ZERVIDJANNA

Oprogramowanie systemu Zervi

Lokalna rozmowa, rozpoznawanie twarzy i sterowanie prototypami

Cel dokumentu
Publiczny opis działającego prototypu oprogramowania: od uruchamiania modułów, przez mowę i wzrok, aż po ruch szczęki oraz pojedynczego napędu przyszłej nogi. Dokument rozdziela funkcje już zintegrowane od elementów pozostających planem.

Roboczy ekran startowy systemu Zervidjanny podczas inicjalizacji lokalnego modułu STT.

Wersja dokumentu 0.1 | 23 sierpnia 2026 | Zervi Labs

1. Zakres i aktualny stan systemu

System Zervi jest warstwą programową, która ma połączyć mowę, percepcję i ruch w jedną spójną osobowość robota. Nie jest osobnym systemem operacyjnym w znaczeniu Windowsa czy Linuksa. To aplikacja oraz zestaw współpracujących modułów uruchamianych na komputerze robota. Jej zadaniem jest odebrać sygnały z mikrofonu i kamery, zrozumieć sytuację, przygotować odpowiedź oraz — gdy jest to bezpieczne i celowe — wywołać fizyczną reakcję prototypu.

Najważniejsza decyzja architektoniczna: moduły są rozdzielone. Rozpoznawanie mowy, model rozmowy, synteza głosu, analiza obrazu i sterowanie napędem można uruchamiać oraz rozwijać niezależnie. Przy projekcie budowanym etapami to bardzo praktyczne: awaria kamery nie musi uciszać robota, a test nogi nie wymaga przebudowy całej warstwy rozmowy.

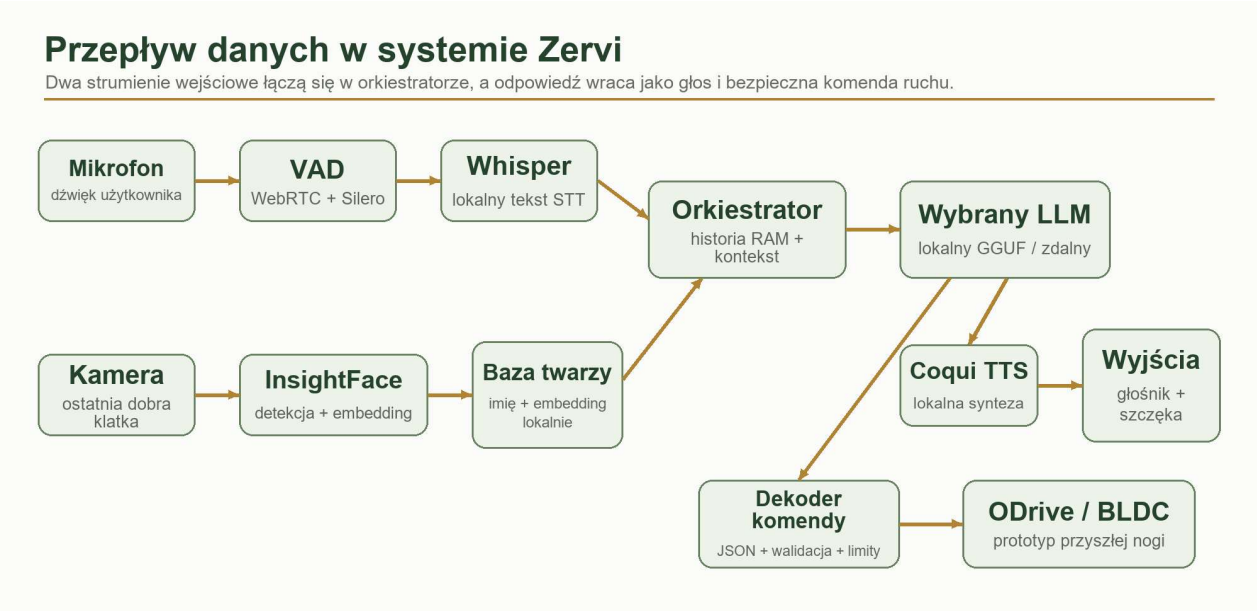
Stan prototypu
STT, TTS, analiza twarzy i baza embeddingów działają lokalnie. Kod obsługuje domyślnie lokalny skwantyzowany model językowy. Konfiguracja rozmowy może także korzystać z modelu zewnętrznego. Dzisiejszy system jest więc hybrydowy; pełna praca offline oraz online jest wariantem wspieranym przez architekturę w zależności od potrzeb użytkownika.

