Lokalny asystent językowy i głosowy dla robota Unitree G1

Modele LLM, rozpoznawanie i synteza mowy, architektura ROS 2/DDS

Streszczenie

Przedmiotem analizy jest możliwość uruchomienia w pełni lokalnego systemu interakcji głosowej i semantycznej na komputerze developerskim robota humanoidalnego Unitree G1. Analizowano ograniczenia platformy NVIDIA Jetson Orin NX, dziewięć kandydatów modeli językowych, pięć silników automatycznego rozpoznawania mowy (ASR) i pięć rozwiązań syntezy mowy (TTS), ze szczególnym uwzględnieniem języka polskiego, pracy bez Internetu i konkurencji o zasoby obliczeniowe z percepcją robota. Proponowana architektura oddziela moduł probabilistyczny LLM od deterministycznego systemu wykonania umiejętności i sprzętowej kontroli ruchu. Opracowano kontrakty komunikacyjne ROS 2/DDS, reguły Safety Gate, system przełączania trybów jakości i degradacji, procedury instalacji offline, monitoringu, testów HIL i wycofywania aktualizacji. Dane katalogowe i wyniki opublikowanych benchmarków oddzielono od prognoz inżynierskich wymagających odrębnej weryfikacji na robotycznej platformie docelowej.

Słowa kluczowe: robotyka humanoidalna; Unitree G1; Jetson Orin NX; lokalne modele językowe; ASR; TTS; ROS 2; DDS; Edge AI; bezpieczeństwo funkcjonalne; optymalizacja inferencji.

1. Cel, kryteria i granice wnioskowania

Głównym celem jest zdefiniowanie architektury działającej bez usług chmurowych, w której operator może prowadzić rozmowę z robotem oraz inicjować wyłącznie dopuszczone działania. Ocenie podlega nie tylko jakość modelu językowego, ale też całkowity czas odpowiedzi głosowej, stabilność długiej pracy, degradacja przy współdzieleniu GPU, podatność na błędy rozpoznania mowy oraz bezpieczeństwo działań fizycznych.

Kategoria danych	Sposób interpretacji
Specyfikacja sprzętu	Dane deklarowane dla wariantu PC2 / Orin NX, wymagające audytu konfiguracji konkretnego robota.
Dane o modelach	Parametry modeli i formatów; dostępność artefaktów oraz warunki użycia zależą od wersji.
Benchmarki zewnętrzne	Wyniki ograniczone do podanego sprzętu, kontekstu i silnika, nieprzenoszalne bez testów.
Budżety latencji i RAM	Cele oraz wstępne oszacowania. Wymagają pomiarów P50/P95/P99 i warunków termicznych.
Bezpieczeństwo	Wymagania projektowe, niezależne od jakości odpowiedzi modelu.

2. Platforma sprzętowa i granice integracji

Wersje/konfiguracje G1 bywają różne, ale z perspektywy uruchomienia LLM kluczowe są trzy fakty:

1. Po pierwsze, G1 ma zamkniętą warstwę sterowania lokomocją („locomotion computer”, czarna skrzynka) i oddzielny komputer do uruchamiania aplikacji użytkownika („development computer”). To na tym drugim należy uruchamiać LLM i całą logikę aplikacyjną.
2. Równolegle w polskich materiałach sprzedażowych G1 pojawia się informacja o 8-rdzeniowym CPU i opcjonalnych wariantach z NVIDIA Jetson Orin.
3. Po trzecie, cechy Jetson Orin NX 16GB istotne dla LLM: 16GB LPDDR5 (128-bit), przepustowość pamięci ~102 GB/s, CPU do 8 rdzeni, GPU Ampere (1024 CUDA, 32 Tensor), a tryby mocy dla 16GB obejmują 10W/15W/25W (w praktyce na robotach często liczy się headroom termiczny i stabilność zasilania).

W G1 urządzenia są spięte przez wewnętrzny przełącznik L2; domyślnie nie ma routera i DHCP, a więc konfiguracja IP jest statyczna. Przykładowo: locomotion computer bywa pod 192.168.123.161, development computer pod 192.168.123.164, a domyślne konto to unitree/123 (wprost zalecone jest natychmiastowe zmienienie hasła przy podłączaniu do Internetu).

Z punktu widzenia wdrożenia LLM oznacza to, że „zdalny offload” (edge server) jest wykonalny, ale trzeba go świadomie zaprojektować (routing, firewall, segmentacja), bo włączenie Internetu dla podsieci robota może objąć także komponenty krytyczne.

Materiały opisujące G1 (w tym PL) wskazują typowe sensory i multimedia: m.in. czujniki/algorytmy percepcyjne, elementy audio (np. głośnik) i łączność bezprzewodową; w manualach i kartach produktu przewijają się również elementy typu kamera głębi / lidar w zależności od wariantu.

W praktyce dla LLM największym ograniczeniem nie jest „czy CPU istnieje", tylko czy mamy stabilny budżet mocy i termikę: Jetson Orin NX ma tryby 10–25W, a przy obciążeniu GPU + wideo + ASR łatwo wejść w throttling, co powoduje jitter latencji (a to jest zabójcze dla interakcji człowiek–robot).

Bez inwentaryzacji konkretnego egzemplarza G1 nie są znane: dokładna wersja JetPack/Ubuntu na komputerze developerskim, typ nośnika (NVMe/eMMC), dostępna przestrzeń dyskowa na modele, oraz czy kamera RGB- D jest podpięta do locomotion computer czy development computer (w przewodniku wspomniano, że zależało to od partii produkcyjnej).

Dla konfiguracji referencyjnej przyjęto założenie: development computer = Jetson Orin NX 16GB i środowisko Jetson Linux/JetPack (ARM64 + CUDA), Przed wdrożeniem konieczna jest weryfikacja tej konfiguracji.

Wariant PC2=Jetson Orin NX 16 GB jest konfiguracją referencyjną. Użycie innych odmian G1 lub innego obrazu JetPack wymaga ponownej weryfikacji dostępnej pamięci GPU, dostępu do kamer, struktury sieci oraz kompatybilności binarnej. Traktowanie wszystkich jednostek G1 jako identycznych sprzętowo prowadziłoby do niepowtarzalnych wyników.

Przewodniki integracyjne G1 opisują statyczną sieć wewnętrzną i sugerują zmianę domyślnych haseł, gdy robot ma mieć kontakt z Internetem. Dla trybu w pełni offline jest to korzystne: utrzymywana jest szczelna segmentacja oraz minimalizowana powierzchnia ataku, a jednocześnie należy zaplanować offline-updates (nośnik, repozytorium artefaktów, podpisy).

3. Modele LLM: pełne porównanie kandydatów

Zestawienie obejmuje modele tekstowe i konwersacyjny model mowy PersonaPlex. Szacunki pamięci i latencji opierają się na wielkości wag, publicznym benchmarku llama.cpp dla Jetson Orin NX 16 GB oraz charakterystyce KV-cache. W tym benchmarku dla Llama 2 7B Q4_0 raportowano około 18 tok./s na GPU CUDA i 9 tok./s na CPU. Nie są to bezpośrednie pomiary G1.

Uwaga o „microGPT": to przede wszystkim minimalna implementacja treningu/inferencji małego transformera „od zera" (edukacyjna, a nie gotowy, silny model dialogowy). Nadaje się do demonstracji i ewentualnie do bardzo wąskich modeli znakowych, ale nie jest realnym kandydatem na moduł decyzyjny robota w produkcji.

Wniosek porównawczy: jeżeli celem jest „realny" asystent na robocie, to najbardziej wdrażalne i przewidywalne są Qwen3.5- 2B/4B (int4) lub Phi- 3- mini (int4/ONNX). Modele 7B+ (Mistral, PersonaPlex, Qwen3.5- 9B) są możliwe, ale rosną ryzyka: termika, konkurencja o RAM z percepcją (wideo), niestabilność latencji i OOM.

Model 1: Qwen3.5- 2B

Licencja	Apache- 2.0
Liczba parametrów	2B
Kwantyzacja i formaty	Int4/Int8/FP16 via GGUF (llama.cpp) + AWQ/GPTQ praktycznie dostępne przez ekosystem; kontrola kontekstu konieczna (domyślnie 262k)
Wymagania pamięci	FP16 ~3.7 GiB, int4 ~0.93 GiB (same wagi; + KV-cache zależny od kontekstu) (estymacja z liczby parametrów)
Wydajność referencyjna i szacunki	Przy int4 i krótkim kontekście: ~40–80 tok/s, TTFT typowo <0.5 s dla krótkiego promptu (skalowanie względem 7B Q4_0 na Orin NX)
Środowisko uruchomieniowe	Transformers/vLLM/SGLang/KTransformers (oficjalne przykłady serwowania)
Dojrzałość i ograniczenia	Wysoka (oficjalne karty modelu + rekomendowane silniki serwujące)

Model 2: Qwen3.5- 4B

Licencja	Apache- 2.0
Liczba parametrów	4B
Kwantyzacja i formaty	Analogicznie do wariantu 2B; konieczne ograniczenie kontekstu (262k default). Dodatkowo Qwen3.5 domyślnie generuje „thinking content", co warto wyłączyć dla robota (latencja/ryzyko pętli)
Wymagania pamięci	FP16 ~7.45 GiB, int4 ~1.86 GiB + KV-cache (estymacja)

