Vision_Of_Autonomic_Computing.
- [3] M. Minsky, *The Society of Mind*. New York, United States: Simon & Schuster, 1986, ISBN: 978-0671657130.
- [4] D. P. Doyle, „Architecture Evolution for Automation and Network Programmability”, *Ericsson Review*, 2014. adr.: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7006f5e8c7b83f2ea265a6155ff9b4cfe11d4011>.
- [5] N. Alliance, „Automation and Autonomous System Architecture Framework”, spraw. tech. Version 1.0, 2022. adr.: <https://google.com>.
- [6] C. Benzaid i T. Taleb, „AI-Driven Zero Touch Network and Service Management in 5G and Beyond: Challenges and Research Directions”, *IEEE Network*, 2020. DOI: 10.1109/MNET.001.1900252. adr.: <https://google.com>.
- [7] L. Fallon, J. Keeney i R. K. Verma, „Autonomic Closed Control Loops for Management, an Idea Whose Time Has Come?”, w *Conference on Network and Service Management (CNSM)*, 2019. DOI: 10.23919/CNSM46954.2019.9012687. adr.: <https://google.com>.
- [8] ONAP, „Open Network Automation Platform (ONAP) Architecture White Paper”, spraw. tech., 2018. adr.: <https://google.com>.
- [9] ETSI, „Zero-touch Network and Service Management (ZSM); Reference Architecture”, ETSI, spraw. tech. ETSI GS ZSM 002 V1.1.1, 2018. adr.: <https://google.com>.
- [10] J. R. Boyd, *The Essence of Winning and Losing*, Edited by Chet Richards and Chuck Spinney, 1995. adr.: https://slightlyeastofnew.com/wp-content/uploads/2010/03/essence_of_winning_losing.pdf.

Wykaz symboli i skrótów

Architektura – Zbiór reguł i metod opisujących funkcjonalność, organizację oraz implementację systemu.

Dane – Fakty oraz statystyk zebrane razem w celu analizy. Stanowią podstawę do wydobywania z nich Informacji

Definicje Zasobów Własnych (ang. *Custom Resource Definitions*) – Pliki manifestacyjne YAML służące do zarejestrowania w klastrze nowe zasoby własne

Doświadczenie (ang. *Experience*) – Proces zdobywania wiedzy przez system podczas funkcjonowania w środowisku docelowym .

Edukacja (ang. *Education*) – Proces zdobywania wiedzy przez system poza środowiskiem docelowym, np. podczas trenowania modelu .

Wiedza (ang. *Knowledge*) – Zestaw wzorców, które są wykorzystywane do wyjaśniania, a także przewidywania tego, co się wydarzyło, dzieje lub może się wydarzyć w przyszłości. Bazuje na danych, informacjach oraz umiejętnościach zdobytych poprzez doświadczenie oraz edukację

ENI – Experiential Networked Intelligence

ETSI – European Telecommunications Standards Institute

Reprezentacja pojęć będących przedmiotem zainteresowania środowiska w formie niezależnej od formy danych

Kognitywność (ang. *cognition*) – Proces rozumienia danych oraz informacji w celu produkcji nowych danych, informacji oraz wiedzy

Kognitywny System Zarządzania Siecią – System zarządzania siecią, który jest jednocześnie kognitywny. Docelowy system specyfikowany przez ENI

Kontroler (ang. *controller*) – Proces monitorujący stan obiektów API w klastrze i doprowadzający je do stanu pożądanego

Obiekt API (ang. *API object*) – Zasób reprezentujący element klastra, np. Poda, Deployment, Service. Jest przechowywany w etcd i zarządzany przez API Server. Jest to konkretna instancja danego typu (ang. *Kind*) zasobów

Plik Manifestowy YAML (ang. *YAML Manifest File*) – Plik konfiguracyjny zapisany w formacie YAML, definiujący obiekty API Kubernetes. Służy do deklaratywnego zarządzania zasobami klastra

Polityka (ang. *Policy*) – Zestaw reguł, który jest używany do zarządzania i kontrolowania zmiany i/lub utrzymania stanu jednego lub więcej zarządzanych obiektów

Regulator (ang. *Control System*) – Pojęcie z teorii sterowania. System, który reguluje pracę obiektu

Sterowany danymi (ang. *Data-Driven*) – Nie narzucający żadnej logiki pętli – logika jest interpretowana na podstawie danych.

System ENI – Centralny punkt architektury ENI. Odpowiedzialny za zamkniętą pętlę sterowania, za pomocą której ENI chce osiągnąć kognitywność. Lupus aspiruje do tego, aby zaproponować architekturę dla Systemu ENI na Kubernetes

System kognitywny – System, który uczy się, wnioskuje oraz podejmuje decyzje w sposób przypominający ludzki umysł

Uczenie maszynowe (ang. *Machine Learning*) – proces, który zdobywa nową wiedzę i/lub aktualizuje istniejącą wiedzę w celu optymalizacji funkcjonowania systemu przy użyciu przykładowych obserwacji

Świadomość kontekstu (ang. *Context Aware*) – Posiadanie przez system informacji oraz wiedzy, które opisują środowisko w jakim znajduje się dana jednostka w celu lepszego doboru wzorca do rozwiązania danego problemu

Wzorzec (ang. *pattern*) – Generyczne, reużywalne rozwiązanie danego problemu. Część składowa wiedzy .

Warstwa Sterowania (ang. *Control Plane*) – Zestaw komponentów zarządzających stanem klastra Kubernetes. W szczególności *Controller Manager*, który nadzoruje kontrolery zasobów

Workflow – Sekwencja połączonych węzłów, czasami zależnych warunkowo, która realizuje określony cel. Zazwyczaj workflow definiuje się w celu organizacji pracy.

Wnioskowanie (ang. *reasoning*) – Proces, w którym system wyciąga logiczne wnioski z dostępnych danych i wiedzy

Wzorzec Operator (ang. *Operator Pattern*) – Rozszerzenie Kubernetes pozwalające rozwijać kontrolery dla zasobów własnych

Zasoby Własne (ang. *Custom Resources*) – Mechanizm rozszerzenia klastra Kubernetes o nowe typy obiektów API

Spis rysunków

2.1	Pracowniczkę warszawskiej centrali telefonicznej z końca lat 20. XX wieku . . .	9
2.2	Dwa stany systemu: sprzed, oraz po realizacji zgłoszenia o treści: "D" chce zadzwonić do "F"	10
2.3	Architektura autonomicznych systemów IBM	12
2.4	Zestawienie linii czasu rozwoju systemów zarządzania siecią oraz pętli sterowania	14
2.5	Architektura referencyjna ZSM	17
2.6	Mapowanie pomiędzy elementami składowymi architektury ZSM a zamkniętą pętlą sterowania	18
2.7	Fazy cyklu życia oraz aktywności zamkniętej pętli sterowania	19
2.8	Koncepcja wyniesienia autonomicznych menadżerów na wspólną platformę (//TODO)	20
3.1	Architektura referencyjna dla Lupus	23
3.2	Architektura referencyjna z agentami translacji	24
3.3	Przykładowe workflow pętli	25
3.4	Przykładowe workflow akcji	26
4.1	Architektura Kubernetes	29
4.2	Flow pracy kontrolera	30

4.3 Zaznaczenie elementów dających się zaprogramować podczas flow pracy kontrolera	31
4.4 Komunikacja pomiędzy Elementami Lupus	31
5.1 Architektura Open5GS	40
5.2 Architektura UERANSIM	41
5.3 Stan podów w klastrze	41
5.4 Wdrożenia Open5GS, w tym instancje UPF	42
5.5 Endpoint Egress Agent przyjmujący finalne dane	45
5.6 Workflow Akcji Elementu Lupus	46
5.7 Workflow Pętli	47
5.8 Logi Agentu Ingress	47
5.9 Status obiektu API "lola-demux"	48
5.10 Logi operatora element rekoncyliującego obiekt API "lola-demux"	48
5.11 Statusy obiektów API "lola-upf1" oraz "lola-upf2"	49
5.12 Logi operatora element rekoncyliującego obiekt API "lola-upf1"	49
5.13 Logi Agentu Egress	49
5.14 Logi Agentu Ingress przy pobudzeniu ruchem	50
5.15 Logi z workflow akcji elementu "upf1" przy pobudzeniu ruchem	50

Spis tabel

4.1 Porównanie polimorfizmu przez wskaźniki i przez interfejsy w Go.	36
--	----

Spis załączników

1. Definicje Lupus	55
2. Instalacja Lupus	57
3. Specyfikacja notacji LupN	58
4. Specyfikacja interfejsów Lupus	66
5. Specyfikacja obiektu danych	67
6. Specyfikacja akcji	68
7. Kod Agentu Ingress	71
8. Kod Agentu Egress	73
9. Kod Elementów Zewnętrznych	75
10. Kod LupN	78

Załącznik 1. Definicje Lupus

Specyfikacja w formie elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/docs/defs.md>.

Akcja (ang. *Action*) – Obiekt LupN. Za jej pomocą możliwe są operacje na danych

Akcja Sterująca (ang. *Control Action*) – Akcja wywnioskowana przez System Sterowania i wykonywana na Systemie Zarządzanym przez Agentą Egress. Akcja Sterowania ma za zadanie przybliżyć Stan Aktualny do Stanu Pożądanego.

Agent Translacji (ang. *Translation Agent*) – Agent oprogramowania na styku Lupus oraz Systemu Zarządzanego pełniący rolę integratora. Termin zbiorczy na Agent Ingress i Agent Egress.

Agent Egress (ang. *Egress Agent*) – Jeden z agentów translacji umieszczony na wyjściu z Lupus. Przekazuje finalne dane reprezentujące akcję sterowania do systemu zarządzanego. Implementuje interfejs Lupout.