Co jest już zintegrowane

Warstwa	Rola	Stan
Start i konfiguracja	Ładowanie ustawień JSON, modułów oraz raportowanie postępu do ekranu Kivy.	działa w prototypie
Wejście głosowe	VAD wykrywa wypowiedź, a Whisper zamienia ją na polski tekst.	lokalnie
Rozmowa	Wybieralny model językowy i krótka historia dialogu przechowywana w RAM.	profil hybrydowy
Wzrok	Detekcja twarzy, embeddingi, rozpoznawanie znanych osób i etykiety Unknown#X.	lokalnie
Uczenie osób	Przypisanie imienia do jednej nieznanej twarzy po naturalnym przedstawieniu się.	prototyp działa
Głos robota	Coqui XTTS tworzy polską mowę, a warstwa odtwarzania obsługuje kolejne fragmenty audio.	lokalnie
Szczęka	Amplituda mowy steruje położeniem serwa przez port szeregowy.	Prototyp, za wolny
Noga	Trzy dozwolone akcje dla jednego ODrive: ruch względny, kilka wychyleń i stop.	prototyp

2. Architektura i przepływ danych

Główna pętla rozmowy łączy dwa niezależne źródła informacji. Mikrofon dostarcza wypowiedź użytkownika, a kamera — krótką semantyczną informację o widocznych osobach. Orkiestrator nie wysyła modelowi obrazu ani embeddingów. Do kontekstu trafia tekst w rodzaju „kamera widzi Annę” albo „widoczna jest nieznana osoba”. Dzięki temu model może naturalniej prowadzić rozmowę, a ciężka analiza obrazu pozostaje w osobnym, lokalnym torze.



Rys. 1. Uproszczony przepływ danych w systemie Zervi. Kolorem złotym zaznaczono kierunek informacji i komend.

Dlaczego moduły, a nie jeden wielki program

- każdy moduł można uruchomić, wyłączyć albo zamienić konfiguracją bez przepisywania całego rdzenia;
- długi start modeli STT, TTS i LLM jest raportowany do interfejsu zamiast wyglądać jak zawieszona aplikacja;
- kamera, rozmowa, odtwarzanie audio i ruch mogą pracować w oddzielnych wątkach;
- błędy łatwiej przypisać do konkretnej warstwy, co przy robocie z silnikami jest ważniejsze niż efektowne „jakoś działa”;
- ten sam rdzeń może służyć podczas testów na komputerze oraz później na docelowej jednostce obliczeniowej robota.

3. Uruchamianie, konfiguracja i orkiestracja

Punkt startowy tworzy menedżer konfiguracji, rdzeń Zervidjanny, kontroler pośredniczący między rdzeniem a interfejsem oraz UI w Kivy. Ustawienia są rozdzielone na pliki JSON: osobno dla modułów rozmowy, ruchu, interfejsu i logowania. Pozwala to zmienić model, czułość detekcji mowy albo limity ruchu bez szukania stałych rozsianych po całym projekcie.

Sekwencja startowa

1. Wczytanie i sprawdzenie konfiguracji potrzebnej do uruchomienia systemu.
2. Utworzenie loggera wielowątkowego, rdzenia Zervidjanny oraz kontrolera UI.
3. Wyświetlenie ekranu startowego i rozpoczęcie ładowania tylko tych modułów, które są włączone.
4. Przekazywanie postępu przez prosty mechanizm Observer do animowanego ekranu Kivy.
5. Po zgłoszeniu „modules_loaded” uruchomienie głównej pętli autonomicznej w osobnym wątku.
6. Przy zamykaniu: zatrzymanie kamery, STT, TTS oraz napędu i przejście ODrive do stanu bezpiecznego.

Po co ekran ładowania?
Duży model mowy albo syntezy nie pojawia się w pamięci natychmiast. Zamiast patrzeć w nieruchome okno i zgadywać, czy program żyje, użytkownik widzi aktualnie inicjalizowany moduł i postęp startu. To drobiazg, ale bardzo poprawia pracę z prototypem.

Stan i diagnostyka

Logger zapisuje informacje z wielu wątków bez mieszania komunikatów, a moduły przekazują zdarzenia do warstwy nadrzędnej. Rozbudowany panel HMI z mapą, pełną diagnostyką i podglądem wszystkich sensorów pozostaje planem. Obecny interfejs skupia się na starcie systemu i podstawowym przejściu do działania, więc dokumentacja nie udaje, że na ekranie znajduje się już centrum dowodzenia rodem z filmu science fiction.

4. Wejście głosowe: od mikrofonu do tekstu

Rozmowa zaczyna się od ciągłego nasłuchu mikrofonu. Sam zapis audio to jednak za mało: program musi wiedzieć, kiedy użytkownik naprawdę zaczął mówić i kiedy skończył. W tym celu stosowane są dwa mechanizmy — VAD oraz model Silero. Ich zadaniem jest odróżnienie głosu od ciszy i typowego szumu tła.

Tor STT

1. PyAudio pobiera próbki z wybranego wejścia dźwiękowego.
2. VAD wykrywa początek wypowiedzi i utrzymuje nagranie do chwili odpowiednio długiej ciszy.
3. Zebrany fragment trafia do osobnego procesu transkrypcji, żeby cięższe obliczenia nie blokowały reszty aplikacji.
4. Lokalny Whisper przetwarza nagranie na tekst w języku polskim; jeśli jest dostępna CUDA, może użyć GPU, a w przeciwnym razie przejść na CPU.
5. Pusta albo przypadkowa transkrypcja jest pomijana, a poprawny tekst trafia do głównej pętli rozmowy.

Aktualny profil używa modelu Whisper large-v3 i transkrybuje wypowiedź po jej zakończeniu. Kod zawiera eksperymenty z podglądem tekstu w czasie rzeczywistym oraz słowem wybudzającym “hej Zervi”, ale nie są one aktywną częścią obecnej konfiguracji. To celowe rozróżnienie: eksperyment w repozytorium nie staje się automatycznie gotową funkcją robota.