Wydajność referencyjna i szacunki	~20–40 tok/s na GPU dla int4; CPU-only wyraźnie wolniej (kilka–kilkanaście tok/s)
Środowisko uruchomieniowe	Transformers/vLLM/SGLang/KTransformers; API OpenAI- kompatybilne łatwe do wystawienia
Dojrzałość i ograniczenia	Wysoka (duża adopcja ekosystemu Qwen; jasne ścieżki serwowania)

Model 3: Qwen3.5-9B

Licencja	Apache- 2.0
Liczba parametrów	9B
Kwantyzacja i formaty	Int4 konieczny na 16GB; długi kontekst trzeba zredukować (262k default, deklarowana możliwość rozszerzania)
Wymagania pamięci	FP16 ~16.8 GiB (praktycznie za dużo), int4 ~4.2 GiB + KV-cache (estymacja)
Wydajność referencyjna i szacunki	~8–18 tok/s (int4, krótki kontekst), TTFT zwykle większy niż dla 2–4B (estymacja na bazie 7B Q4_0)
Środowisko uruchomieniowe	Transformers/vLLM/SGLang/KTransformers
Dojrzałość i ograniczenia	Wysoka, ale większe ryzyko OOM/thermal throttling na robocie

Model 4: Microsoft Phi-3-mini-4k-instruct

Licencja	MIT
Liczba parametrów	3.8B (wprost w karcie jako „3.8B- param”)
Kwantyzacja i formaty	Oficjalnie opisane: ONNX Runtime (w tym int4 przez AWQ/RTN) + GGUF dla CPU; flash- attention domyślnie
Wymagania pamięci	FP16 ~7.1 GiB, int4 ~1.77 GiB + KV-cache (estymacja)
Wydajność referencyjna i szacunki	Podobnie do 4B: ~20–40 tok/s (int4) na GPU; plus atut: gotowe artefakty ONNX/int4
Środowisko uruchomieniowe	Transformers, ONNX Runtime (cross- platform), GGUF
Dojrzałość i ograniczenia	Bardzo wysoka (MIT + szeroki ekosystem, ONNX)

Model 5: Meta Llama-3.2-3B-Instruct

Licencja	Licencja społecznościowa Meta + AUP (warunki komercyjne)
Liczba parametrów	3.21B
Kwantyzacja i formaty	Dostępne liczne konwersje GGUF/int4 (ekosystem); uwaga na zgodność licencji i dystrybucję
Wymagania pamięci	FP16 ~6.0 GiB, int4 ~1.5 GiB (estymacja)
Wydajność referencyjna i szacunki	~25–45 tok/s (int4) na GPU (estymacja)
Środowisko uruchomieniowe	Transformers + „original llama codebase”
Dojrzałość i ograniczenia	Bardzo wysoka technicznie, średnia prawnie (restrykcje licencji)

Model 6: Mistral- 7B- Instruct- v0.2

Licencja	Apache- 2.0
Liczba parametrów	7B
Kwantyzacja i formaty	Bardzo bogaty ekosystem quantów (GGUF/GPTQ/AWQ w społeczności); model jawnie bez mechanizmów moderacji
Wymagania pamięci	FP16 ~13.0 GiB, int4 ~3.26 GiB (estymacja)
Wydajność referencyjna i szacunki	Zbliżone do 7B z benchmarku: ~12–20 tok/s (int4) na Orin NX GPU
Środowisko uruchomieniowe	Transformers + liczne serwery (community)
Dojrzałość i ograniczenia	Wysoka, ale wymaga warstwy safety (brak moderacji)

Model 7: Google Gemma 2 (2B/2B- IT)

Licencja	Gemma Terms of Use (niestandardowe)
Liczba parametrów	~2B
Kwantyzacja i formaty	Dla lokalnego uruchamiania: gemma.cpp (minimalistyczny C++ runtime) + ekosystem GGUF (llama.cpp)
Wymagania pamięci	Zależnie od wariantu/formatu; w GGUF Q4 często ~1–1.5 GiB dla 2B- klasy (orientacyjnie, zależne od konkretnego pliku)
Wydajność referencyjna i szacunki	W praktyce szybkie na CPU/GPU dzięki wyspecjalizowanym runtime'om, ale tu decyduje prawo/licencja i łatwość dystrybucji
Środowisko uruchomieniowe	gemma.cpp, Transformers, GGUF
Dojrzałość i ograniczenia	Wysoka technicznie, średnia operacyjnie (akceptacja licencji, gating)

Model 8: NVIDIA PersonaPlex- 7B- v1 (speech- to- speech, full- duplex)

Licencja	Wagi: NVIDIA Open Model License (gated), kod: MIT
Liczba parametrów	7B
Kwantyzacja i formaty	Brak „typowej" kwantyzacji w karcie; repo przewiduje CPU offload przy braku VRAM (accelerate)
Wymagania pamięci	Wagi klasy 7B + dodatkowe komponenty audio; realnie wymaga mocnego GPU lub kompromisów (offload)
Wydajność referencyjna i szacunki	Latencje z benchmarku FullDuplexBench są niskie (rzędu setek ms), ale raportowane na mocnym sprzęcie; na Jetson spodziewany duży spadek
Środowisko uruchomieniowe	PyTorch + serwer moshi
Dojrzałość i ograniczenia	Wysoka naukowo (świeże badania), średnia wdrożeniowo na edge (ciężar + licencja)

Model 9: microGPT (Karpathy, „art project")

Licencja	Brak „standardowej" licencji modelu; to 1- plikowy kod edukacyjny
Liczba parametrów	„mikro" (zależnie od ustawień)
Kwantyzacja i formaty	Nie dotyczy typowych quantów LLM; to eksperyment/trening od zera
Wymagania pamięci	Minimalne; działa nawet bez numpy/pytorch

Wydajność referencyjna i szacunki	Nieporównywalne z LLM dialogowymi; niezalecane jako moduł sterowania robota
Środowisko uruchomieniowe	Pure Python (bez zależności)
Dojrzałość i ograniczenia	Niska jako produkt LLM; wysoka jako materiał dydaktyczny

Oszacowania szybkości wyrażone w tok./s nie są wynikami jednego kontrolowanego eksperymentu na G1. Ich zestawienie służy do wyboru kandydatów do pomiaru, a nie do formalnego rankingu statystycznego. Minimalny rozmiar samych wag w int4 nie opisuje całego zapotrzebowania pamięci; konieczne są osobne rezerwy na KV-cache, buforowanie i percepcję.

4. Dobór modelu i zasady działania inferencji

Rekomendacja bazowa („domyślny mózg rozmowy"): Qwen3.5-4B w int4, z ograniczonym kontekstem i wyłączonym „thinking content".

Uzasadnienie: Qwen3.5-4B ma Apache-2.0 (proste w dystrybucji komercyjnej), jest wystarczająco mały, by w kwantyzacji 4-bit zmieścić się komfortowo w 16GB klasy Orin NX (waga rzędu kilku GiB + KV-cache), a jednocześnie daje znaczący skok zdolności względem 2B przy nadal sensownej latencji.

Ponieważ Qwen3.5 domyślnie operuje w trybie „thinking" (wstawia blok <think>…</think>), dla robota zaleca się tryb non-thinking jako domyślny (mniej tokenów, mniejsza latencja, mniejsze ryzyko pętli generacji). Sam model ostrzega też, że w mniejszym wariancie 0.8B mogą pojawiać się „thinking loops", co w systemie robotycznym trzeba traktować jako realne ryzyko zawieszenia dialogu.

Rekomendacja „lekka" (gdy liczy się czas reakcji i zasoby na percepcję): Qwen3.5-2B w int4.

Ten wariant jest bardzo atrakcyjny, gdy równolegle uruchamiasz ASR/TTS/wizję, a robot ma zachować płynność odpowiedzi. Przy założeniu podobnej optymalizacji jak w benchmarku 7B Q4_0 na Orin NX, 2B da zwykle kilkadziesiąt tok/s i niską latencję pierwszego tokena.

Rekomendacja alternatywna (silna przenośność + gotowe ONNX/int4): Phi-3-mini-4k-instruct (MIT).

Phi-3 ma bardzo mocną stronę operacyjną: w karcie modelu opisano wsparcie ONNX Runtime na wielu platformach oraz gotowe konfiguracje int4 (m.in. AWQ/RTN w kontekście ONNX). To bywa przewagą na embedded, gdy wymagane jest uniknąć ciężkiego stosu Pythona albo chcesz standaryzować pipeline pod ONNX.

Kiedy w ogóle rozważać większy model

Qwen3.5-9B (int4) ma sens jako tryb „heavy" dla zadań wymagających dłuższego rozumowania albo lepszego planowania narzędzi, ale tylko jeśli:

- ograniczysz kontekst (KV-cache),
- zapewnisz stabilny budżet termiczny,
- rozdzielisz zasoby GPU między percepcję a LLM (QoS),
- dostępna jest twarde timeouts i fallback.

Qwen3.5-9B ma domyślnie 262k kontekstu, co na urządzeniu edge jest praktycznie proszeniem się o OOM, jeśli pozostawisz ustawienia default.