Agent Ingress (ang. *Ingress Agent*) – Jeden z agentów translacji umieszczony na wejściu do Lupus. Przekazuje stan aktualny zarządzanego systemu do Lupus. Implementuje interfejs Lupin.

Część Obliczeniowa (ang. *Computing Part*) – Część logiki pętli odpowiedzialna za obliczenia. Wykonywana jest przez elementy zewnętrzne, nie elementy Lupus.

Dane (ang. *Data*) – Nośnik informacji w jednej iteracji pętli. Informacje zapisane są w formacie JSON.

Destynacja (ang. *Destination*) – Referencja do Elementu Zewnętrznego. Jednoznacznie opisuje żądanie HTTP, jakie ma wykonać Operator Zasobu Element.

Element (ang. *Element*) – Typ (ang. *kind*) obiektu API stanowiącego Element Lupus.

Element Egress (ang. *Egress Element*) – Ostatni element w danej ścieżce workflow pętli. Implementuje interfejs Lupout.

Element Ingres (ang. *Ingress Element*) – Pierwszy element w workflow pętli. Implementuje interfejs Lupin.

Element Lupus (ang. *Lupus Element*) – Jeden z elementów pętli. Działa w warstwie sterowania Kubernetes. Jego celem jest wyrażenie workflow pętli oraz delegacja części obliczeniowej do elementów zewnętrznych.

Element Pętli (ang. *Loop Element*) – Element realizujący jednostkę logiki pętli. Termin zbiorczy na Element Lupus oraz Element Zewnętrzny.

Element Zewnętrzny (ang. *External Element*) – Jeden z elementów pętli. Działa poza klastrem Kubernetes. Jego celem jest część obliczeniowa logiki pętli.

Finalne dane (ang. *Final Data*) – Postać danych po wykonaniu ostatniej akcji z workflow akcji Elementu Egress. To ona definiuje Akcję Sterującą.

Funkcje użytkownika (ang. *User Functions*) – Alternatywa dla wdrażania serwerów HTTP do prostych obliczeń. Funkcje użytkownika działają jak elementy zewnętrzne, ale są wykonywane w klastrze Kubernetes. Jest to jedna z dostępnych Destynacji w Lupin.

Interfejs Lupin (ang. *Lupin interface*) – Interfejs wejściowy do Lupus. Implementować go musi Agent Ingress oraz Element Ingress.

Interfejs Lupout (ang. *Lupout interface*) – Interfejs wyjściowy z Lupus. Implementować go musi Agent Egress oraz Element Egress.

Interfejsy Lupus (ang. *Lupus interfaces*) – Termin zbiorczy na Interfejs Lupin oraz Interfejs Lupout.

Kod LupN (ang. *LupN code*) – Kod wyrażający notację LupN. Zbiór obiektów LupN zapisanych w YAML. Zapisany jest w pliku LupN.

Logika Pętli (ang. *Loop logic*) – Kroki, które muszą zostać wykonane, aby w każdej iteracji pętli przybliżyć Stan Aktualny do Stanu Pożądanego. Instrukcje określające, jak na podstawie reprezentacji Stanu Aktualnego ma zostać *wywnioskowana* Akcja Sterująca.

LupN (ang. *LupN*) – Notacja do wyrażania workflow pętli za pomocą obiektów LupN. Zapisywana w pliku LupN.

Lupus (ang. *Lupus*) – System o proponowanej w pracy architekturze, który pełni rolę Systemu Sterowania.

Lupus Master (ang. *Lupus Master*) – Nadzorca pojedynczej pętli Lupus. Odpowiedzialny za kreację obiektów Zasobu Lupus Element.

Master (ang. *Master*) – Typ (ang. *kind*) obiektu API dla Lupus Master.

Open Policy Agent (ang. *Open Policy Agent*) – Serwer HTTP dedykowany do definiowania polityk i stosowania ich w systemach informatycznych.

Operator Zasobu Master (ang. *Master Operator*) – Kontroler zasobu Master. Odpowiedzialny za interpretację workflow pętli notacji LupN i tworzenie obiektów Element.

Obiekt LupN (ang. *LupN Object*) – Obiekty składające się na składnię LupN za pomocą, których wyrażane jest workflow pętli

Operator Zasobu Element (ang. *Element Operator*) – Kontroler zasobu Element. Odpowiedzialny za interpretację workflow akcji notacji LupN i jego wykonanie.

Plik LupN (ang. *LupN file*) – Plik manifestowy YAML zasobu Master. Zapisana jest w nim notacja LupN.

Pole Danych (ang. *Data Field*) – Jedno pole JSON w danych. Dostępne pod kluczem. Przekazywane jako wejście/wyjście akcji.

Problem Zarządzania (ang. *Management Problem*) – Problem występujący w Systemie Zarządzanym, który System Sterowania ma za cel rozwiązać.

Projektant (ang. *Designer*) – Jednostka użytkownika odpowiedzialna jedynie za projekt pętli. Nie musi posiadać umiejętności technicznych, a jedynie posługiwać się notacją LupN.

Sprężenie Zwrotne (ang. *Control Feedback*) – Reprezentacja Stanu Aktualnego wysyłana przez (lub pobierana z) Systemu Zarządzanego.

Stan Aktualny (ang. *Current State*) – Obserwowany w danej chwili stan Systemu Zarządzanego. Kontrastuje ze Stanem Pożądanym.

Stan Pożądany (ang. *Desired State*) – Pożądany stan obserwowany w Systemie Zarządzanym, który System Sterowania próbuje osiągnąć. Zazwyczaj można go wywieść z Problemu Zarządzania.

System Sterowania (ang. *Control System*) – System odpowiedzialny za rozwiązanie

Problemu Zarządzania w Systemie Zarządzanym za pomocą Zamkniętej Pętli Sterowania. Lupus aspiruje do tej roli.

System Zarządzany (ang. *Managed System*) – Dowolny system w posiadaniu Użytkownika, w którym występuje Problem Zarządzania.

Użytkownik (ang. *User*) – Organizacja lub indywidualność, która ma za cel użyć Lupus w celu rozwiązania Problemu Zarządzania w swoim Systemie Zarządzanym. W zespole użytkownika potrzebne są kompetencje programistyczne (do wdrożenia agentów translacji oraz elementów zewnętrznych), a także wyróżnia się rolę Projektanta.

Wdrożenie Lupus (ang. *Lupus deployment, Lupus usage or Lupus application*) – Pojedyncze rozwiązanie Problemu Zarządzania przez Użytkownika w danym Systemie Zarządzanym.

Workflow Akcji (ang. *Actions Workflow*) – *Workflow* złożone z Akcji w pojedynczym Elementie Lupus. Interpretowane przez Operator Zasobu Element .

Workflow Pętli (ang. *Loop Workflow*) – *Workflow* złożone z Elementów Pętli. Musi realizować Logikę Pętli.

Zamknięta Pętla Sterowania (ang. *Closed Control Loop*) – Sposób, w jaki Lupus jako System Sterowania rozwiązuje Problem Zarządzania.

Załącznik 2. Instalacja Lupus

Specyfikacja w formie elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/docs/installation.md>.

2.1. Przedślowie

Instalacja **Lupus** wymaga umiejętności technicznych oraz podstawowej znajomości operacyjnej Kubernetes.

Lupus jest zaimplementowany jako projekt Kubebuilder¹⁶. Zalecanym sposobem instalacji **Lupus** jest sklonowanie tego repozytorium i przyjęcie roli dewelopera tego projektu.

Nie istnieje coś takiego jak instalacja **Lupus** (np. w systemie operacyjnym). Można zainstalować **Zasoby Własne** dla **Elementów Lupus** w klastrze Kubernetes i uruchomić dla nich **Kontrolery**. Niniejszy załącznik opisuje właśnie taki proces.

2.2. Wymagania wstępne

Użytkownik musi posiadać działający klaster Kubernetes. Może to być Minikube¹⁷, zainstalowany silnik kontenerów (ang. *container engine*) (np. Docker¹⁸) oraz język Go¹⁹.

2.2.1. Instalacja Kubebuilder

Instrukcja dostępna pod adresem: <https://book.kubebuilder.io/quick-start>.

¹⁶<https://book.kubebuilder.io>

¹⁷<https://minikube.sigs.k8s.io/docs/>

¹⁸<https://docs.docker.com>

¹⁹<https://go.dev>

2.3. Klonowanie repozytorium

Listing 17. Klonowanie repozytorium

```
git clone https://github.com/0x41gawor/lupus
cd lupus
```

2.4. Instalacja CRD w klastrze

To polecenie zastosuje **CRD** (pl. Definicje **Zasobów Własnych**) dla **Master** i **Element**, umożliwiając ich użycie.

Listing 18. Instalacja CRD

```
make install
```

2.5. Uruchomienie kontrolerów

To polecenie uruchomi *kontrolery* dla *zasobów własnych* **master** i **element**.

Listing 19. Uruchomienie kontrolerów

```
make run
```

Istnieje możliwość uruchomienia kontrolerów jako pody w klastrze Kubernetes. W tym celu użytkownik jest zaproszony do bliższego zapoznania się z platformą Kubebuilder. Dopóki **Użytkownik** jest pewien, że nie będzie dopisywał **Funkcji Użytkownika** nie jest to zalecane podejście.

Załącznik 3. Specyfikacja notacji LupN

Specyfikacja w formie elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/docs/spec/lupn.md>.

LupN (od ang. *loop* oraz *Notation*) to język/notacja służąca do wyrażania **Workflow Pętli**. Nie zawiera opisu **Części Obliczeniowej Logiki Pętli**. **Część Obliczeniowa** jest określona poza **Lupus**, w **Elementach Zewnętrznych**.