Dlaczego lokalny STT jest ważny
Surowy głos nie musi być wysyłany do zewnętrznej usługi tylko po to, aby powstał tekst. System może działać bez sieci, a opóźnienie zależy głównie od lokalnego sprzętu. To szczególnie istotne w urządzeniu, które ma reagować w domu, warsztacie albo w terenie, bez dostępu do globalnej sieci.

5. Model rozmowy, osobowość i pamięć

Rdzeń systemu nie jest przywiązany do jednego modelu językowego. Menedżer modułów potrafi uruchomić wyuczony wcześniej lokalny model językowy przez llama.cpp albo klienta zewnętrznego API dla bardziej “inteligentnych” modeli działających na większym np. prywatnym serwerze. Główna pętla wybiera dostępny moduł i przekazuje mu wspólny zestaw wiadomości: opis systemowy, ostatnią część rozmowy, aktualny kontekst z kamery oraz bieżącą wypowiedź użytkownika.

Co trafia do modelu

- krótka definicja charakteru i roli Zervidjanny dla trybu API lub brak tych danych przy pracy lokalnej;
- do 100 ostatnich wpisów dialogu przechowywanych w pamięci RAM;
- bieżąca transkrypcja wypowiedzi użytkownika;
- tekstowa obserwacja z kamery, np. lista rozpoznanych osób;
- instrukcja formatu dozwolonej komendy ODrive, jeśli moduł nogi jest włączony.

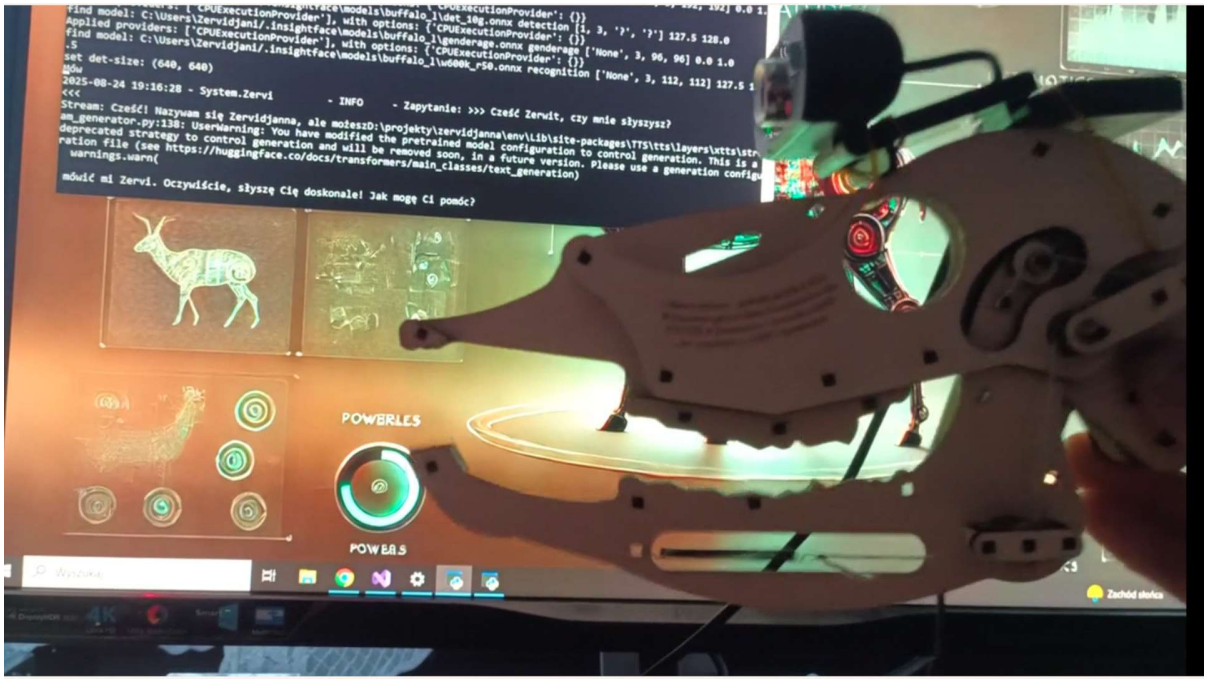
W celu prezentacji systemu historia rozmowy nie jest pamięcią długoterminową, choć taką może być po odpowiedniej konfiguracji. Po restarcie programu znika, a chwilowe dane z kamery nie są dopisywane do niej na stałe. To dobre zachowanie dla prototypu, bo ogranicza gromadzenie przypadkowych informacji, ale w przyszłości trzeba będzie zaprojektować świadomą pamięć zdarzeń: z zasadami zapisu, usuwania i zgodą użytkownika.

Tryb LLM	Zalety	Ograniczenia
Lokalny model	Praca bez łącza, pełna kontrola nad modelem i tekstem rozmowy, brak kosztu pojedynczego zapytania.	Wymaga pamięci RAM/VRAM, energii i czasu na dobranie modelu; jakość zależy od lokalnego sprzętu I wytrenowania modelu.
Model zewnętrzny	Łatwy dostęp do większego modelu i szybkie testowanie różnych wariantów rozmowy.	Transkrypcja i tekstowy kontekst osób opuszczają komputer; działanie zależy od sieci i usługi.