W przypadku rozważenia Mistral-7B jako model bazowy (bo Apache-2.0 i dojrzałość), pamiętaj, że karta modelu wprost mówi o braku mechanizmów moderacji. W robotyce oznacza to konieczność dodatkowej warstwy „policy + gating", niezależnie od tego, jak dobry jest model.

4.1. Optymalizacja wag, kontekstu i zasobów

Największe „dźwignie" na embedded to:

Kwantyzacja wag: dla llama.cpp standardem jest GGUF. Narzędzie quantize bierze model GGUF w wysokiej precyzji (F32/BF16) i konwertuje do formatów niskobitowych, redukując rozmiar i często przyspieszając inferencję; jakość można poprawiać użyciem imatrix (kalibracja).

Dla modeli Qwen istnieje też oficjalna ścieżka kwantyzacji GGUF obejmująca m.in. możliwość zastosowania skali AWQ.

Ograniczenie kontekstu: Qwen3.5 ma domyślny kontekst 262k; na robocie należy ustawić realny limit (np. 8k–16k) i świadomie zarządzać pamięcią rozmowy (streszczanie, pamięć epizodyczna). W przeciwnym razie KV-cache wciągnie RAM i doprowadzi do OOM.

llama.cpp: świetne dla GGUF + prosty serwer/CLI; ma praktyczne mechanizmy kwantyzacji i bogaty ekosystem.

MLC LLM: kompiluje/uruchamia modele na wielu backendach (CUDA/Vulkan/Metal; dokumentacja opisuje też konfigurację GPU i SDK). To bywa atrakcyjne na embedded, bo potrafi dać dobre osiągi bez ręcznego strojenia kernelami.

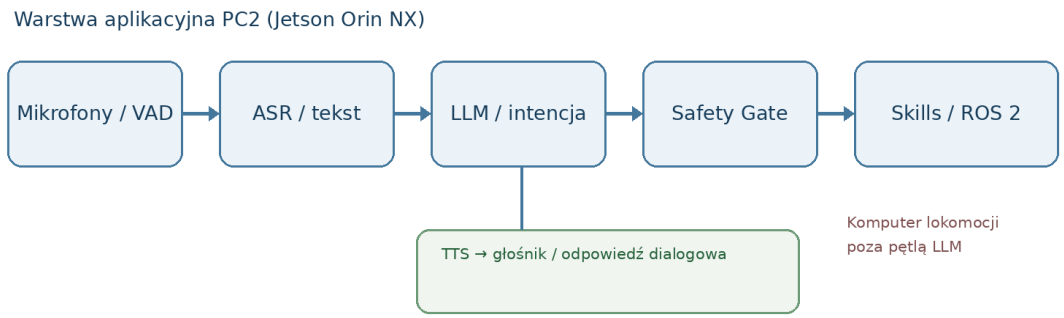
ONNX Runtime: szczególnie sensowne dla Phi- 3, bo karta modelu opisuje gotowe warianty ONNX/int4 i wsparcie cross- platform.

Strojenie na Jetson: Orin NX ma tryby mocy 10/15/25W i (zależnie od JetPack) tryby „SUPER"; w praktyce należy dobrać profil tak, by uniknąć throttlingu podczas dłuższych dialogów + percepcji. Sama społeczność llama.cpp raportuje, że na Orin NX 16GB w trybie wysokiej wydajności uzyskuje ~18 tok/s dla 7B Q4_0 na GPU, ale też wskazuje na wrażliwość na ustawienia i ograniczenia architektury (np. brak FP8 na Ampere).

Offload i degradacja jakości: PersonaPlex wprost dokumentuje --cpu-offload (dla braku VRAM), co jest analogiczną strategią do „layer offload" w innych runtime'ach – użyteczne jako bezpiecznik, ale kosztuje latencję.

Przy wdrożeniu wymagane jest jawne skonfigurowanie maksymalnej długości wejścia i odpowiedzi, polityki kompakcji historii rozmowy, poziomu kwantyzacji i limitu procesów współbieżnych. Kontekst 8–16 tys. tokenów stanowi zakres startowy do testów, a nie gwarancję stabilności pamięci w każdej konfiguracji.

5. Architektura referencyjna i separacja odpowiedzialności



Rysunek 1. Podział warstwy percepcji, LLM i deterministycznej warstwy wykonania. Odgałęzienie TTS nie może omijać bramki bezpieczeństwa dla poleceń sterujących.

Warstwa LLM powinna być projektowana jako komponent o miękkich wymaganiach czasowych (soft real- time) i nigdy nie może być jedynym źródłem prawdy o bezpieczeństwie ruchu. Zamiast tego LLM „wybiera intencję", a wykonanie odbywa się przez deterministyczne kontrolery/skills.

W Unitree G1 komunikacja z warstwą lokomocji i API odbywa się przez DDS (CycloneDDS), a integracja z ROS2 jest możliwa, bo ROS2 i CycloneDDS są kompatybilne na poziomie DDS. Jest to kluczowe, bo pozwala potraktować LLM jako kolejny węzeł (node) w grafie ROS2, ale bez naruszania wewnętrznej kontroli stabilizacji. W praktyce nie „przekazuje się" do LLM surowych danych czujników. Budujesz warstwę percepcji i streszczeń stanu, np.:

- Mowa: ASR zamienia audio → tekst + confidence + timestamps; dodatkowo VAD (wykrycie mowy) i diarization (opcjonalnie).
- Wizja: detekcje obiektów / semantyka sceny / „co robot widzi" w formie krótkich faktów; przy ograniczonym sprzęcie lepiej robić to lekkim modelem i podawać streszczenie do LLM, a nie uruchamiać ciężki VLM.
- Propriocepcja: pozycje stawów, prędkości, stan kontaktu stóp, tryb (normal/debug), ograniczenia, alarmy (przegrzanie, niskie napięcie).
- „World model" (pamięć robota): aktualny cel, kontekst zadania, mapka topologiczna, lista „dozwolonych czynności".

Zalecany wzorzec to ***tool- calling / function- calling***, ale z rygorem: LLM nie wysyła „surowych komend ruchu", tylko wywołuje zdefiniowane narzędzia o małym zakresie i weryfikowalnych parametrach. Przykładowy kontrakt narzędzi (idee):

- say(text, voice_id, priority)
- ask_clarification(question, options)
- navigate_to(pose, speed_profile)
- manipulate(object_id, grasp_type, constraints)
- set_posture(mode)
- stop_all(reason) (zawsze dozwolone)

Każde narzędzie przechodzi przez Safety Gate: walidacja schematu, limity prędkości/sił, strefy zakazane, kontrola stanu (czy robot jest stabilny), oraz timeout (jeśli nie dostaniemy potwierdzenia z kontrolera w X ms → awaryjne stop).

W dokumentacji G1 występuje pojęcie przełączenia w tryb debug (pełna kontrola użytkownika). To jest przydatne do badań, ale dla systemu opartego o LLM trzeba to traktować jako „tryb wysokiego ryzyka" i ograniczyć dostęp (np. tylko operator).

Dodatkowo, G1 ma domyślne ustawienia sieci i hasła, które są słabe; jeżeli robot ma mieć Internet, konieczna jest zmiana hasła i twarde reguły firewall.

Pojęcie stanu bezpiecznego nie powinno być utożsamiane z automatycznym zerowaniem wszystkich momentów napędów. Operację STOP i procedurę degradacji należy dostosować do trybów i gwarancji rzeczywistego kontrolera ruchu, tak aby nie destabilizować robota. Warstwa językowa nie przejmuje odpowiedzialności za stabilizację ani zabezpieczenia mechaniczne.

6. Kompletny lokalny potok audio

Zalecany projekt to pipeline strumieniowy, gdzie audio jest przetwarzane w stałych ramkach, a system wytwarza partial transcripts i partial audio w sposób ciągły. To minimalizuje „ciszę po wypowiedzi użytkownika" i poprawia UX. Kluczowe etapy:

• Audio capture: 16 kHz mono PCM (spójnie dla VAD/ASR; whisper.cpp i Silero VAD są często używane w 16 kHz).

• VAD: detekcja start/stop mowy, „hangover" (np. 200–400 ms ciszy zanim uznasz koniec wypowiedzi) oraz filtr krótkich szumów.

• ASR streaming:

• tryb „partial" co N ms (np. 200–500 ms),

• tryb „final" po końcu mowy (EoU – end-of-utterance).

• W whisper.cpp istnieje podejście demonstracyjne, gdzie narzędzie strumieniowe próbuje audio co 0.5 s i robi transkrypcję ciągłą – to jest dobry punkt odniesienia dla interwału partial.

• Vosk wprost promuje streaming API o niskiej latencji i małych modelach.

• Normalizacja tekstu: usuwanie wtrąceń, detekcja języka (jeśli potrzebujesz), opcjonalna interpunkcja. Whisper ma zwykle lepszą interpunkcję „z pudełka", bo jest multitask; Vosk często wymaga dodatkowej obróbki (np. model interpunkcji albo heurystyki).

• LLM: przetwarza tekst w schemacie rozmowy i produkuje albo odpowiedź tekstową, albo wywołania narzędzi (function-calling). W Qwen3.5 należy jawnie wyłączyć thinking mode w trybie dialogowym robota (zmniejsza latencję i ryzyko pętli generacji).