LupN specyfikuje:

- **Workflow Pętli**, czyli workflow **Elementów Lupus**,
- odniesienia do **Elementów Zewnętrznych**, wyrażone jako **Destynacje**,
- **Workflow Akcji** w ramach **Elementu Lupus**,
- odniesienie (lub odniesienia) do **Agenta Egress** jako **Destynacja**.

Jak można zauważyć, LupN wyraża **workflow** na dwóch poziomach: globalnym (czyli **Workflow Elementów Lupus**) oraz wewnątrz **Elementu Lupus** (czyli **Workflow Akcji**). Możliwości obu poziomów są do siebie zbliżone, ale ostatecznie różne. Ten dokument również omówi tę kwestię.

Z punktu widzenia implementacji **Plik LupN** to w rzeczywistości *YAML manifest file* dla *Zasobu Własnego Master*. Po zaaplikowaniu (ang. *apply*), **Operator Zasobu Master** uruchamia **Elementy Lupus**, które realizują wyrażone **Workflow Pętli**.

LupN wyraża **Workflow Pętli** poprzez specyfikację różnych obiektów w notacji YAML. Nazwijmy te obiekty **Obiektami LupN**. Załącznik 3 specyfikuje te obiekty oraz relacje między nimi. Wskazuje również, co oznacza użycie każdego z nich w kontekście **Workflow Pętli** i jak **Operator Zasobu Element** je interpretuje w czasie działania.

Okazuje się, że obiekty YAML w **YAML manifest files** są pochodnymi struktur Golang (typów Golang), dlatego możemy opisać **Obiekty LupN** na podstawie tych struktur Golang. Obiekty YAML w **Pliku LupN** są pochodnymi struktur Go, dlatego **Obiekty LupN** możemy opisać na ich podstawie.

Wymagane jest wcześniejsze zapoznanie się z YAML. Ten dokument nie obejmuje translacji dokonywanej przez Kubernetes między strukturami Go a reprezentacjami obiektów YAML. Serializacja jest wykonywana przez `controller-gen` i opisana w *Kubebuilder Book*. Tłumaczenie to można łatwo zaobserwować i nauczyć się, analizując przykłady zamieszczone w repozytorium projektu: <https://github.com/0x41gawor/lupus/tree/master/examples>

3.1. Możliwości LupN

Ze względu na różnice implementacyjne węzłów omówione w sekcji 4.3.5, możliwości **Workflow Pętli** oraz **Workflow Akcji** różnią się od siebie. Sekwencja wykonawcza elementów jest definiowana poprzez obiekty `next`. Każdy element definiuje listę następnych elementów. Zazwyczaj jest to jeden element. Możliwe są rozgałęzienia (ang. *forks*) czyli lista z większą ilością elementów, ale wtedy zostaje wywołane wiele elementów niezależnie. Nie ma możliwości na zsynchronizowane powrotne złączenie przepływu danych. W przypadku akcji stosowany jest ten sam mechanizm opierający się na definiowaniu następnej akcji, z tym, że tutaj może być ona tylko jedna. Flow akcji interpretowane jest bowiem przez kontroler elementu, który procesuje je po jednej na raz. Z tego powodu możliwe jest sterowanie przepływem (ang. *flow control*), które zostało zaimplementowane jako specjalny typ akcji - `switch`.

3.2. Specyfikacja

Plik LupN posiada 4 główne pola (ang. *top-level fields*).

Listing 20. Główne pola pliku LupN

```
apiVersion: lupus.gawor.io/v1
kind: Master
metadata:
  labels:
    app.kubernetes.io/name: lupus
    app.kubernetes.io/managed-by: kustomize
name: lola
```

spec:

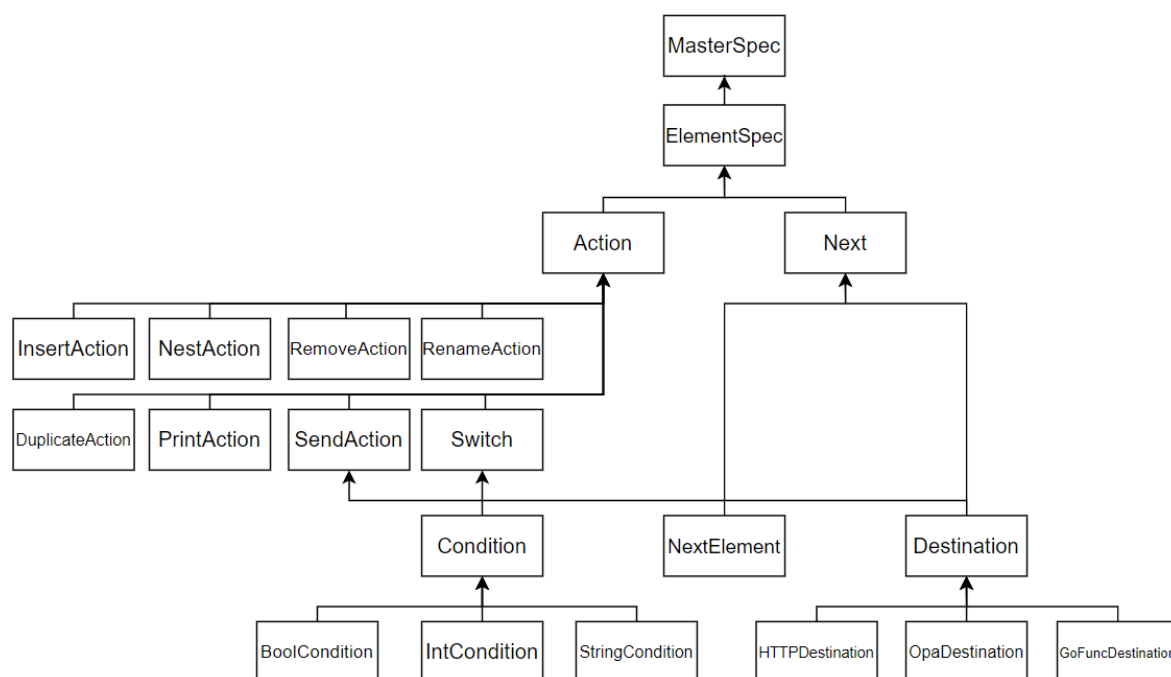
<lupn-objects>

Każde z nich musi być ustawione jak w 20 oprócz `metadata.name`, to pole odróżnia instancje pętli między sobą w obrębie klastra Kubernetes.

Notacja LupN rozpoczyna się od pola `spec`. Każdy obiekt Lupn zostanie opisany poprzez swoją definicję w Go.

3.2.1. Drzewo obiektów LupN

Podczas przeglądania specyfikacji **obiektów LupN** pomocne będzie śledzenie aktualnej pozycji w drzewie zależności obiektów.



Rysunek 3.1. Drzewko zależności obiektów LupN

3.2.2. MasterSpec

Listing 21. MasterSpec

```
// MasterSpec defines the desired state of Master
type MasterSpec struct {
    // Name of the Master CR (indicating the name of the loop)
    Name string `json:"name"`
    // Elements is a list of Lupus-Elements
    Elements []*ElementSpec `json:"elements"`
}
```

Każdy element na liście `Elements` spowoduje, że **Operator Zasobu Master** stworzy obiekt API typu **Lupus Element**.

3.2.3. ElementSpec

Listing 22. MasterSpec

```
// ElementSpec defines the desired state of Element
type ElementSpec struct {
    // Name is the name of the element, its distinct from Kubernetes API Object name,
    // but rather serves ease of management aspect for loop-designer
    Name string `json:"name"`
    // Descr is the description of the lupus-element, same as Name it serves as
    // the ease of management aspect for loop-designer
    Descr string `json:"descr"`
    // Actions is a list of Actions that lupus-element has to perform
    Actions []Action `json:"actions,omitempty"`
    // Next is a list of next objects (can be lupus-element or external-element)
    // to which send the final-data
    Next []Next `json:"next,omitempty"`
    // Name of master element (used as prefix for lupus-element name)
    Master string `json:"master,omitempty"`
}
```

3.2.4. Next

Listing 23. Next

```
// Next specifies the next loop-element in a loop workflow,
// it may be either lupus-element or reference to an external-element
// It allows to forward the whole final-data, but also parts of it
type Next struct {
    // Type specifies the type of next loop-element, lupus-element (element)
    // or external-element (destination)
    Type string `json:"type"␣kubebuilder:"validation:Enum=element,destination"`
    // List of input keys (Data fields) that have to be forwarded
    // Pass array with single element '*' to forward the whole input
    Keys []string `json:"keys"`
    // One of the fields below is not null
    Element *NextElement `json:"element,omitempty"␣kubebuilder:"validation:Optional"`
    Destination *Destination `json:"destination,omitempty"␣kubebuilder:"validation:Optional"`
}
```

3.2.5. NextElement

Listing 24. NextElement

```
// NextElement indicates the next loop-element
// in loop-workflow of type lupus-element
type NextElement struct {
    // Name is the lupus-name of lupus-element
    // (the one specified in Element struct)
    Name string `json:"name"`
}
```

3.2.6. Destination

Listing 25. Destination

```
// Destination represents an external-element
// It holds all the info needed to make a call to an external-element
// It supports calls to HTTP server, Open Policy Agent or user-functions
type Destination struct {
    // Type specifies if the external element is: a HTTP server in general,
    // a special kind of HTTP server like Open Policy Agent or internal, a user-function
    Type string `json:"type"␣kubebuilder:"validation:Enum=http;opa;gofunc"`
    // One of these fields is not null depending on a Type
    HTTP *HTTPDestination `json:"http,omitempty"␣kubebuilder:"validation:Optional"`
    Opa *OpaDestination `json:"opa,omitempty"␣kubebuilder:"validation:Optional"`
}
```

```

    GoFunc *GoFuncDestination 'json:"gofunc,omitempty"␣kubebuilder:"validation:Optional"'
}