Aktualna konfiguracja
Profil używany obecnie jest lokalny: lokalne STT, TTS i widzenie współpracują z modelem rozmowy uruchamianym na tym samym sprzęcie. Przełączenie na tor online wymaga wyłączenia lokalnego GGUF oraz wcześniejszego przygotowania modelu i zasobów obliczeniowych w zewnętrznym API.

6. Synteza mowy i ruch szczęki

Odpowiedź tekstowa trafia do lokalnej syntezy Coqui XTTS v2. Warstwa „TextToAudioStream” przygotowuje fragmenty mowy, kolejkuje audio i przekazuje je do odtwarzacza. W obecnym głównym torze odpowiedź modelu jest najpierw składana w całość, a dopiero później czytana — dlatego nie opisuję dialogu jako całkowicie strumieniowego od pierwszego tokenu do głośnika.



Fot. 1. Test rozmowy i ruchu prototypowej szczęki. Mechanika zaczyna pełnić funkcję ekspresji, a nie tylko chwytaka.

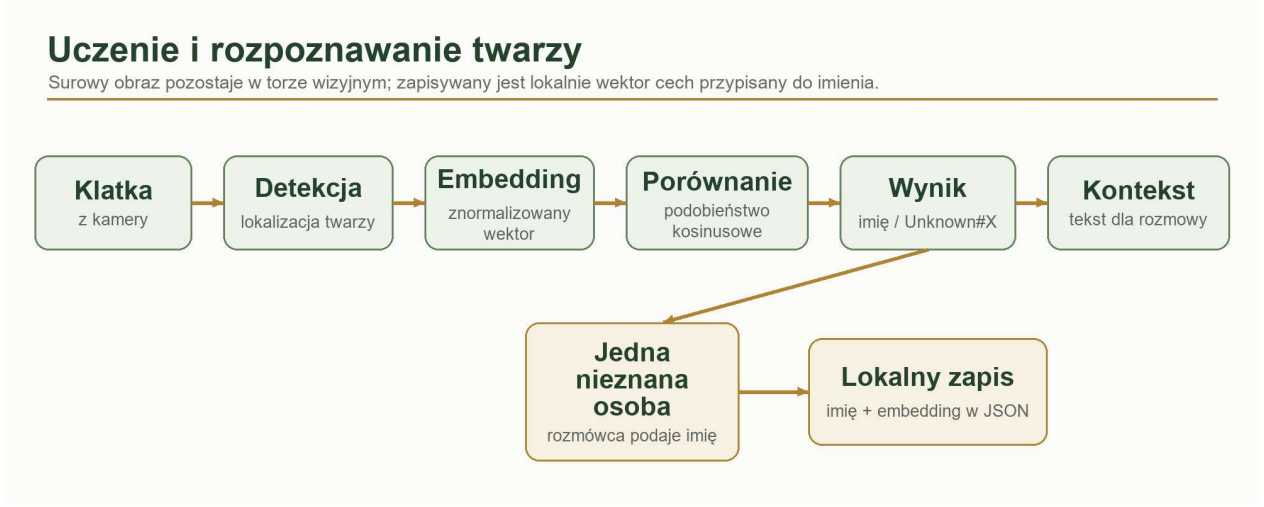
Jak mowa porusza szczęką

Podczas odtwarzania kolejnych próbek audio obliczana jest ich wartość RMS, czyli prosta miara chwilowej amplitudy. Wynik przechodzi przez wygładzanie EMA i ograniczenie maksymalnego skoku, po czym zostaje zamieniony na impuls sterujący serwem. Głośniejszy fragment szerzej otwiera szczękę, cisza ją domyka. Nie jest to analiza fonemów ani próba idealnego odwzorowania ludzkich ust — to celowo prosty mechanizm dający czytelną synchronizację ruchu z mową.

- TTS pozostaje lokalny, więc dźwięk może powstawać bez połączenia z siecią;
- znacznik sterowania nogą jest usuwany przed syntezą i nigdy nie powinien zostać przeczytany;
- serwo szczęki otrzymuje ramki przez połączenie szeregowe;
- parametry portu i mapowania amplitudy są jeszcze rozwiązaniem prototypowym i powinny docelowo trafić do konfiguracji.

7. Wzrok: detekcja i rozpoznawanie osób

Kamera działa w osobnym wątku, aby stale utrzymywać świeżą klatkę. Pierwsze obrazy po uruchomieniu oraz klatki zbyt ciemne nie nadpisują ostatniego poprawnego widoku. Analiza nie musi pracować z pełną liczbą klatek na sekundę; w prototypie wystarcza okresowa aktualizacja, dzięki której model twarzy nie zabiera całej mocy obliczeniowej rozmowie.



Rys. 2. Lokalny tor rozpoznawania twarzy i naturalnego przypisywania imienia.

Embedding zamiast albumu zdjęć

System Zervidjanny wykrywa twarz i zamienia ją na embedding — wektor liczb opisujący charakterystyczne cechy. Wektor jest normalizowany, a następnie porównywany z zapisanymi osobami przez podobieństwo kosinusowe. Jeśli najlepszy wynik przekracza próg prototypu 0,65, system używa zapisanego imienia. W przeciwnym razie twarz otrzymuje tymczasową etykietę Unknown#X.