• TTS streaming: generuje audio w chunkach (np. 100–200 ms) i odtwarza praktycznie od razu; w trybie offline TTS (np. Riva offline) czekasz na całość i tracisz UX.

Współbieżność z wizją/propriocepcją:

• Wątek/proces audio powinien mieć priorytet czasowy (soft-RT): stałe ramki, minimalny jitter.

• LLM bywa najbardziej „burzliwy" obliczeniowo (GPU), więc sensowne jest uruchomienie ASR i TTS w trybie CPU-first (Vosk + Piper), a dopiero w trybie „high accuracy" przenoszenie ASR na GPU (faster-whisper).

• Na Orin NX limit mocy/termiki jest realny (10/15/25W), więc dobór profilu i równoległych obciążeń jest częścią projektu.

6.1. Detekcja VAD, buforowanie i znacznik czasu

VAD:

• Silero VAD: MIT, przenośny przez PyTorch i ONNX; repo podkreśla brak telemetrii i „zero strings attached".

• WebRTC VAD: bardzo lekki i szybki; py-webrtcvad to interfejs do WebRTC VAD, a dokumentacje społeczności wskazują licencję MIT dla wrappera i BSD-style dla oryginalnego kodu WebRTC.

W praktyce: WebRTC VAD jako „pierwszy filtr", Silero jako „drugi filtr" zwiększający jakość (redukcja fałszywych startów).

Diarization (opcjonalnie):

• Jeśli robot rozmawia zwykle z jedną osobą, diarization może być wyłączona (oszczędność CPU/GPU).

• Jeśli potrzebujesz rozróżniania mówców w rozmowie grupowej, rozważ wykorzystanie sherpa-onnx, który deklaruje wsparcie speaker diarization offline (onnxruntime) w tym samym SDK.

• Alternatywnie, utrzymuj prostą heurystykę „push-to-talk”/„wake word” (jeśli dostępne) i unikaj diarization w prototyp MVP.

Progi ufności i obsługa błędów:

• Dla ASR przechowuj: confidence (jeśli silnik udostępnia), długość wypowiedzi, SNR/poziom RMS, liczbę poprawek w partial.

• Wprowadź progi:

• CONF_HIGH: automatycznie wykonaj polecenie,

• CONF_MED: wykonaj confirm dialog („Czy chodziło Ci o…?”),

• CONF_LOW: poproś o powtórzenie i przejdź w tryb bezpieczny (brak ruchu, tylko dialog).

• Dla komend ruchu zawsze wymuszaj confirmation, jeśli komenda dotyczy ryzykownych działań (podniesienie obiektu blisko człowieka, szybki marsz, obrót w tłumie).

7. Rozpoznawanie mowy ASR: pięć wariantów

W robotyce użytkowej (komendy, rozmowa w ruchu) ASR musi łączyć cztery cechy: niska latencja (partial results), odporność na hałas i pogłos, stabilność strumieniowa oraz przewidywalny koszt zasobów. Model Whisper (OpenAI) jest silny jakościowo i trenowany w skali 680k godzin danych wielojęzycznych i multitask, co poprawia odporność m.in. na szum i akcenty. Jednocześnie Whisper bywa cięższy, a modele większe zwiększają latencję i utrudniają wdrożenia na urządzeniach o ograniczonych zasobach (to jest też obiekt badań pod kątem kwantyzacji i wdrażania). W praktyce zaleca się dwa tryby ASR:

• **Tryb jakości**: faster-whisper (CTranslate2) z modelem Whisper w int8/FP16, z streaming policy (partial), gdy GPU ma headroom.

• **Tryb deterministycznie lekki**: Vosk (małe modele, streaming), gdy GPU jest obciążone (wizja/LLM) albo gdy wymagane jest maksymalnie niskiej mocy/temperatury.

W tabeli rozdzielono „silnik” (engine) od „wag” (modele). W przypadku Whisper rozmiary modeli określono na podstawie danych z whisper.cpp (dysk/pamięć) jako praktycznej referencji wdrożeniowej.

ASR 1: Faster-Whisper (CTranslate2)

Licencja silnika	MIT (pakiet i repo)
Modele i wymagania	Wagi Whisper (tiny…large). W ekosystemie typowe konwersje do CTranslate2 z int8.
Kwantyzacja i formaty	CTranslate2 wspiera quant (m.in. int8_*). Dokumentacja opisuje mechanikę kwantyzacji.
Pamięć	RAM/VRAM zależne od wariantu i batch; int8 znacząco poprawia footprint. Faster-whisper deklaruje mniejsze zużycie pamięci i do ~4× szybkości vs openai/whisper, oraz poprawę dzięki 8-bit quant na CPU i GPU.
Tryb pracy i latencja	Dobre do streaming (przez otoczkę/politykę segmentacji) i offline. Repo wskazuje też ekosystem narzędzi streamingowych używających faster-whisper jako backend.
Frameworki i interfejsy	Python + CTranslate2 (C++), CUDA/CPU, łatwy Docker.
Dojrzałość i ryzyka	Bardzo wysoka adopcja (Whisper ecosystem), stabilny runtime produkcyjny (CT2).

ASR 2: whisper.cpp

Licencja silnika	MIT
Modele i wymagania	Modele Whisper w formacie ggml/gguf. Repo publikuje orientacyjne rozmiary: np. tiny ~75 MiB dysk/~273 MB RAM; base ~142 MiB/~388 MB; small ~466 MiB/~852 MB; medium ~1.5 GiB/~2.1 GB; large ~2.9 GiB/~3.9 GB.
Kwantyzacja i formaty	Wspiera integer quantization modeli ggml (np. Q5_0 itd.) i dostarcza narzędzia do kwantyzacji.
Pamięć	CPU-friendly, opcjonalnie CUDA. Dla małych modeli realne uruchomienie nawet bez GPU, ale przy jakości/hałasie lepiej mieć headroom. Ryzyko: raportowane przypadki bardzo długiego czasu ładowania na niektórych Jetsonach (cold-start bottleneck).

Tryb pracy i latencja	Ma tryb „stream” – przykład narzędzia próbkowania audio co 0.5 s i ciągłej transkrypcji (to daje naturalny rytm partial results).
Frameworki i interfejsy	C/C++ (ggml), proste wiązania; bardzo dobre do integracji w systemy wbudowane.
Dojrzałość i ryzyka	Bardzo wysoka dojrzałość (częste releasy), ogromna społeczność.

ASR 3: Vosk

Licencja silnika	Apache-2.0 (repo i strona)
Modele i wymagania	Modele „small” rzędu ~50 MB (deklaracja w PyPI) i większe dla jakości; dostępne modele PL.
Kwantyzacja i formaty	Brak „typowego” int4/int8 jak w transformerach; optymalizacja przez dobór małych modeli i dekodera.
Pamięć	Bardzo niski koszt pamięci (dziesiątki MB + narzut), dobra praca na CPU; nadaje się, gdy GPU jest zajęte.
Tryb pracy i latencja	Streaming API; PyPI opisuje „zero-latency response with streaming API” (w praktyce: szybkie partial results).
Frameworki i interfejsy	Wiele bindingów: Python/Java/C#/C++/Node i inne.
Dojrzałość i ryzyka	Wysoka dojrzałość wdrożeniowa (IoT/RPi), szeroka społeczność.

ASR 4: sherpa-onnx (STT)

Licencja silnika	Apache-2.0
Modele i wymagania	Zależne od wybranego modelu ONNX (CTC/RNNT itd.); projekt jest frameworkiem deploymentowym.
Kwantyzacja i formaty	ONNX Runtime (int8 możliwy zależnie od modelu i pipeline’u); często łatwiej o akceleracją niż w „czystym” PyTorch na edge.
Pamięć	Footprint zależny od modelu; duża wartość: jedna biblioteka obejmuje STT+TTS+VAD+diaryzację offline.
Tryb pracy i latencja	Wprost promuje działanie „without Internet connection” i offline aplikacje.
Frameworki i interfejsy	C++/Python i wiele platform embedded (wg opisu projektu).
Dojrzałość i ryzyka	Średnio-wysoka dojrzałość; mniejsza ekosystemowo niż Whisper, ale mocna jako „szwajcarski scyzoryk” ONNX.

ASR 5: NVIDIA Riva (opcjonalnie)

Licencja silnika	Komercyjny SDK/mikroserwisy NVIDIA
Modele i wymagania	Modele zależne od pakietu Riva; support matrix podaje też, które modele są wspierane na embedded.
Kwantyzacja i formaty	GPU-accelerated; liczne tryby offline/streaming (ASR, TTS).
Pamięć	Support matrix podaje liczby: np. ASR conformer-ctc ma GPU mem ~3.6 GB (streaming) do ~5.8 GB (offline) i jest oznaczony jako wspierany na embedded; z kolei whisper-large-v3 w Riva nie jest wspierany na embedded (wysokie VRAM).
Tryb pracy i latencja	Mikroserwisy pod real-time; dobre gdy akceptujesz cięższy stack i licencję NVIDIA.
Frameworki i interfejsy	gRPC, kontenery (typowo), GPU.
Dojrzałość i ryzyka	Wysoka dojrzałość enterprise; mniejsza elastyczność licencyjna/offline-update w porównaniu do MIT/Apache.