```

3.2.7. HTTPDestination

Listing 26. HTTPDestination

```

// HTTPDestination defines fields specific to a HTTP type
// This is information needed to make a HTTP request
type HTTPDestination struct {
    // Path specifies HTTP URI
    Path string 'json:"path"'
    // Method specifies HTTP method
    Method string 'json:"method"'
}

```

3.2.8. OpaDestination

Listing 27. OpaDestination

```

// OpaDestination defines fields specific to Open Policy Agent type
// This is information needed to make an Open Policy Agent request
// Call to Opa is actually a special type of HTTP call
type OpaDestination struct {
    // Path specifies HTTP URI, since method is known
    Path string 'json:"path"'
}

```

3.2.9. GoFuncDestination

Listing 28. GoFuncDestination

```

// GoFuncDestination defines fields specific to GoFunc type
// This is information needed to call an user-function
type GoFuncDestination struct {
    // Name specifies the name of the function
    Name string 'json:"name"'
}

```

3.2.10. Action

Listing 29. Action

```

// Action represents operation that is performed on Data
// Action is used in Element spec. Element has a list of Actions
// and executes them in a workflow manner
// In general, each action has an input and output keys that define
// which Data fields it has to work on
// Each action indicates the name of the next Action in Action Chain
// There is special type - Switch. Actually, it does not perform any operation on Data,

```

```
// but rather controls the flow of Actions chain
type Action struct {
    // Name of the Action, it is for designer to ease the management of the Loop
    Name string `json:"name"`
    // Type of Action
    Type string `json:"type"␣kubebuilder:"validation:Enum=send,nest,remove,rename,duplicate,print,insert,switch"`
    // One of these fields is not null depending on a Type.
    Send      *SendAction      `json:"send,omitempty"␣kubebuilder:"validation:Optional"`
    Nest      *NestAction      `json:"nest,omitempty"␣kubebuilder:"validation:Optional"`
    Remove    *RemoveAction    `json:"remove,omitempty"␣kubebuilder:"validation:Optional"`
    Rename    *RenameAction    `json:"rename,omitempty"␣kubebuilder:"validation:Optional"`
    Duplicate *DuplicateAction `json:"duplicate,omitempty"␣kubebuilder:"validation:Optional"`
    Print     *PrintAction     `json:"print,omitempty"␣kubebuilder:"validation:Optional"`
    Insert    *InsertAction    `json:"insert,omitempty"␣kubebuilder:"validation:Optional"`
    Switch    *Switch          `json:"switch,omitempty"␣kubebuilder:"validation:Optional"`
    // Next is the name of the next action to execute, in the case of Switch-type action it stands as a default branch
    Next string `json:"next"`
}
```

Pole Next, oprócz nazw akcji, może przyjąć jedną z dwóch zdefiniowanych wartości. Wartość final oznacza, że postać **Danych** po tej akcji jest już w swojej **Finalnej Postaci** i musi zostać przekazana do następnego **Elementu Lupus**. Wartość exit oznacza nagłe zaniechanie aktualnej iteracji **Pętli Sterowania** (zazwyczaj wskutek błędu).

3.2.11. SendAction

Listing 30. SendAction

```
// SendAction is used to make call to external-element
// Element's controller obtains a data field using InputKey,
// and attaches it as a json body when performing a call to destination.
// Response is saved in data under an OutputKey
type SendAction struct {
    InputKey    string    `json:"inputKey"`
    Destination Destination `json:"destination"`
    OutputKey   string    `json:"outputKey"`
}
```

3.2.12. InsertAction

Listing 31. InsertAction

```
// InsertAction is used to make a new field and insert value to it
// Normally new fields are created as an outcome of other types of actions
// It is useful in debugging or logging,
// e.g. can indicate the path taken by the actions workflow
type InsertAction struct {
    OutputKey string    `json:"outputKey"`
    Value     runtime.RawExtension `json:"value"`
}
```

3.2.13. NestAction

Listing 32. NestAction

*// NestAction is used to group a number of data-fields together.
 // Element's controllers gather fields indicated by InputKeys list
 // and nest them in a new field under an OutputKey.*

```
type NestAction struct {
    InputKeys []string 'json:"inputKeys" '
    OutputKey string    'json:"outputKey" '
}
```

3.2.14. RemoveAction

Listing 33. RemoveAction

*// RemoveAction is used to delete a data-field.
 // Elements' controllers remove fields indicated by the list InputKeys*

```
type RemoveAction struct {
    InputKeys []string 'json:"inputKeys" '
}
```

3.2.15. RenameAction

Listing 34. RenameAction

*// RenameAction is used to change the name of a data-field.
 // InputKey indicates a field to be renamed
 // OutputKey is the new field name.*

```
type RenameAction struct {
    InputKey  string 'json:"inputKey" '
    OutputKey string 'json:"outputKey" '
}
```

3.2.16. DuplicateAction

Listing 35. DuplicateAction

*// DuplicateAction is used to make a copy of a data-field.
 // InputKey indicates the field of which value has to be copied.
 // OutputKey indicates the field to which values have to be pasted in.*

```
type DuplicateAction struct {
    InputKey  string 'json:"inputKey" '
    OutputKey string 'json:"outputKey" '
}
```


3.2.17. PrintAction

Listing 36. PrintAction

```
// PrintAction is used to print the value of each field  
// indicated by InputKeys in a controller's console.  
// It is useful in debugging or logging.  
type PrintAction struct {  
    InputKeys []string 'json:"inputKeys"'  
}
```

3.2.18. Switch

Listing 37. Switch

```
// Switch is a special type of action used for flow-control  
// When Element's controller encounters switch action on the chain  
// it emulates the work of a switch known in other programming languages  
type Switch struct {  
    Conditions []Condition 'json:"conditions"'  
}
```

3.2.19. Condition

Listing 38. Condition

```
// Condition represents a single condition present in Switch action  
// It defines on which Data field it has to be performed,  
// the actual condition to be evaluated,  
// and the next Action if evaluation returns true.  
type Condition struct {  
    // Key indicates the Data field that has to be retrieved  
    Key string 'json:"key"'  
    // Operator defines the comparison operation, e.g. eq, ne, gt, lt  
    Operator string 'json:"operator"␣kubebuilder:"validation:Enum=eq,ne,gt,lt"'  
    // Type specifies the type of the value: string, int, float, bool  
    Type string 'json:"type"␣kubebuilder:"validation:Enum=string,int,float,bool"'  
    // One of these fields is not null depending on a Type.  
    BoolCondition *BoolCondition 'json:"bool,omitempty"␣kubebuilder:"validation:Optional"'  
    IntCondition *IntCondition 'json:"int,omitempty"␣kubebuilder:"validation:Optional"'  
    StringCondition *StringCondition 'json:"string,omitempty"␣kubebuilder:"validation:Optional"'  
    // Next specifies the name of the next action to execute if evaluation returns true  
    Next string 'json:"next"'  
}
```

3.2.20. BoolCondition

Listing 39. BoolCondition

```
// BoolCondition defines a boolean-specific condition  
type BoolCondition struct {  
    Value bool 'json:"value"'  
}
```

3.2.21. IntCondition

Listing 40. IntCondition

```
// IntCondition defines an integer-specific condition
type IntCondition struct {
    Value int 'json:"value"'
}
```

3.2.22. StringCondition

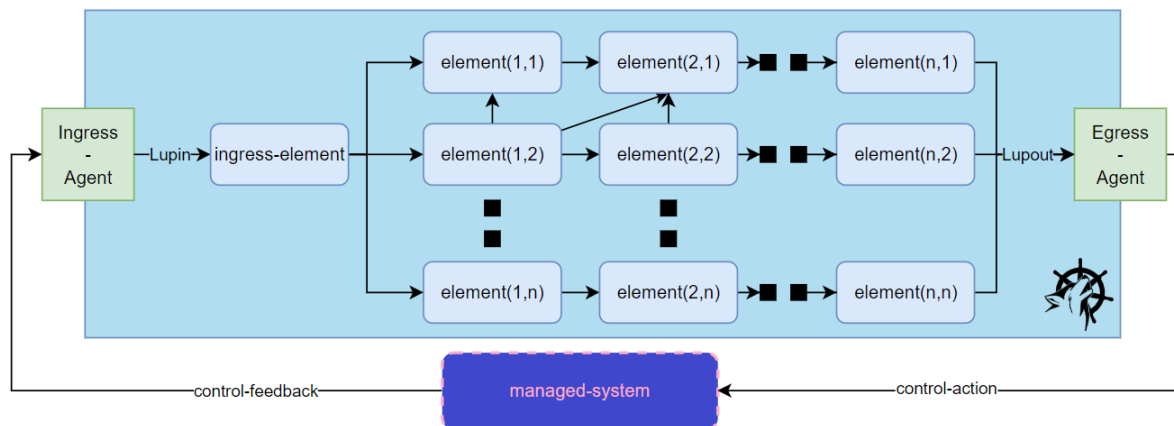
Listing 41. StringCondition

```
// StringCondition defines a string-specific condition
type StringCondition struct {
    Value string 'json:"value"'
}
```

Załącznik 4. Specyfikacja interfejsów Lupus

Specyfikacja w formie elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/docs/spec/lupin-lupout.md>.

4.1. Architektura



Rysunek 4.1. Architektura Lupus

4.2. Interfejs Lupin

Projektant może zdefiniować wiele **Elementów Lupus**, połączonych na różne, skomplikowane sposoby. Musi jednak zdecydować, który z nich zostanie wywołany przez **Agent Ingress**. Taki element można nazwać **Elementem Ingress**. **Lupus** zaleca posiadanie tylko jednego **Elementu Ingress**.