Nieznane osoby są śledzone między kolejnymi analizami przez porównanie położenia ramek IoU. Embedding jest delikatnie uśredniany, aby pojedyncza gorsza klatka nie zdominowała profilu. To nadal prosty mechanizm śledzenia, ale na stanowisku z jedną kamerą daje stabilniejsze etykiety niż nadawanie nowego numeru w każdej klatce.

8. Naturalne uczenie twarzy

Uczenie osoby odbywa się w trakcie zwykłej rozmowy. Nie trzeba otwierać osobnego formularza ani ręcznie wybierać zdjęcia. Jeśli kamera widzi jedną nieznaną twarz, a rozmówca powie „mam na imię Anna”, „nazywam się Piotr” lub użyje podobnej obsługiwanej konstrukcji, system łączy imię z aktualnym embeddingiem.

1. Kamera wykrywa twarz, dla której nie ma dopasowania w lokalnej bazie.
2. Twarz otrzymuje stabilną etykietę tymczasową, np. Unknown#1.
3. Rozmówca podaje imię w naturalnym zdaniu, a prosty parser wyodrębnia nazwę.
4. System sprawdza, czy w kadrze jest dokładnie jedna nieznana osoba.
5. Embedding zostaje przypisany do imienia i zapisany w lokalnym pliku JSON.
6. W bieżącym kontekście etykieta Unknown zostaje zastąpiona imieniem, więc rozmowa może od razu to uwzględnić.

Zabezpieczenie przed przypadkowym przypisaniem
Jeśli kamera widzi kilka nieznanych osób, automatyczny zapis jest blokowany. Samo usłyszenie imienia nie wystarcza wtedy do ustalenia, której twarzy dotyczy. To mały warunek, ale chroni bazę przed bardzo przekonującymi pomyłkami.

Prywatność danych biometrycznych

Aktywny tor nie zapisuje surowych klatek z kamery. Trwałym elementem jest imię i embedding twarzy. Nadal są to jednak dane biometryczne i lokalność nie czyni ich automatycznie bezpiecznymi. Baza ma obecnie postać zwykłego pliku JSON, bez szyfrowania. W docelowej wersji powinna dostać kontrolę dostępu, możliwość jawnego usuwania profilu, kopię zapasową oraz szyfrowanie danych w spoczynku.

9. Kontekst wizyjny w rozmowie

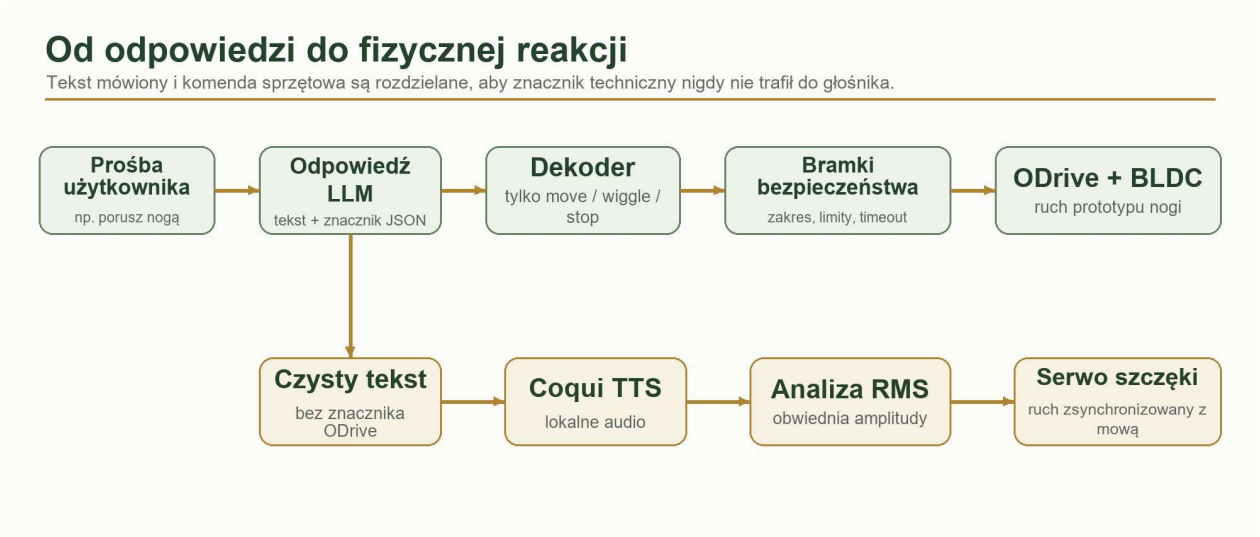
Rozpoznanie twarzy nie jest końcem toru wizyjnego. Najważniejsze jest to, jak wynik zostaje użyty. Zamiast przesyłać modelowi zdjęcie, system buduje krótką wiadomość sytuacyjną: kto jest widoczny, czy obecna jest osoba nieznana i czy właśnie zapisano nowe imię. Wiadomość ma rolę obserwacji, a nie wypowiedzi rozmówcy.