Dla języka polskiego pierwszorzędne znaczenie mają testy WER/CER w pogłosie, zakłóceniach, podczas lokomocji i z różnymi mówcami. Wartość marketingowa „do 4× szybciej” wymaga porównania z dokładnie określonym silnikiem referencyjnym, modelem, liczbą wątków i trybem kwantyzacji.

8. Synteza mowy TTS: pięć wariantów

W humanoidzie TTS ma inny profil niż ASR: ważniejsze są (a) „czas do pierwszego dźwięku” (TTST – time-to-start-talking), (b) stabilność streamingu i brak glitchy audio, (c) kontrola prozodii/intonacji w ograniczonych zasobach oraz (d) licencja głosu i zależności.

Z punktu widzenia wdrożenia zasadne jest zastosować dwie warstwy TTS:

- Neuralny TTS główny (naturalność): Piper (VITS-like) w ONNX, z polskim głosem (np. gosia/justyna/jarvis_wg_glos itd.).
- Fallback ultralekki: eSpeak NG (formant) – działa niemal zawsze, ale jest GPL-3.0+ (konsekwencje licencyjne dystrybucji).

Jeśli potrzebujesz ścieżki „enterprise” (np. twarde SLA, gotowe mikroserwisy, integracja GPU): Riva TTS oferuje streaming i offline inference, a support matrix wskazuje np. TTS fastpitch-hifigan jako wspierany na embedded z GPU mem ~2 GB.

TTS 1: Piper

Licencja silnika	Aktualny silnik OHF: GPL-3.0; historyczne rhasspy/piper: MIT (archiwalne). Głosy mają odrębne licencje.
Głosy i modele polskie	Oficjalne piper-voices (MIT) zawiera pl_PL głosy; np. pl_PL-gosia-medium ma ~63.2 MB (ONNX).
Kwantyzacja i formaty	ONNX (runtime). Jakość ma poziomy: x_low (5–7M parametrów), low (15–20M), medium (15–20M), high (28–32M).
Pamięć i zasoby	Zwykle CPU-friendly; RAM od setek MB (zależnie od modelu i buforów). Na Jetson często opłaca się trzymać TTS na CPU, zostawiając GPU dla LLM/wizji. (Estymacja inżynierska, do potwierdzenia benchmarkiem).
Latencja i strumieniowanie	Bardzo dobry do streamingu „prawie natychmiast”: TTST typowo dziesiątki–setki ms przy małych buforach; prędkość generacji zwykle >1× real-time na współczesnych CPU dla małych modeli (zależne od głosu).
Frameworki i integracja	Binarki arm64 dostępne; można używać jako proces CLI lub biblioteka.
Dojrzałość oraz ryzyka	Bardzo wysoka popularność. Ryzyko licencyjne w praktyce dotyczy zależności fonemizera: w dyskusjach wskazuje się użycie espeak-ng w piper-phonemizer (GPL), co trzeba przeanalizować pod kątem dystrybucji.

TTS 2: sherpa-onnx (TTS)

Licencja silnika	Apache-2.0 (silnik)
Głosy i modele polskie	Strona projektu publikuje listy polskich modeli TTS (np. vits-piper-pl_PL-gosia-medium itd.).
Kwantyzacja i formaty	ONNX Runtime; format zależny od modelu (często VITS-style).
Pamięć i zasoby	Podobne do Piper (bo często te same rodziny głosów); plus wartość: spójny stack offline (STT+TTS+VAD+diaryzacja).
Latencja i strumieniowanie	W zależności od implementacji potrafi robić streaming i offline; dobry gdy wymagane jest jeden SDK zamiast wielu komponentów.
Frameworki i integracja	C++/Python + ONNX Runtime.
Dojrzałość oraz ryzyka	Średnio-wysoka dojrzałość, mniejsza „mainstreamowość” niż Piper, ale mocne zalety integracyjne.

TTS 3: Coqui TTS (toolkit)

Licencja silnika	Kod: MPL-2.0 (komercyjnie dozwolone), modele mogą mieć inne licencje
Głosy i modele polskie	Możliwość użycia VITS/HiFi-GAN itd. (toolkit), ale polskie, gotowe, „bezpiecznie komercyjne” głosy trzeba zweryfikować per-model.
Kwantyzacja i formaty	W praktyce: PyTorch; kwantyzacja możliwa przez ONNX/TensorRT, ale nie jest „no-brainer” jak w Piper.

Pamięć i zasoby	Zależne od modelu; XTTS/klonowanie jest zwykle cięższe niż Piper i bywa słabe licencyjnie do produktu.
Latencja i strumieniowanie	Może mieć świetną jakość, ale TTST i zasoby zależą od architektury (VITS vs Tacotron2+vocoder).
Frameworki i integracja	PyTorch, Docker, pipeline'y treningowe.
Dojrzałość oraz ryzyka	Ryzyko: część popularnych modeli (np. XTTS-v2) jest pod CPML, który wprost ogranicza do użytku niekomercyjnego; w dyskusjach społeczność stwierdza brak ścieżki pozyskania licencji komercyjnej po zamknięciu firmy.

TTS 4: NVIDIA Riva TTS (opcjonalnie)

Licencja silnika	Komercyjny SDK NVIDIA
Głosy i modele polskie	Modele TTS w Riva; support matrix wskazuje TTS fastpitch-hifigan jako wspierany na embedded (GPU mem ok. 2 GB).
Kwantyzacja i formaty	GPU-accelerated; tryb streaming i offline.
Pamięć i zasoby	Wymagania pamięci jawne w support matrix.
Latencja i strumieniowanie	Streaming: audio zwracane w trakcie syntezy; offline: zwraca po wygenerowaniu całości (większa przepustowość).
Frameworki i integracja	Mikroserwisy (typowo kontenery), gRPC.
Dojrzałość oraz ryzyka	Bardzo dobra jakość i wydajność, ale koszty operacyjne (kontenery, wersjonowanie) i licencja.

TTS 5: eSpeak NG (fallback)

Licencja silnika	GPL-3.0+
Głosy i modele polskie	Języki liczne, w tym europejskie; jakość „robotyczna", ale stabilność i niski koszt zasobów.
Kwantyzacja i formaty	Nie dotyczy (synteza formantowa).
Pamięć i zasoby	Bardzo niski koszt CPU/RAM, praktycznie zawsze działa.
Latencja i strumieniowanie	Natychmiastowy start mówienia (TTST minimalny).
Frameworki i integracja	Biblioteki/CLI, prosta integracja.
Dojrzałość oraz ryzyka	Kluczowe ryzyko: GPL w produkcie wbudowanym (dystrybucja).

Stan licencyjny: aktywnie utrzymywana gałąź Piper (OHF-Voice/piper1-gpl) jest udostępniana na GPL-3.0; historyczny projekt rhasspy/piper był objęty MIT i jest archiwalny. Licencję głosów należy sprawdzać osobno dla konkretnego MODEL_CARD. Ocena prawna dotyczy sposobu wykorzystania i dystrybucji, a nie samego lokalnego wykonania programu.

9. Synchronizacja mowy i budżet czasowy

Definicje:

- TTFT (Time To First Token): czas od wysłania promptu do LLM do otrzymania pierwszego tokena.
- TTST (Time To Start Talking): czas od podania tekstu do TTS do rozpoczęcia odtwarzania dźwięku (pierwszy chunk PCM).
- Dla UX robota interesuje Cię „Time to First Audio Response" = (VAD end) + (ASR finalize) + (LLM TTFT) + (TTS TTST).

Na Orin NX 16GB można przyjąć **realistyczny budżet projektowy** (do weryfikacji w testach HIL):

- VAD/segmentacja: 10–50 ms obliczeń na sekundę audio (zwykle <5 ms na ramkę 20–30 ms; zależy od implementacji).

• ASR finalize: ~0.2–1.2 s dla krótkich komend (zależnie od silnika; Vosk zwykle szybciej i bardziej „online”, Whisper wolniej, ale jakościowo lepiej).

• LLM TTFT: ~0.2–0.8 s (dla 2–4B int4, non-thinking, krótki kontekst – estymacja). Jako punkt odniesienia dla generacji: w dyskusji llama.cpp dla Orin NX 16GB podano ok. 9 tok/s dla generacji (test tg128) na CPU dla 7B Q4_0, co wyznacza konserwatywny dolny pułap; GPU zwykle poprawi wynik, ale zależy od runtime i konfiguracji.

• TTS TTST (Piper): ~0.05–0.3 s przy streamingu i krótkich buforach (estymacja), zależne od „quality level” i długości tekstu; poziomy jakości i rząd wielkości parametrów są jawne.

Wypadkowo: 0.7–2.5 s do pierwszego dźwięku odpowiedzi dla typowych poleceń głosowych jest realnym celem.

Miary czasu należy rozdzielić: opóźnienie VAD mierzy detekcję końca wypowiedzi, ASR-finalize czas ustalenia ostatecznej transkrypcji, TTFT pierwszego tokena LLM, a TTST czas do pierwszej próbki odtwarzanego głosu. Metriką końcową jest czas od końca wypowiedzi użytkownika do pierwszego dźwięku odpowiedzi, mierzony monotonicznym zegarem na wspólnym urządzeniu.