Jeśli **Agent Ingress** chce zasygnalizować, że można zaobserwować nowy stan **Systemu Zarządzanego** (co oznacza, że musi zostać uruchomiona nowa iteracja **Pętli Sterowania**), musi zmodyfikować pole `Status.Input` w *obiekcie API Elementu Ingress*. Wartość umieszczona w tym polu będzie reprezentować nowy **Aktualny Stan**.

Pole `Status.Input` w **Ingress Element CR** jest typu `RawExtension`, co oznacza, że podlega pod specyfikację **Danych** (Załącznik 5).

JSON przesłany w tym miejscu będzie stanowił **Dane** dla tego elementu.

Oprogramowanie implementuje interfejs `Lupin`, jeśli w pewnym miejscu swojego kodu wysła żądanie HTTP do **kube-api-server**, które aktualizuje status **Elementu Ingress**, a dokładniej pole `input`. Wartość musi być obiektem JSON, który reprezentuje **Aktualny Stan Systemu Zarządzanego**.

4.3. Interfejs Lupout

Punktem wyjścia z **Systemu Sterowania Lupus** jest ostatni **Element Lupus**, czyli **Element Egress**. Wysła on swoje **Finalne Dane** (lub ich część) do **Agentu Egress**. **Agent Egress** musi przekształcić to wejście w **Akcję Sterowania**, wykonywaną bezpośrednio na **Systemie Zarządzanym**.

Oprogramowanie implementuje interfejs `Lupout`, jeśli implementuje serwer HTTP, który akceptuje wejściowe **Dane** w formacie JSON i tłumaczy je na **Akcję Sterowania**, wykonywaną na **Systemie Zarządzanym**.

Załącznik 5. Specyfikacja obiektu danych

Specyfikacja w formie elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/docs/spec/data.md>.

Dane są kluczowym elementem spełnienia Wymagania 5. **Dane** to sposób, w jaki **Użytkownik**, podczas każdej iteracji, może:

- uzyskać informacje o **Aktualnym Stanie**,
- przechowywać pomocnicze informacje (takie jak odpowiedzi od **Elementów Zewnętrznych**),
- przechowywać informacje debugingowe,
- zapisywać informacje potrzebne do sformułowania **Akcji Sterowania**.

Dane są reprezentowane jako JSON dający się zapisać w strukturze `Go map[string] interface`. Nie mogą, więc być jedną z następujących form obiektu JSON:

- typem prymitywnym,
- tablicą,
- obiektem JSON z kluczami innymi niż `string`.

To, w jaki sposób można operować na **Danych**, prezentuje specyfikacja akcji (Załącznik 6).

Kluczowym pojęciem jest **Pole Danych**. Jest to pojedyncze pole JSON dostępne pod

danym kluczem. Jest jednostką operacyjną danych. Przekazywane jako wejście lub wyjście akcji.

Parametry kluczy wejściowych lub wyjściowych akcji akceptują znak *. Jego użycie wskazuje na przekazanie całego obiektu danych jako wejście/wyjście do/z akcji. Dodatkowo też klucze wejściowe/wyjściowe akcji wspierają zagnieżdżanie pól za pomocą operatora kropki (.).

Załącznik 6. Specyfikacja akcji

Specyfikacja w formie elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/docs/spec/actions.md>.

Specyfikacja **Akcji** w **LupN** znajduje się w Załączniku 3. Niniejszy dokument prezentuje przykładowe działanie **Akcji** na **Danych**.

Akcje zostały opracowane jako najbardziej atomowe operacje, które, gdy zostaną odpowiednio połączone, stanowią narzędzie umożliwiające **Projektantowi Pętli** pełne operowanie na **Danych**.

Czasami operacja, która na pierwszy rzut oka wydaje się atomowa, wymaga użycia dwóch połączonych **Akcji**. Z drugiej strony, zdarza się, że operacja początkowo uznana za atomową okazuje się jedynie szczególnym przypadkiem bardziej ogólnej operacji. Dobrym przykładem jest nieistniejąca już **Akcja** concat. Została ona zaprojektowana do łączenia dwóch pól w jedno, jednak okazało się, że jest to specyficzny przypadek **Akcji** nest, w której lista InputKey zawiera tylko dwa elementy.

6.1. Podział ogólny

Mamy 8 typów akcji:

- **Send**
- **Nest**
- **Remove**
- **Rename**
- **Duplicate**
- **Insert**
- **Print**
- **Switch**

Możemy wyróżnić następujące kategorie:

- 6 akcji, które mogą być używane do modyfikacji danych:
{Send, Nest, Remove, Rename, Duplicate, Insert}
- 1 akcja do komunikacji z **Elementami Zewnętrznymi**:
{Send}
- 2 akcje do debugowania:
{Insert, Print}

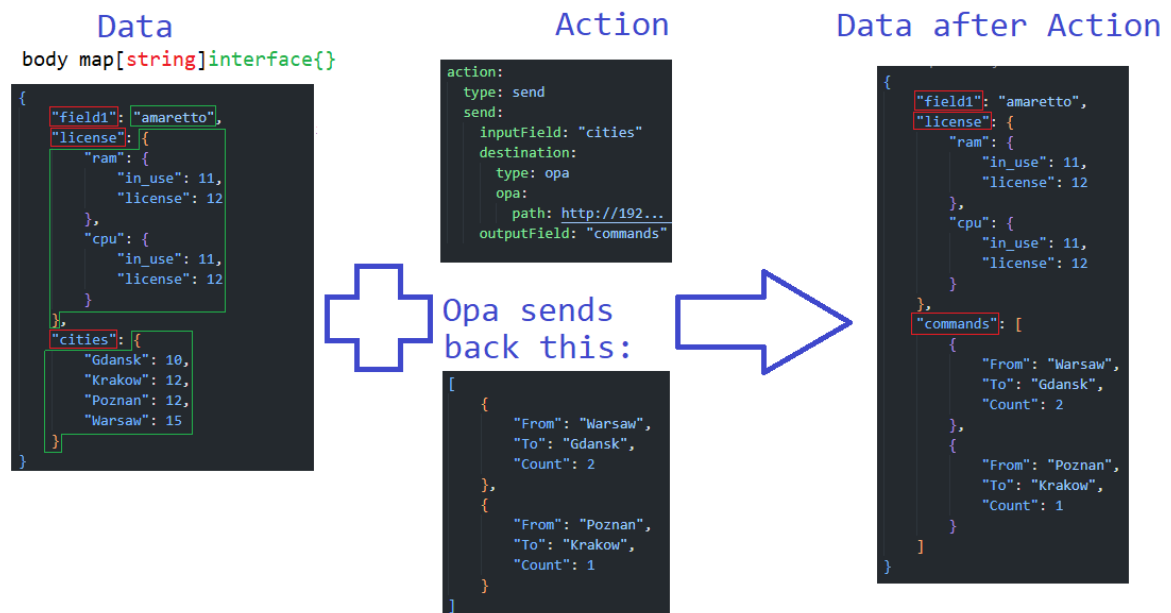
- 1 akcja do logowania:
{Print}
- 1 akcja do sterowania przepływem **workflow akcji**:
{Switch}

6.2. Przykłady

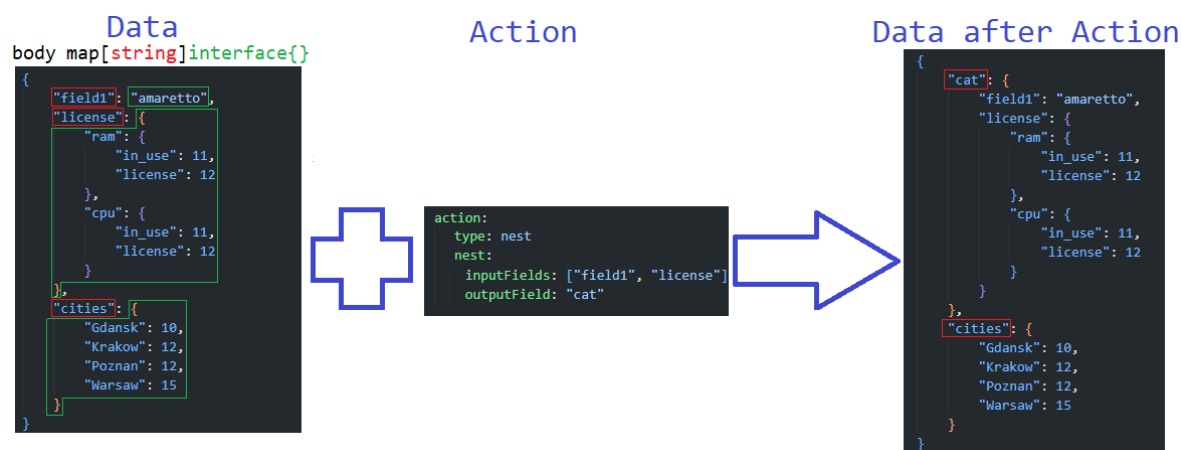
Ta sekcja przedstawi przykładowe użycie 6 akcji, które mogą modyfikować dane. Każdy przykład zawiera:

- reprezentację JSON stanu **danych** przed modyfikacją akcji,
- notację **LupN** zastosowanej akcji,
- reprezentację JSON stanu **danych** po modyfikacji akcji