Dane	Pozostają lokalnie	Mogą trafić do LLM
Klatka z kamery	tak	nie
Embedding twarzy	tak	nie
Imię zapisanej osoby	tak w bazie	tak jako tekst kontekstu
Etykieta Unknown#X	tak w stanie bieżącym	tak jako tekst kontekstu
Transkrypcja wypowiedzi	tak w bieżącej sesji	tak

Przy lokalnym LLM cały ten kontekst pozostaje na komputerze robota. Przy modelu zewnętrznym sama transkrypcja i tekstowe nazwy osób są wysyłane do usługi. Obraz i embedding co ważne nadal zostają lokalnie. Jako że imię również może być informacją prywatną bez dodatkowych danych w postaci zdjęcia czy embeddingu, użytkownik powinien wiedzieć, który tryb jest aktywny.

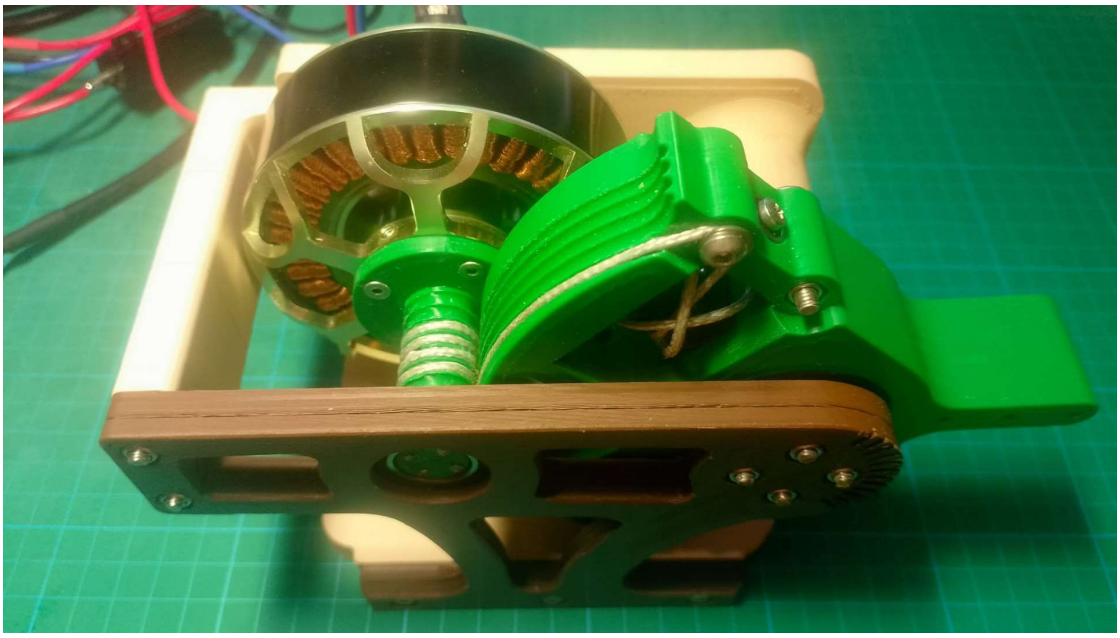
Efekt dla rozmowy
Zervidjanna może przywitać znaną osobę, zapytać nieznaną o imię albo odnieść się do tego, kto aktualnie stoi przed kamerą. Nie musi przy tym udawać, że analizuje każdy piksel — dostaje już gotową, krótką obserwację przygotowaną przez lokalny moduł wizyjny.

10. Sterowanie prototypami: noga i szczęka



Rys. 3. Rozdzielenie mowy od komend sprzętowych oraz dwie obecne ścieżki reakcji fizycznej.

Sterowanie fizycznym napędem nie może polegać na bezpośrednim wykonaniu dowolnego tekstu wygenerowanego przez model. Dla nogi przyjęty został mały, zamknięty zestaw akcji: „move”, „wiggle” i „stop”. Model może dołączyć do odpowiedzi niewypowiadany znacznik JSON, ale dekoder akceptuje tylko znany format, liczby skończone i dozwolone nazwy akcji. Jeśli znacznika zabraknie, prosty lokalny parser potrafi odczytać podstawową prośbę użytkownika.



Fot. 2. Stanowisko napędu capstan z silnikiem BLDC i niewidocznym na fotce sterownikiem ODrive — fizyczny odbiorca komend testowych z systemu Zervi.

Zabezpieczenia obecne w sterowniku nogi

- ograniczony zakres całego stawu oraz maksymalny pojedynczy ruch względny;
- limity amplitudy i liczby powtórzeń dla gestu „wiggle”;
- limity prędkości, przyspieszenia, hamowania i prądu ODrive;
- tylko jeden aktywny ruch naraz, timeout dojścia do celu i możliwość anulowania;
- sprawdzenie błędów osi, kalibracji silnika i gotowości enkodera przed ruchem;
- tryb „dry_run”, który pozwala testować rozmowę i dekodowanie bez podłączonego napędu;
- przejście do IDLE przy zatrzymaniu, błędzie albo zamykaniu programu.

Pozycja enkodera z chwili pierwszego połączenia staje się programowym zerem. Nie jest to absolutny kąt mechaniczny, dlatego ustawienie stanowiska przed startem ma znaczenie. Kalibracja ODrive i kasowanie błędów pozostają osobnym, jawnym etapem — aplikacja rozmowy celowo nie robi tego automatycznie.