10. ROS 2 i DDS: szczegółowe kontrakty komunikacyjne

Ponieważ G1 używa DDS w warstwie komunikacji (a przewodniki integracyjne wskazują CycloneDDS i kompatybilność z ROS2), najczystsze jest zrobienie audio pipeline jako zestawu węzłów ROS2 publikujących/odbierających zdarzenia, a nie „czarnych skryptów”. Proponowane wiadomości ROS2 (przykładowe, minimalistyczne):

```
# pcdb_msgs/msg/AudioFrame.msg
builtin_interfaces/Time stamp
uint32 sample_rate_hz
uint8  channels
string encoding       # "pcm_s16le"
uint8[] data
# pcdb_msgs/msg/VadEvent.msg
builtin_interfaces/Time stamp
string event          # "speech_start" | "speech_end" | "speech_continue"
float32 speech_prob   # 0..1 (jeśli dostępne)
# pcdb_msgs/msg/AsrPartial.msg
builtin_interfaces/Time stamp
string text
float32 confidence
bool is_final
# pcdb_msgs/msg/DialogTurn.msg
builtin_interfaces/Time stamp
string role           # "user" | "assistant" | "system"
string text
float32 confidence    # dla user; dla assistant = 1.0
# pcdb_msgs/msg/TtsChunk.msg
builtin_interfaces/Time stamp
uint32 sample_rate_hz
uint8 channels
string encoding       # "pcm_s16le"
bool is_last
uint8[] data
W warstwie LLM zalecam kontrakt w stylu function-calling, ale tylko do zdefiniowanych „skills”:
{
"name": "call_skill",
"description": "Wywołuje bezpieczny skill robota z walidacją parametrów i limitami.",
"parameters": {
"type": "object",
"properties": {
"skill_id": { "type": "string", "enum": ["say", "navigate_to", "set_posture", "stop_all", "manipulate"] },
"args": { "type": "object" },
"requires_confirmation": { "type": "boolean" }
},
"required": ["skill_id", "args"]
}
}
```

Następnie „Skill Executor” (poza LLM) waliduje skill_id, sprawdza limity i wykonuje dopiero bezpieczny zestaw komend do SDK/sterowania (DDS). To podtrzymuje zasadę: LLM nie steruje bezpośrednio ruchem.

Powyższe schematy stanowią kontrakty referencyjne. W implementacji należy uzupełnić je o identyfikator sesji, identyfikator wiadomości, identyfikator korelacyjny oraz zdefiniowaną semantykę ponawiania, deduplikacji i anulowania. Pole confidence wolno interpretować jako skalibrowane prawdopodobieństwo wyłącznie po stosownej kalibracji; wiele silników ASR dostarcza oceny nieskalibrowane.

11. Przetwarzanie offline i reprodukowalność instalacji

Ponieważ Unitree wskazuje „Storage 2T” w konfiguracji komputera developerskiego, dostępna jest przestrzeń na utrzymanie wielu wariantów modeli (ASR/TTS/LLM) i rollout A/B. Jednocześnie Orin NX wspiera NVMe przez PCIe, co jest preferowane dla dużych modeli i szybkiego ładowania. Praktyczna struktura offline:

- /opt/models/asr/whisper/base-int8-ct2/…
- /opt/models/asr/vosk/pl-small/…
- /opt/models/tts/piper/pl_PL-gosia-medium/…
- /opt/models/llm/qwen35-4b-int4/…
- /opt/models/MANIFEST.json (hashy SHA-256, wersje, licencje, źródła)

W trybie offline kluczowe są „manifesty" i spójny mechanizm „model packów", bo nie można w locie pobierać brakujących tokenizatorów czy plików config.

W systemach wbudowanych cold-start jest częstą pułapką. Dla whisper.cpp raportowano przypadki, w których „first load time" na Jetsonach (AGX Xavier/Orin) był rzędu wielu minut i stawał się bottleneckiem produkcyjnym. Nawet jeśli to nie wystąpi w docelowej konfiguracji, w robocie i tak warto:

- uruchamiać ASR i TTS jako długowieczne demony (systemd/contener) i utrzymywać modele w pamięci,
- stosować „warm-pool" (1–2 procesy ASR gotowe do pracy) zamiast odpalania procesu per zapytanie,
- ograniczyć liczbę wariantów modeli aktywnych jednocześnie (swapping kontrolowany).

Swapping jest potrzebny (tryb jakości vs low-power), ale musi być deterministyczny:

- ASR: przełączaj silnik, nie tylko model (np. Vosk ↔ faster-whisper) w zależności od obciążenia GPU/temperatury.
- TTS: przełączaj głos (Piper: różne quality levels) i fallback na eSpeak NG przy przeciążeniu.
- LLM: przełączaj 2B ↔ 4B w zależności od zadania i czasu reakcji.

Dopuszcza się następujące strategie:

- „Jedna instancja na raz" + szybki restart (proste, ale ryzyko przerwy),
- lub „dwóch rezydentów" (np. Vosk + faster-whisper, Piper + eSpeak) i routing (bardziej niezawodne, ale koszt RAM).

NVIDIA publikuje przewodniki o uruchamianiu kontenerów na Jetson; Docker jest integralną częścią ekosystemu JetPack, a na Jetsonach używa się NVIDIA Container Runtime.

Tryb offline oznacza:

- budujesz obrazy ARM64 w CI (lub na stacji roboczej z buildx),
- eksportujesz je jako tar (docker save),
- przenosi się nośnikiem do robota i importuje (docker load),
- a same modele dostarczasz jako „model packs" z podpisem i hashami (patrz sekcja wdrożeniowa niżej).

Uwaga specyficzna dla G1: w przewodnikach sugeruje się, że Jetson jest na dedykowanej płycie Unitree i wymaga właściwego BSP (nie flashować „obrazami zewnętrznymi"), więc aktualizacje systemu trzeba planować „ostrożnie i minimalnie inwazyjnie".

Każdy model pack musi zawierać komplet zależności runtime, tokenizer, plik konfiguracji, wariant architektury, precyzję wag, identyfikator modelu, sumy SHA-256, warunki licencyjne i wynik walidacji zgodności. Współdzielenie tej samej nazwy wydania między nieidentycznymi wagami lub konfiguracjami uniemożliwia odtworzenie pomiaru.

12. Optymalizacja pod Jetson i degradacja jakości

Kwantyzacja i formaty:

- Faster-whisper + CTranslate2: korzystaj z int8 (na CPU i GPU) tam, gdzie akceptujesz minimalny spadek jakości; sam projekt wskazuje, że 8-bit quant zwiększa efektywność.
- whisper.cpp: używaj quant modeli ggml (repo opisuje wsparcie kwantyzacji i narzędzie quantize).
- Piper / ONNX: dobieraj „quality level" pod budżet TTST i naturalność; parametryzacja (x_low…high) jest jawna, co ułatwia tworzenie profili „low-power" i „high-quality".

Streaming i batching:

- W ASR priorytetem jest streaming i małe porcje, a nie duże batche (batche zwiększają throughput, ale pogarszają interakcyjność).
- W TTS w praktyce generuj audio w chunkach (np. 100–200 ms) i odtwarzaj natychmiast; tryb offline-only (generuj wszystko, potem odtwórz) jest UX-owo gorszy. Riva TTS dokumentuje różnicę: offline zwraca audio dopiero po wygenerowaniu całej sekwencji, a streaming zwraca podczas syntezy.

Podział CPU/GPU:

• Jeśli GPU jest krytyczny dla LLM/wizji, ustaw ASR na Vosk (CPU) i TTS na Piper (CPU), co stabilizuje latencję dialogu.

• Gdy GPU ma headroom: przenieś ASR na faster-whisper (CUDA) dla jakości w hałasie.

Profil mocy i throttling:

• Orin NX 16GB ma tryby 10W/15W/25W; w robocie „długie rozmowy" potrafią doprowadzić do throttlingu, więc profil mocy i chłodzenie muszą być częścią walidacji.

Ryzyko specyficzne dla audio w robocie to „złe rozpoznanie → zła akcja". Dlatego:

• Stosuj progi ufności i potwierdzenia opisane wcześniej.

• Dla komend „ruchowych" zawsze stosuj regułę: jeśli ASR nie ma wysokiej ufności, robot nie wykonuje ruchu, tylko dopytuje.

• Wprowadź „canned responses" i zachowania regułowe (np. „Nie zrozumiałem. Powtórz.") w przypadkach:

• przekroczenia time-out LLM,

• awarii ASR/TTS,

• przeciążenia termicznego (temperatura).

• Dodatkowo rozważ „tryb ciszy": jeśli VAD wykrywa hałas/krzyki (wysokie RMS), robot ogranicza ruch i prosi o podejście bliżej.

Wybór wariantu ASR/LLM/TTS powinien zależeć od mierzonego obciążenia, a nie tylko od deklarowanej klasy zadania. Profil regulujący obciążenie może być aktualizowany w fazie bezczynności, aby uniknąć niekontrolowanych restartów modeli podczas wykonywania operacji fizycznych.

13. Bezpieczeństwo działania, prywatność i licencje

Prywatność:

• W trybie offline dostępna jest przewagę: audio nie opuszcza robota. Ale nadal należy zdecydować, czy nagrywasz i logujesz próbki do debugowania.