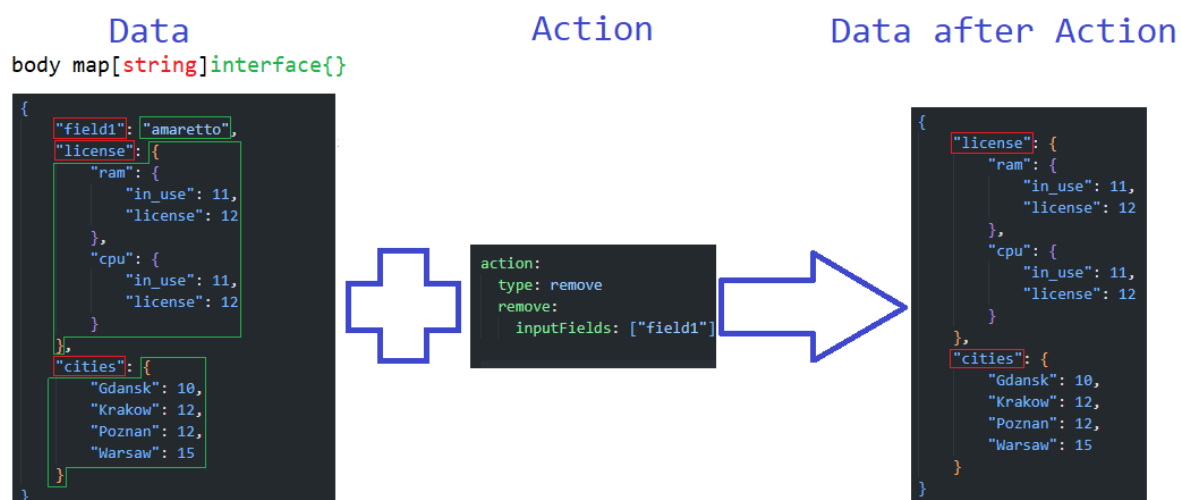
6.2.1. Send



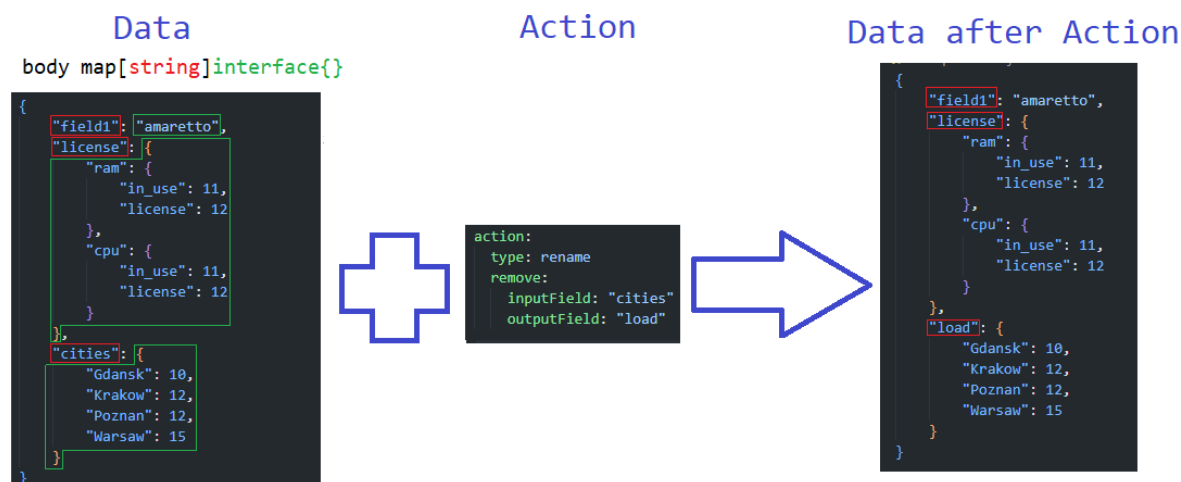
Rysunek 6.1. Przykład modyfikacji danych przez akcję Send



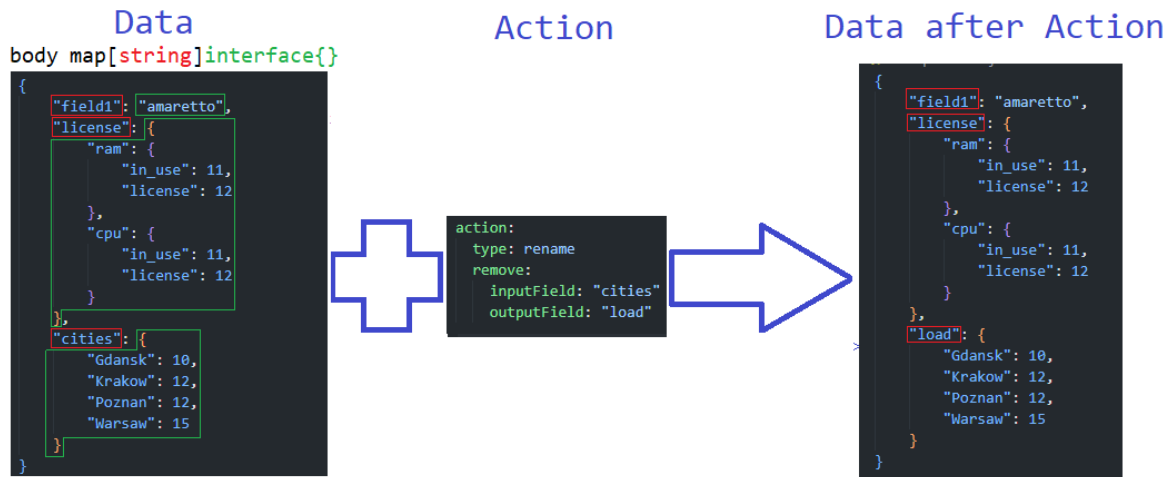
Rysunek 6.2. Przykład modyfikacji danych przez akcje Nest



Rysunek 6.3. Przykład modyfikacji danych przez akcje Remove



Rysunek 6.4. Przykład modyfikacji danych przez akcje Rename



Rysunek 6.5. Przykład modyfikacji danych przez akcje Duplicate

6.2.2. Nest

6.2.3. Remove

6.2.4. Rename

6.2.5. Duplicate

6.2.6. Insert

Załącznik 7. Kod Agenta Ingress

Kod w wersji elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/examples/open5gs/sample-loop/ingress-agent.py>

Listing 42. Kod Agenta Ingress

```
from kubernetes import client, config
import argparse
import time
import json
from datetime import datetime
import subprocess
from kubernetes.client import CustomObjectsApi
from flask import Flask, request, jsonify
from threading import Thread

# Load Kubernetes configuration
config.load_kube_config()

# Kubernetes API clients
core_v1_api = client.CoreV1Api()
apps_v1_api = client.AppsV1Api()

import logging
log = logging.getLogger('werkzeug')
log.setLevel(logging.ERROR)

global interval
round = 0

def get_pods_by_deployment_prefix(prefix):
    try:
        # List all deployments in the current namespace
        deployments = apps_v1_api.list_deployment_for_all_namespaces(watch=False)
```

```

    matching_deployments = [d for d in deployments.items if d.metadata.name.startswith(prefix)]
    # Find all pods associated with matching deployments
    pods = core_v1_api.list_pod_for_all_namespaces(watch=False)
    deployment_pod_map = {}
    for deployment in matching_deployments:
        filtered_pods = []
        for pod in pods.items:
            if pod.metadata.owner_references:
                for owner in pod.metadata.owner_references:
                    if owner.kind == "ReplicaSet" and owner.name.startswith(deployment.metadata.name):
                        filtered_pods.append(pod)
        deployment_pod_map[deployment.metadata.name] = filtered_pods

    return deployment_pod_map
except client.rest.ApiException as e:
    print(f"Exception when calling Kubernetes API: {e}")
    return {}

def get_pod_resources(pod):
    containers = pod.spec.containers
    resource_info = []
    for container in containers:
        resources = container.resources
        resource_info.append({
            "container_name": container.name,
            "requests": resources.requests or {},
            "limits": resources.limits or {},
        })
    return resource_info

def get_pod_actual_usage(pod_name, namespace):
    try:
        # Use kubectl top to fetch actual usage (requires metrics-server installed)
        result = subprocess.run(
            ["kubectl", "top", "pod", pod_name, "-n", namespace, "--no-headers"],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True
        )
        if result.returncode == 0:
            usage_data = result.stdout.strip().split()
            if len(usage_data) >= 3:
                return {
                    "cpu": usage_data[1],
                    "memory": usage_data[2],
                }
            else:
                print(f"Error fetching actual usage for pod {pod_name}: {result.stderr}")
                return {}
    except Exception as e:
        print(f"Error running kubectl top for pod {pod_name}: {e}")
        return {}

def get_json_data():
    deployment_prefix = "open5gs-upf"
    deployment_pod_map = get_pods_by_deployment_prefix(deployment_prefix)
    metrics = {}

    for deployment_name, pods in deployment_pod_map.items():
        # Initialize deployment entry in metrics
        metrics[deployment_name] = {
            "requests": {},
            "limits": {},
            "actual": {}
        }

    for pod in pods:
        pod_name = pod.metadata.name
        namespace = pod.metadata.namespace

        # Resource requests and limits
        resource_info = get_pod_resources(pod)
        for container_info in resource_info:
            metrics[deployment_name]["requests"] = container_info["requests"]
            metrics[deployment_name]["limits"] = container_info["limits"]

        # Actual resource usage
        actual_usage = get_pod_actual_usage(pod_name, namespace)
        metrics[deployment_name]["actual"] = actual_usage

    return json.dumps(metrics)

def send_to_kube(state):
    custom_objects_api = CustomObjectsApi() # Use CustomObjectsApi to interact with CRDs
    try:

```



```

    # Retrieve the custom resource
    observe = custom_objects_api.get_namespaced_custom_object(
        group="lupus.gawor.io",
        version="v1",
        namespace='default',
        plural="elements",
        name='lola-demux'
    )

    # Update the 'status.input' field with the state
    observe_status = observe.get('status', {})
    observe_status['input'] = json.loads(state) # Convert JSON string to an object
    observe_status['lastUpdated'] = datetime.utcnow().isoformat() + "Z" # Proper ISO 8601 format

    # Update the custom resource's status
    custom_objects_api.patch_namespaced_custom_object_status(
        group="lupus.gawor.io",
        version="v1",
        namespace='default',
        plural="elements",
        name='lola-demux',
        body={"status": observe_status} # Send only the 'status' field
    )
    print("Updated Kubernetes custom resource status successfully.")
except Exception as e:
    print(f"Error updating custom resource: {e}")

def periodic_task():
    json_data = get_json_data()
    timestamp = datetime.utcnow().strftime('%Y/%m/%d_%H:%M:%S')
    global round
    round = round + 1
    print(timestamp + "Round: " + str(round) + "\n" + json_data)
    send_to_kube(json_data)

app = Flask(__name__)

@app.route('/api/interval', methods=['POST'])
def update_interval():
    global interval
    try:
        data = request.get_json()
        if "value" in data and isinstance(data["value"], int) and data["value"] > 0:
            interval = data["value"]
            return jsonify({"message": "Interval updated successfully", "new_interval": interval}), 200
        else:
            return jsonify({"error": "Invalid input. 'value' must be a positive integer."}), 400
    except Exception as e:
        return jsonify({"error": str(e)}), 500

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Periodic K8s metrics fetcher')
    parser.add_argument('--interval', type=int, default=60, help='Interval in seconds for periodic task')
    args = parser.parse_args()

    # Declare that you are modifying the global variable
    interval = args.interval

    # Start Flask server in a separate thread
    flask_thread = Thread(target=lambda: app.run(host='0.0.0.0', port=9000, debug=False, use_reloader=False))
    flask_thread.daemon = True
    flask_thread.start()

    while True:
        periodic_task()
        time.sleep(interval)

```

Załącznik 8. Kod Agenta Egress

Kod w wersji elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/examples/open5gs/sample-loop/egress-agent.py>

Listing 43. Kod Agenta Egress

```

from flask import Flask, request, jsonify
from kubernetes import client, config
from kubernetes.client.rest import ApiException

app = Flask(__name__)

```

```

# Load Kubernetes config (use kubeconfig if running locally, in-cluster config if running in a cluster)
try:
    config.load_incluster_config()
except:
    config.load_kube_config()

# Function to patch deployment resources
def patch_deployment_resources(namespace, deployment_name, resource_type, cpu, memory):
    try:
        api_instance = client.AppsV1Api()

        # Define the patch
        patch = {
            "spec": {
                "template": {
                    "spec": {
                        "containers": [
                            {
                                "name": "upf",
                                "resources": {
                                    resource_type.split(".")[1]: { # Extract "requests" or "limits"
                                        "cpu": cpu,
                                        "memory": memory
                                    }
                                }
                            }
                        ]
                    }
                }
            }
        }

        # Patch the deployment
        api_response = api_instance.patch_namespaced_deployment(name=deployment_name, namespace=namespace, body=patch)
        return api_response

    except ApiException as e:
        return str(e)

import requests

def send_interval_request(url: str, value: int):
    headers = {
        "Content-Type": "application/json"
    }
    data = {
        "value": value
    }
    try:
        response = requests.post(url, json=data, headers=headers)
        response.raise_for_status() # Raise an exception for HTTP errors
        return response.json()
    except requests.RequestException as e:
        print(f"Request failed: {e}")
        return None

@app.route('/api/data', methods=['POST'])
def get_data():
    data = request.get_json()
    print(data)
    if "spec" in data:
        deployment_name = data.get('name')
        namespace = 'open5gs'
        lim_cpu = data['spec']['limits'].get('cpu')
        lim_ram = data['spec']['limits'].get('memory')
        req_cpu = data['spec']['requests'].get('cpu')
        req_ram = data['spec']['requests'].get('memory')
        res1 = patch_deployment_resources(namespace, deployment_name, 'resources.limits', lim_cpu, lim_ram)
        res2 = patch_deployment_resources(namespace, deployment_name, 'resources.requests', req_cpu, req_ram)

    if "interval" in data:
        interval = data['interval']
        response = send_interval_request("http://192.168.56.112:9000/api/interval", interval)

    return jsonify({"res": "ok"})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9001)

```

Załącznik 9. Kod Elementów Zewnętrznych

Kod w wersji elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/examples/open5gs/sample-loop/opa.py>

Listing 44. Kod Elementów Zewnętrznych

```
from flask import Flask, request, jsonify

app = Flask(__name__)

# Hardcoded default values
DEFAULT_VALUES = {
    "requests": {
        "memory": "128Mi",
        "cpu": "100m"
    },
    "limits": {
        "memory": "256Mi",
        "cpu": "250m"
    }
}

# ----- Parsing Helpers ----- #
def parse_cpu(cpu_str: str) -> int:
    """
    Convert a CPU string to an integer representing millicores.

    Examples:
    "300m" -> 300
    "2" -> 2000 (interpreted as 2 CPU cores -> 2000 millicores)
    "1.5" -> 1500
    """
    cpu_str = cpu_str.strip().lower()
    if cpu_str.endswith("m"):
        # e.g. "300m" -> 300
        numeric_part = cpu_str[:-1] # remove "m"
        return int(float(numeric_part))
    else:
        # e.g. "2" -> 2000, "1.5" -> 1500
        return int(float(cpu_str) * 1000)

def parse_memory(mem_str: str) -> int:
    """
    Convert a memory string to an integer representing megabytes (MB).

    Examples:
    "300Mi" -> 300
    "1Gi" -> 1024
    "1024Ki" -> 1
    "512" -> 512 (no unit, assume MB)
    """
    mem_str = mem_str.strip()
    lower_str = mem_str.lower()

    if lower_str.endswith("mi"):
        # e.g. "300Mi"
        numeric_part = lower_str.replace("mi", "")
        return int(float(numeric_part))
    elif lower_str.endswith("gi"):
        # e.g. "2Gi" -> 2 * 1024 = 2048
        numeric_part = lower_str.replace("gi", "")
        return int(float(numeric_part) * 1024)
    elif lower_str.endswith("ki"):
        # e.g. "1024Ki" -> 1024 / 1024 = 1
        numeric_part = lower_str.replace("ki", "")
        return int(float(numeric_part) / 1024)
    else:
        # e.g. "512" -> 512
        return int(float(lower_str))

# ----- Comparison Helpers ----- #
def is_higher_cpu(actual_cpu_str, default_cpu_str) -> bool:
    """Return True if actual_cpu_str is higher than default_cpu_str (in millicores)."""
    return parse_cpu(actual_cpu_str) > parse_cpu(default_cpu_str)

def is_higher_memory(actual_mem_str, default_mem_str) -> bool:
    """Return True if actual_mem_str is higher than default_mem_str (in MB)."""
    return parse_memory(actual_mem_str) > parse_memory(default_mem_str)
```

```

def is_default_cpu(actual_cpu_str, default_cpu_str) -> bool:
    """Return True if actual == default (in millicores)."""
    return parse_cpu(actual_cpu_str) == parse_cpu(default_cpu_str)

def is_default_memory(actual_mem_str, default_mem_str) -> bool:
    """Return True if actual == default (in MB)."""
    return parse_memory(actual_mem_str) == parse_memory(default_mem_str)

# ----- Core Logic ----- #
def determine_point(data):
    """
    Determine the operational point type:

    1. NORMAL
       - actual < default_values['requests'] for both cpu, memory

    2. NORMAL_TO_CRITICAL
       - requests == default_values['requests']
       - limits == default_values['limits']
       - actual > default_values['requests'] for cpu or memory

    3. CRITICAL
       - actual, requests, limits are ALL higher than the defaults
       (at least one field of each is higher than the default)

    4. CRITICAL_TO_NORMAL
       - requests, limits are above their defaults
       - actual is below the default (requests) for both cpu, memory
    """
    # Extract input values
    req_cpu_str = data["requests"]["cpu"]
    req_mem_str = data["requests"]["memory"]
    lim_cpu_str = data["limits"]["cpu"]
    lim_mem_str = data["limits"]["memory"]
    act_cpu_str = data["actual"]["cpu"]
    act_mem_str = data["actual"]["memory"]

    # Extract defaults
    def_req_cpu = DEFAULT_VALUES["requests"]["cpu"]
    def_req_mem = DEFAULT_VALUES["requests"]["memory"]
    def_lim_cpu = DEFAULT_VALUES["limits"]["cpu"]
    def_lim_mem = DEFAULT_VALUES["limits"]["memory"]

    # 1. NORMAL: actual < default_values.requests (cpu & mem)
    condition_normal = (not is_higher_cpu(act_cpu_str, def_req_cpu) and
                        not is_higher_memory(act_mem_str, def_req_mem))

    # 2. NORMAL_TO_CRITICAL:
    # - requests == default (cpu & mem)
    # - limits == default (cpu & mem)
    # - actual > default.requests for cpu or memory
    condition_normal_to_critical = (
        is_default_cpu(req_cpu_str, def_req_cpu) and
        is_default_memory(req_mem_str, def_req_mem) and
        is_default_cpu(lim_cpu_str, def_lim_cpu) and
        is_default_memory(lim_mem_str, def_lim_mem) and
        (
            is_higher_cpu(act_cpu_str, def_req_cpu) or
            is_higher_memory(act_mem_str, def_req_mem)
        )
    )

    # 3. CRITICAL:
    # - actual is higher than default requests (cpu or mem)
    # - requests is higher than default requests (cpu or mem)
    # - limits is higher than default limits (cpu or mem)
    condition_critical = (
        (is_higher_cpu(act_cpu_str, def_req_cpu) or is_higher_memory(act_mem_str, def_req_mem)) and
        (is_higher_cpu(req_cpu_str, def_req_cpu) or is_higher_memory(req_mem_str, def_req_mem)) and
        (is_higher_cpu(lim_cpu_str, def_lim_cpu) or is_higher_memory(lim_mem_str, def_lim_mem))
    )

    # 4. CRITICAL_TO_NORMAL:
    # - requests, limits are above defaults
    # - actual is below default.requests for cpu & mem
    # => requests > default, limits > default, actual < default
    condition_critical_to_normal = (
        (is_higher_cpu(req_cpu_str, def_req_cpu) or is_higher_memory(req_mem_str, def_req_mem)) and
        (is_higher_cpu(lim_cpu_str, def_lim_cpu) or is_higher_memory(lim_mem_str, def_lim_mem)) and
        (not is_higher_cpu(act_cpu_str, def_req_cpu)) and
        (not is_higher_memory(act_mem_str, def_req_mem))
    )

    # Decide which point applies in priority order
    if condition_critical:
        return "CRITICAL"