Granica obecnych zabezpieczeń
Są to blokady przydatne na stanowisku testowym, lecz nie zastępują niezależnego watchdoga, fizycznego E-stopu, krańcówek ani limitów działających w sterowniku czasu rzeczywistego. W pełnym robocie komenda pochodząca z modelu powinna dodatkowo przejść przez osobną bramkę potwierdzającą rzeczywistą intencję użytkownika.

11. Dlaczego lokalne przetwarzanie ma sens

Robot to nie zwykły formularz internetowy. Ma mikrofony, kamery i napędy, a część informacji dotyczy konkretnych osób stojących przed urządzeniem. Im bliżej źródła można przetworzyć dane, tym łatwiej zachować kontrolę nad tym, co jest zapisywane i co opuszcza komputer. Lokalność nie jest tu modnym dodatkiem — jest elementem architektury urządzenia.

Korzyść	Znaczenie w Zervidjannie
Prywatność	Surowe audio, obraz i embeddingi mogą pozostać przy robocie zamiast wędrować przez kolejne usługi.
Praca bez sieci	Rozpoznawanie i przetwarzanie mowy, głos i twarze nadal działają w warsztacie lub terenie bez stabilnego łącza.
Opóźnienie	Brak transmisji sieciowej upraszcza reakcje głosowe i fizyczne, jeśli lokalny sprzęt jest wystarczająco wydajny.
Kontrola wersji	Można zachować dokładnie wybrany model i konfigurację, co ułatwia powtarzalne testy.
Koszt działania	Po przygotowaniu sprzętu lokalny tor nie nalicza opłaty za każdą wypowiedź czy analizę obrazu.
Integracja sprzętu	Komendy, limity i awaryjne zatrzymanie pozostają blisko sterowników zamiast zależeć od odległej usługi.

Lokalnie nie znaczy automatycznie lepiej we wszystkim

- duże modele wymagają pamięci, mocnej karty obliczeniowej, energii i chłodzenia;
- mniejszy model lokalny może rozmawiać gorzej niż większy model zewnętrzny;
- aktualizacje i kopie modeli trzeba przygotowywać samodzielnie;
- lokalny plik z biometrią nadal można utracić albo skopiować, jeśli komputer nie jest chroniony;
- tryb hybrydowy bywa rozsądnym etapem rozwoju, ale interfejs powinien jasno pokazywać, kiedy tekst opuszcza urządzenie.

Docelowy kierunek
Najważniejsze funkcje sprzętowe i dane biometryczne powinny pozostać lokalne. Model rozmowy może być wybieralny: lokalny do pełnej autonomii i prywatności, zewnętrzny do testów większych modeli. To użytkownik powinien świadomie wybrać kompromis, a nie odkrywać go przypadkiem po odłączeniu Wi-Fi.

12. Bezpieczeństwo i ograniczenia prototypu

System potrafi już wywołać ruch fizycznego napędu, dlatego nawet demonstracyjny kod trzeba traktować poważnie. Warstwa programowa zawiera limity i kontrolę błędów, ale nie zastępuje niezależnych zabezpieczeń sprzętowych. W pełnym robocie awaria procesu, zawieszenie komputera albo błędny odczyt czujnika nie mogą pozostawić silnika bez nadzoru.

Zasady testów

- pierwsze próby wykonywać w „dry_run”, a następnie bez obciążenia mechanicznego;
- kalibrować ODrive osobnymi narzędziami i sprawdzać błędy przed uruchomieniem rozmowy;
- utrzymywać fizyczne odcięcie zasilania w zasięgu ręki;
- zaczynać od małych kątów, prędkości i prądów, obserwując mechanikę oraz enkoder;
- nie traktować programowego zera jako absolutnej pozycji bez dodatkowego czujnika bazowania;
- nie przechowywać publicznie bazy embeddingów ani ustawień dostępowych do usług zewnętrznych.

Co trzeba dodać przed pełnym robotem

Obszar	Następny poziom zabezpieczenia
Napędy	Niezależny watchdog, fizyczny E-stop, krańcówki/bazowanie, limity po stronie sterownika czasu rzeczywistego.
Komendy AI	Druga bramka potwierdzająca intencję użytkownika także dla poprawnego znacznika zwróconego przez model.
Baza twarzy	Szyfrowanie, kontrola dostępu, jawne usuwanie profilu i polityka przechowywania danych.
Interfejs	Czytelny stan lokalny/zdalny, aktywne sensory, zgody oraz natychmiastowy przycisk stop.
Odporność	Obsługa zaniku modułu, automatyczne przejście do bezpiecznego stanu i testy po restarcie.

13. Granica między funkcją a planem

Repozytorium systemu robota na chwilę obecną zawiera również eksperymenty z detekcją obiektów, segmentacją, słowem wybudzającym, nawigacją, równowagą i rozbudowanym HMI. Są ważne jako kierunek prac, ale nie są jeszcze połączone z główną pętlą w sposób pozwalający nazwać je gotowymi funkcjami Zervidjanny. Ta dokumentacja celowo koncentruje się na torze, który rzeczywiście tworzy dziś spójny demonstrator.