• Polityka rekomendowana: domyślnie brak trwałego zapisu surowego audio; jeśli włączasz nagrania do debug, to:

• rotacja (np. ostatnie 10–30 minut),

• szyfrowanie na dysku,

• dostęp tylko w trybie serwisowym,

• anonimizacja metadanych.

Bezpieczeństwo infrastruktury:

• Unitree przewodniki zwracają uwagę na ryzyko domyślnych ustawień dostępu (hasła) przy podłączeniu do Internetu; nawet offline warto zmienić domyślne hasła i ograniczyć usługi.

• API między modułami (ASR/LLM/TTS) powinno być lokalne (localhost/ROS2), bez nasłuchiwania na zewnętrznych interfejsach.

Licencje jako element bezpieczeństwa wdrożenia:

• Piper ma MIT, ale w społeczności wskazuje się, że część pipeline'u fonemizacji używa espeak-ng (GPL) i trzeba to sprawdzić w docelowej konfiguracji zależności.

• eSpeak NG jest GPL-3.0+ i wbudowanie go w produkt może wymuszać określone obowiązki dystrybucyjne.

• Coqui TTS: kod (toolkit) jest MPL-2.0 i dopuszcza komercję, ale popularne wagi XTTS są pod CPML i wprost ograniczają użycie do niekomercyjnego (oraz społeczność wskazuje brak ścieżki legalnego pozyskania licencji komercyjnej).

Najważniejsze ryzyko LLM w robotyce to nie „halucynacje tekstowe", tylko halucynacje akcji. Z tego powodu:

• LLM nie może mieć bezpośredniego kanału do momentów/sił; tylko do prymitywów z limitami.

• Wymuszaj „bezpieczny język komend" i schematy. Odpowiedź niezgodna ze schematem = odrzucenie + prośba o doprecyzowanie.

• Wprowadź watchdog: jeśli LLM nie odpowiada w czasie, system przechodzi w stan „bezpieczny" i dalej działa deterministycznie.

• Ogranicz sieć: ponieważ robot ma statyczne IP i brak DHCP/routera, każde „wystawienie" API LLM wymaga świadomej konfiguracji podsieci; dokumentacja ostrzega o słabym haśle domyślnym i zaleca jego zmianę przy dostępie do Internetu.

• Qwen3.5: Apache- 2.0 (wagi i konfiguracje w HF). To zwykle najprostsza ścieżka do komercyjnego wdrożenia lokalnego.

• Phi- 3- mini: MIT. Bardzo „czysta" licencja, dobra do produktów.

• Mistral- 7B- Instruct- v0.2: Apache- 2.0, ale brak moderacji wymusza własne guardrails (to bardziej kwestia bezpieczeństwa niż prawa).

• Llama- 3.2: licencja społecznościowa Meta + Acceptable Use Policy; technicznie mocna opcja, ale prawnie bardziej złożona (szczególnie przy redystrybucji artefaktów).

• Gemma: niestandardowe Terms of Use Google; często wymaga akceptacji warunków (gating) – to zwiększa koszty operacyjne (zgodność, procesy, audyt).

• PersonaPlex: wagi są gated i objęte NVIDIA Open Model License; karta HF wskazuje konieczność akceptacji warunków i udostępnienia danych kontaktowych. To może być problematyczne w projektach, które wymagają w pełni „niezależnej" dystrybucji.

• microGPT: to kod edukacyjny (art project), a nie standardowy „model do wdrożenia"; z punktu widzenia compliance i ryzyk produktu zwykle odpada.

System dialogowy stanowi komponent wspomagający. W kontekście bezpieczeństwa fizycznego wymagane są niezależne ograniczenia prędkości, siły i przestrzeni pracy, priorytetyzacja operatora, niezawodna obsługa zatrzymania oraz ograniczenie uprawnień narzędzi, niezależne od modelu LLM.

14. Projekt badań i protokół HIL

Do oceny ASR w języku polskim (offline, command-and-control + rozmowa) przyjęto trzy klasy danych:

• Mozilla Common Voice – Polish: oficjalne datasheet wskazuje ok. 187.97 h nagrań (175.72 h zwalidowanych), ~3455 mówców w wersji 24.0; Fundacja Mozilla komunikuje, że dataset jest udostępniany na licencji CC0.

• Spokes (CLARIN-PL/PELCRA): korpus rozmów nieformalnych; w CLARIN wiki podano skalę ~1024 h i 658k wypowiedzi; to dobre do testów w dialogu spontanicznym (VAD, segmentacja, overlap).

• SpokesBiz: korpus konwersacyjny >650 h, z diarization i ręczną anotacją interpunkcji i casing – szczególnie cenny do walidacji diarization i jakości końcowej transkrypcji w rozmowie.

Do TTS (polski) i/lub walidacji jakości syntezy:

• M-AILABS Polish: opis korpusu wskazuje ~53.83 h, ~26,322 wypowiedzi, 2 mówców; to może służyć do trenowania/porównywania lub dopasowania lekkich głosów (choć 2 mówców ogranicza różnorodność).

• Dla gotowych głosów Piper: korzystaj z oficjalnych pl_PL głosów lub utrzymywanych zestawów polskich modeli; w ekosystemie wprost występują polskie modele „vits-piper-pl_PL-…".

Metryki ASR:

• WER/CER na Common Voice PL i na własnym zestawie komend robota (ok. 200–1000 fraz).

• Real-Time Factor (RTF) i „partial stability" (liczba zmian w partial tekstu na sekundę).

• Odporność na hałas: testy w ruchu, z wentylatorem, w hali, w pobliżu muzyki.

Metryki TTS:

• TTST, RTF syntezy, dropouty audio (glitch count), poziom głośności i clipping.

• Subiektywne MOS (min. 10–20 uczestników), ale w wersji robota ważniejsze jest „zrozumiałość w hałasie" niż „radiowa naturalność".

Metryki end-to-end:

• Time-to-First-Audio-Response (od końca mowy użytkownika do pierwszego dźwięku odpowiedzi).

• Task success rate (czy robot wykonał prawidłowy skill / czy odmówił przy niepewności).

• Safety events: liczba wymuszonych potwierdzeń, liczba „stop_all", liczba odrzuconych tool-calls przez Safety Gate.

14.1. Tabela budżetu zasobów

Składnik 1: VAD

Wariant referencyjny	Silero VAD ONNX + opcjonalnie WebRTC VAD
Pamięć	dziesiątki MB lub mniej (model mały), CPU

Wydajność lub opóźnienie	opóźnienie na ramkach audio zwykle bardzo niskie; decyzje co 10–30 ms
Komentarz	Silero MIT i przenośność ONNX; WebRTC bardzo lekki.

Składnik 2: ASR

Wariant referencyjny	Tryb „quality”: faster-whisper int8
Pamięć	zależne od modelu; w praktyce od setek MB do kilku GB (large)
Wydajność lub opóźnienie	celem jest near-real-time dla krótkich komend; projekt deklaruje do ~4× szybciej i mniej pamięci vs openai/whisper, plus int8 na CPU/GPU
Komentarz	Preferuj tiny/base/small dla robota (latencja), large tylko „offline batch”.

Składnik 3: ASR

Wariant referencyjny	Tryb „light”: Vosk small PL
Pamięć	~50 MB dla małych modeli (deklaracja)
Wydajność lub opóźnienie	partial w czasie rzeczywistym; finalize zwykle setki ms po końcu mowy (zależne od akustyki)
Komentarz	Bardzo dobry, gdy GPU jest zajęte.

Składnik 4: LLM

Wariant referencyjny	Qwen3.5-2B/4B int4, non-thinking
Pamięć	typowo 1–3+ GB na wagi + KV cache (zależnie od limitu kontekstu)
Wydajność lub opóźnienie	TTFT ~0.2–0.8 s (cel); tok/s zależne od runtime. Konserwatywny punkt odniesienia: llama.cpp na Orin NX 16GB raportuje ok. 9 tok/s generacji na CPU dla 7B Q4_0.
Komentarz	Ogranicz kontekst i wyłącz thinking mode.

Składnik 5: TTS

Wariant referencyjny	Piper pl_PL medium
Pamięć	dziesiątki MB na model (np. ~63 MB)
Wydajność lub opóźnienie	TTST typ. 50–300 ms (cel); prędkość zwykle >1× real-time dla prostych głosów (do weryfikacji)
Komentarz	Modele mają poziomy jakości i parametry (x_low…high).

Składnik 6: TTS fallback

Wariant referencyjny	eSpeak NG
Pamięć	bardzo mały, CPU
Wydajność lub opóźnienie	praktycznie natychmiastowy TTST
Komentarz	GPL-3.0+.

W protokole walidacji należy raportować odsetek wykonań, liczbę powtórzeń, medianę i percentyle opóźnienia, wielkość kontekstu, temperaturę, obciążenie CPU/GPU, wersje modeli i runtime, a także czy robot wykonywał równolegle percepcję i lokomocję. Wyniki z trybu idle nie są równoważne wynikom przy współbieżnym ruchu i analizie obrazu.