```

```

elif condition_normal_to_critical:
    return "NORMAL_TO_CRITICAL"
elif condition_critical_to_normal:
    return "CRITICAL_TO_NORMAL"
elif condition_normal:
    return "NORMAL"
else:
    # Fallback if no exact condition matched
    return "NORMAL"

def _int_mul(value: int, factor: float) -> int:
    """
    Multiplies an integer by a float factor and returns an int.
    E.g. _int_mul(100, 1.2) -> 120
    """
    return int(value * factor)

def _cpu_to_str(millicores: int) -> str:
    """
    Format an integer millicore value back into a string with the 'm' suffix.
    E.g. 120 -> '120m'
    """
    return f"{millicores}m"

def _mem_to_str(megabytes: int) -> str:
    """
    Format an integer MB value back into a string with the 'Mi' suffix.
    E.g. 256 -> '256Mi'
    """
    return f"{megabytes}Mi"

def generate_spec(actual: dict) -> dict:
    """
    Given something like:
    actual = {"cpu": "110m", "memory": "18Mi"}
    Return a dict of the form:
    {
        "spec": {
            "requests": {"cpu": "...", "memory": "..."},
            "limits": {"cpu": "...", "memory": "..."}
        }
    }
    Where each CPU/memory is either scaled from actual (if above defaults)
    or uses the default value (if at or below defaults).
    """
    # Parse default values
    def_req_cpu = parse_cpu(DEFAULT_VALUES["requests"]["cpu"])
    def_req_mem = parse_memory(DEFAULT_VALUES["requests"]["memory"])
    def_lim_cpu = parse_cpu(DEFAULT_VALUES["limits"]["cpu"])
    def_lim_mem = parse_memory(DEFAULT_VALUES["limits"]["memory"])

    # Parse actual values
    actual_cpu = parse_cpu(actual["cpu"])
    actual_mem = parse_memory(actual["memory"])

    # ----- CPU logic -----
    if actual_cpu > def_req_cpu:
        requests_cpu = _int_mul(actual_cpu, 1.2)
        limits_cpu = _int_mul(actual_cpu, 2.4)
    else:
        requests_cpu = def_req_cpu
        limits_cpu = def_lim_cpu

    # ----- Memory logic -----
    if actual_mem > def_req_mem:
        requests_mem = _int_mul(actual_mem, 1.2)
        limits_mem = _int_mul(actual_mem, 2.4)
    else:
        requests_mem = def_req_mem
        limits_mem = def_lim_mem

    return {
        "result": {
            "requests": {
                "cpu": _cpu_to_str(requests_cpu),
                "memory": _mem_to_str(requests_mem)
            },
            "limits": {
                "cpu": _cpu_to_str(limits_cpu),
                "memory": _mem_to_str(limits_mem)
            }
        }
    }

# ----- Flask Endpoints ----- #
@app.route("/v1/data/policy/point", methods=["POST"])

```

```

def logic_endpoint():
    data = request.get_json(force=True)
    point = determine_point(data['input'])
    return jsonify({"result": point})

@app.route("/v1/data/policy/spec", methods=["POST"])
def spec_endpoint():
    data = request.get_json(force=True)
    actual = data['input']
    spec_obj = generate_spec(actual)
    return jsonify(spec_obj)

@app.route("/v1/data/policy/interval", methods=["POST"])
def interval_endpoint():
    data = request.get_json(force=True)
    point_value = data['input']
    if point_value in ("NORMAL_TO_CRITICAL", "CRITICAL"):
        interval = "HIGH"
    else:
        interval = "LOW"

    return jsonify({"result": interval})

if __name__ == "__main__":
    # Run the Flask app. You can also set a different port, debug mode, etc.
    app.run(host="0.0.0.0", port=9500, debug=True)

```

Załącznik 10. Kod LupN

Kod w wersji elektronicznej znajduje się pod linkiem: <https://github.com/0x41gawor/lupus/blob/master/examples/open5gs/sample-loop/master.yaml>

Listing 45. *Kod LupN*

```

apiVersion: lupus.gawor.io/v1
kind: Master
metadata:
  labels:
    app.kubernetes.io/name: lupus
    app.kubernetes.io/managed-by: kustomize
name: lola
spec:
  name: "lola"
  elements:
    - name: "demux"
      descr: "Demuxes Data input into separate elements for each UPF deployment record"
      actions:
        - name: "insert1"
          type: insert
          insert:
            outputKey: "open5gs-upf1"
            value: {name: "open5gs-upf1"}
            next: "insert2"
        - name: "insert2"
          type: insert

```

```

    insert:
      outputKey: "open5gs-upf2"
      value: {name: "open5gs-upf2"}
      next: "print"
- name: "print"
  type: print
  print:
    inputKeys: ["*"]
    next: final
next:
- type: element
  element:
    name: "upf1"
    keys: ["open5gs-upf1"]
- type: element
  element:
    name: "upf2"
    keys: ["open5gs-upf2"]
- name: "upf1"
  descr: "Reconciliation_of_UPF1_deployment"
  actions:
    - name: "print1"
      type: print
      print:
        inputKeys: ["*"]
        next: "opa-point"
    - name: "opa-point"
      type: send
      send:
        inputKey: "*"
        destination:
          type: opa
          opa:
            path: http://192.168.56.112:9500/v1/data/policy/point
            outputKey: "point"
        next: "print2"
    - name: "print2"
      type: print
      print:
        inputKeys: ["*"]
        next: "switch1"
    - name: "switch1"

```

```

type: switch
switch:
  conditions:
    - key: "point"
      operator: eq
      type: string
      string:
        value: "NORMAL"
      next: final
  next: "opa-spec"
- name: "opa-spec"
  type: send
  send:
    inputKey: "actual"
    destination:
      type: opa
      opa:
        path: http://192.168.56.112:9500/v1/data/policy/spec
        outputKey: "spec"
      next: "print3"
- name: "print3"
  type: print
  print:
    inputKeys: ["*"]
    next: "switch2"
- name: "switch2"
  type: switch
  switch:
    conditions:
      - key: "point"
        operator: eq
        type: string
        string:
          value: "CRITICAL"
        next: final
    next: "print4"
- name: "print4"
  type: print
  print:
    inputKeys: ["*"]
    next: "opa-interval"
- name: "opa-interval"

```



```

    type: send
  send:
    inputKey: "point"
    destination:
      type: opa
    opa:
      path: http://192.168.56.112:9500/v1/data/policy/interval
    outputKey: "interval"
  next: "print5"
- name: "print5"
  type: print
  print:
    inputKeys: ["*"]
  next: final
next:
- type: destination
  destination:
    type: http
  http:
    path: http://192.168.56.112:9001/api/data
    method: POST
    keys: ["*"]
- name: "upf2"
  descr: "Reconciliation_of_UPF2_deployment"
  actions:
    - name: "print"
      type: print
      print:
        inputKeys: ["*"]
      next: "opa-point"
    - name: "opa-point"
      type: send
      send:
        inputKey: "*"
        destination:
          type: opa
        opa:
          path: http://192.168.56.112:9500/v1/data/policy/point
        outputKey: "point"
      next: "print2"
    - name: "print2"
      type: print

```

```

    print:
      inputKeys: ["*"]
    next: "switch1"
  - name: "switch1"
    type: switch
    switch:
      conditions:
        - key: "point"
          operator: eq
          type: string
          string:
            value: "NORMAL"
          next: final
      next: "opa-spec"
  - name: "opa-spec"
    type: send
    send:
      inputKey: "actual"
      destination:
        type: opa
        opa:
          path: http://192.168.56.112:9500/v1/data/policy/spec
      outputKey: "spec"
    next: "print3"
  - name: "print3"
    type: print
    print:
      inputKeys: ["*"]
    next: "switch2"
  - name: "switch2"
    type: switch
    switch:
      conditions:
        - key: "point"
          operator: eq
          type: string
          string:
            value: "CRITICAL"
          next: final
      next: "print4"
  - name: "print4"
    type: print

```

```

    print:
      inputKeys: ["*"]
      next: "opa-interval"
- name: "opa-interval"
  type: send
  send:
    inputKey: "point"
    destination:
      type: opa
      opa:
        path: http://192.168.56.112:9500/v1/data/policy/interval
        outputKey: "interval"
      next: "print5"
- name: "print5"
  type: print
  print:
    inputKeys: ["*"]
    next: final
next:
- type: destination
  destination:
    type: http
    http:
      path: http://192.168.56.112:9001/api/data
      method: POST
    keys: ["*"]

```