Działa w głównym torze	Eksperyment / dalszy etap
Modułowy start, Kivy i raportowanie postępu	Pełny HMI z mapą, statystykami i diagnostyką wszystkich podsystemów
Lokalny Whisper z VAD	Aktywne słowo wybudzające i stała transkrypcja podglądowa, rozpoznawanie osób po mowie
Twarze: detekcja, rozpoznawanie i lokalna baza	Ogólna detekcja obiektów oraz segmentacja w pętli zachowania
Lokalny TTS i amplitudowy ruch szczęki	Fonemowa animacja pyska i pełna mimika głowy
Jeden napęd ODrive sterowany prostą komendą	Koordynacja wielu stawów, równowaga, chód i planowanie trasy
Historia dialogu w RAM	Kontrolowana pamięć długoterminowa osób, zdarzeń i preferencji

Najbliższy sensowny etap

Dalszy rozwój według mnie powinien iść w dwóch równoległych kierunkach. Po stronie oprogramowania warto dopracować wybór lokalnego i zdalnego LLM, ochronę bazy twarzy, konfigurację szczęki oraz widoczny stan prywatności. Po stronie sprzętu — przenieść krytyczne limity i watchdog do sterownika niższego poziomu, a następnie rozszerzać pojedynczy napęd o rzeczywistą kinematykę nogi. Dopiero wtedy Zervi przestanie tylko rozmawiać o spacerze i randkowaniu, a zacznie mieć szansę naprawdę zrobić krok.

14. Technologie i mapa kodu

Poniższa mapa nie jest instrukcją instalacji ani pełnym spisem zależności. Pokazuje najważniejsze elementy, które tworzą aktualny tor systemu i ułatwiają mi dalsze prace w kodzie.

Element	Technologia / moduł	Odpowiedzialność
Rdzeń	Python 3.11	Orkiestracja rozmowy, stanu, wizji i reakcji sprzętowych.
Interfejs	Kivy + Observer	Ekran startowy, postęp inicjalizacji i komunikacja UI-rdzeń.
Dźwięk wejściowy	PyAudio, WebRTC VAD, Silero VAD	Nagrywanie oraz wyznaczanie początku i końca wypowiedzi.
STT	Whisper / PyTorch	Lokalna transkrypcja polskiej mowy z obsługą CPU lub GPU.
Rozmowa	llama.cpp / klient API	Lokalny model GGUF albo model zewnętrzny wybierany konfiguracją.
TTS	Coqui XTTS v2	Lokalna synteza mowy i przygotowanie kolejnych fragmentów audio.
Wzrok	OpenCV, InsightFace, ONNX Runtime, NumPy	Kamera, twarze, embeddingi, porównanie i kontekst osób.
Szczęka	RMS audio + port szeregowy	Mapowanie amplitudy mowy na impuls serwa prototypowej szczęki.
Noga	ODrive Python API	Walidowany ruch jednego napędu BLDC z limitami i zatrzymaniem.
Konfiguracja	JSON + logger wielowątkowy	Wybór modułów, parametrów i śledzenie zdarzeń diagnostycznych.

Co postaram się dodać w kolejnej aktualizacji

- podgląd kamery z ramką znanej osoby i wynikiem podobieństwa porównanym do nieznanej osoby;
- ekran porównujący tryb lokalnego GGUF z profilem zewnętrznym;
- dokładniejsze opisanie sterownika szczęki i połączenia szeregowego podczas TTS;
- rozbudowane stanowisko ODrive w chwili wykonania polecenia głosowego, z widocznym awaryjnym odcięciem zasilania i informacją zwrotną o błędzie do systemu robota.

15. Podsumowanie

Oprogramowanie Zervidjanny przeszło drogę od osobnych testów mowy, obrazu i napędów do jednego prototypowego toru interakcji. Użytkownik może mówić do robota, lokalny STT zamienia głos na tekst, kamera rozpoznaje osoby, model przygotowuje odpowiedź, TTS nadaje jej głos, a szczęka lub pojedynczy napęd nogi mogą wykonać fizyczną reakcję. To nadal stanowisko badawcze, ale najważniejszy fundament już istnieje: informacja zaczyna przechodzić od człowieka do programu i z programu z powrotem do świata fizycznego.

Najciekawszą częścią tego etapu nie jest sam wybór bibliotek. Jest nią połączenie modułów tak, aby każdy zachował jasną odpowiedzialność. Wzrok ma rozpoznawać, rozmowa ma budować odpowiedź, a sterownik ruchu ma pilnować granic. Model językowy może zaproponować gest, ale silnik nadal powinien słuchać twardych limitów, a nie poetyckiego nastroju Zervi.

Wniosek
System jest gotowy do dalszych testów integracyjnych, lecz nie do autonomicznego sterowania pełnym robotem. Następne wersje będą z biegiem czasu rozwijać lokalność, ochronę danych i niezależne bezpieczeństwo sprzętowe równolegle z mechaniką. Dzięki temu Zervidjanna będzie rosła moduł po module, zamiast pewnego dnia obudzić się jako tysiące linii kodu przyklejonych taśmą do silnika.

ZERVI LABS

Od idei do działającego prototypu.