15. Wdrażanie, monitorowanie i wycofywanie zmian

Ponieważ development computer jest elementem systemu robota i ma ograniczenia BSP, zaleca się podejście „minimalnie inwazyjne”:

Pakowanie: - Preferuj kontenery ARM64 z NVIDIA Container Runtime (na Jetson), ale nie zakładaj, że zawsze da się użyć gotowych obrazów (różnice JetPack). JetPack 6 to Ubuntu 22.04 + kernel 5.15, ale G1 bywa spotykany też na JetPack 5.1.x/Ubuntu 20.04 – to wpływa na binaria i zależności. - Modele duże trzymaj na szybkim nośniku (NVMe). W datasheet Orin NX wspomina się o wsparciu dla zewnętrznego NVMe przez PCIe, co jest praktycznie jedyną sensowną drogą na duże GGUF.

Cross- kompilacja i build: - Dla runtime'ów wymagających kompilacji (np. llama.cpp, MLC) buduj artefakty w CI z docker buildx na ARM64. - W przypadku rozważenia vLLM: jest wsparcie dla ARM64 CPU (NEON), ale zwykle brak gotowych wheel'i – trzeba budować ze źródeł i liczyć się z większym narzutem.

Testy:

- Testy jednostkowe: walidacja schematów narzędzi, timeouts, parser odpowiedzi LLM.
- Testy integracyjne: replay sesji (nagrania audio+telemetria) → oczekiwane narzędzia i brak komend zakazanych.
- Testy „hardware-in-the-loop": uruchamiaj na robocie tylko w trybie bezpiecznym (ograniczone prędkości, miękkie stany), z fizycznym E-stopem/operator mode.

Metryki LLM: tok/s, TTFT (time- to- first- token), zużycie RAM/GPU, OOM count, długości odpowiedzi.

Metryki robota: temperatury, napięcie, watchdog DDS/ROS2, liczba „stop_all".

Na Jetson można skorzystać z gotowych usług monitoringu/integracji API gateway (NVIDIA Jetson Platform Services pokazuje wzorzec: kontener, REST API, integracja z monitoringiem i ingress).

Wersjonuj modele i runtime osobno (np. llm-runtime:vX, model-pack:qwen35-4b-q4km-vY).

W przypadku wzrostu błędów/jitteru: automatyczny powrót do mniejszego modelu (2B) albo do trybu „bez LLM" (reguły + predefiniowane komendy).

Model CI/CD (offline-compatible):

• Buduj „model packs" jako archiwa (np. .tar.zst) zawierające: wagi, configi, tokenizatory, licencje, manifest SHA-256.

• Podpisuj pakiety (np. mechanizmem podpisu artefaktów w infrastrukturze wdrożeniowej) i waliduj podpis na robocie przed instalacją.

• Instalacja offline: nośnik USB / dysk zewnętrzny / fizyczne repo w sieci zamkniętej.

Rollback:

• Trzymaj co najmniej 2 wersje każdego model packa: current i previous.

• Runtime powinien umieć przełączyć się na poprzedni pakiet bez reflasha systemu (atomowy symlink + restart usługi).

Monitoring audio pipeline:

• VAD: liczba wykryć, false-starts (heurystycznie), średnia długość segmentów.

• ASR: RTF, średnia confidence, odsetek CONF_LOW/CONF_MED.

• LLM: TTFT, tok/s, odsetek odmów i odrzuconych tool-calls przez Safety Gate.

• TTS: TTST, dropouty, opóźnienie buforów audio.

Logowanie prywatnościowe:

• Loguj metryki i skróty (hash) transkrypcji, a nie surowe audio.

• Jeśli przechowujesz audio do debug: szyfrowanie + krótka retencja + tryb serwisowy.

Najważniejsze decyzje architektoniczne, które zwykle przesądzają o sukcesie

• strumieniowanie (partial ASR + streaming TTS) zamiast trybu „batch",

• twarde progi ufności i dialog potwierdzeń przed akcją,

• rozdział zasobów CPU/GPU między audio i LLM/wizję,

• utrzymywanie „ciepłych" procesów i kontrola cold-start,

• jawne wersjonowanie i podpisy pakietów modeli na aktualizacje offline.

16. Macierz ryzyka wdrożenia

Ryzyko 1: OOM / niestabilna pamięć (KV-cache, długi kontekst)

Prawdopodobieństwo	Wysokie (domyślne 262k w Qwen3.5)

Skutek	Wysoki
Objawy i wskaźniki	zabicia procesu, restart, freeze dialogu
Działania ograniczające	limit kontekstu 8k–16k; streszczanie; stałe limity tokenów; watchdog

Ryzyko 2: Throttling termiczny → jitter latencji

Prawdopodobieństwo	Średnie–wysokie
Skutek	Wysoki
Objawy i wskaźniki	skoki TTFT/tok/s, zrywanie dialogu
Działania ograniczające	profil mocy (nvpmodel), limity FPS wizji, priorytety GPU, fallback do 2B

Ryzyko 3: Niebezpieczne akcje z LLM (hallucinated tool calls)

Prawdopodobieństwo	Średnie
Skutek	Krytyczny
Objawy i wskaźniki	ruch w złym kierunku, nieprzewidywalne gesty
Działania ograniczające	skill- gating, walidacja schematów, limity prędkości/sił, operator mode, „stop_all”

Ryzyko 4: Ryzyko sieciowe (domyślne hasło, ekspozycja podsieci robota)

Prawdopodobieństwo	Średnie
Skutek	Wysoki
Objawy i wskaźniki	nieautoryzowany dostęp, sterowanie
Działania ograniczające	zmiana haseł, firewall, brak publicznego API, segmentacja, VPN

Ryzyko 5: Złożoność BSP na G1 (nie flashować, zależności JetPack)

Prawdopodobieństwo	Średnie
Skutek	Wysoki
Objawy i wskaźniki	brick systemu, brak sterowników
Działania ograniczające	minimalna inwazyjność, kontenery, snapshoty, polityka aktualizacji

Ryzyko 6: Compliance/licencje (gating, restrykcje dystrybucji)

Prawdopodobieństwo	Średnie
Skutek	Średni–wysoki
Objawy i wskaźniki	blokada prawna release’u
Działania ograniczające	preferuj Apache- 2.0/MIT; audyt licencji i artefaktów; unikać gated wag w core

Ryzyko 7: Brak moderacji w modelu (np. Mistral)

Prawdopodobieństwo	Wysokie
Skutek	Średni–wysoki
Objawy i wskaźniki	toksyczne treści, błędne instrukcje
Działania ograniczające	dodatkowy filtr treści + polityka dialogu; separacja „mowy” od „akcji”

Macierz ryzyka ma charakter jakościowy. Przypisanie klas prawdopodobieństwa nie wynika ze statystycznego pomiaru awaryjności G1. Powinno zostać zweryfikowane w testach HIL, a następnie aktualizowane na podstawie dzienników zdarzeń i wyników audytów.

17. Plan wdrożenia i podział kompetencji

Bardzo zgrubnie:

- prototyp MVP tekstowy (LLM + tool- calling + kilka skills + safety gate): 4–6 tygodni.
- Dodanie ASR/TTS i spójnej synchronizacji dialogu: +2–4 tygodnie.
- Wersja „produkcyjna" z monitoringiem, testami HIL, rollback i twardymi procedurami bezpieczeństwa: +4–8 tygodni.
- PersonaPlex jako full- duplex voice agent (jeśli w ogóle na G1): +6–10 tygodni i wysokie ryzyko, bo model jest celowany w większe GPU, a repo przewiduje co prawda offload, ale kosztem latencji.

Etap	Zakres prac / kryterium zakończenia
E1. Inwentaryzacja	Wersje PC2, JetPack, BSP, sterowników, interfejsów audio i kontrolera; kopia konfiguracji, pomiar pamięci w stanie spoczynku.
E2. Laboratorium modeli	Porównanie 2B/4B/9B i Phi-3: TTFT, tok./s, pamięć, temperatura, lokalne wywołania narzędzi; wybór konfiguracji referencyjnej.
E3. Kontrakty i bramka	Schematy ROS 2/DDS, katalog skills, walidacja parametrów, blokady ruchu, watchdog, testy negatywne bez aktuacji.
E4. Lokalna mowa	Dwa tryby ASR, VAD, TTS, odtwarzanie strumieniowe, barge-in i polityka potwierdzania komend.
E5. Offline i monitoring	Podpisane pakiety modeli, manifesty SHA-256, odtwarzalne środowisko, monitor CPU/GPU i rollback.
E6. HIL i pilot	Testy z operatorem i E-stop, odporność na hałas, przeciążenie i utratę połączeń, raportowanie P95 i zdarzeń safety.

18. Ograniczenia i wnioski

Najbardziej uzasadnionym punktem wyjścia pozostaje lokalny model językowy klasy 2–4 mld parametrów z kontrolowaną długością kontekstu, dwutorowy ASR (jakościowy i lekki) oraz kompaktowy polski TTS. Rekomendacja ma charakter projektowy: przewaga Qwen 3.5 nad Phi-3 lub innymi modelami nie została ustalona we wspólnym eksperymencie wykonanym na badanym robocie.

W kontekście robota humanoidalnego głównym wyzwaniem jest integracja: stabilny budżet pamięci, współbieżność z percepcją, reżim termiczny, skuteczne rozpoznanie komend po polsku i odporność na błędy narzędzi. Wydajność LLM nie może być poprawiana kosztem sterownika odpowiedzialnego za bezpieczeństwo ruchu.

Zaleca się walidację architektury warstwowo: najpierw replay audio i symulowane wywołania skills, następnie integracja z robotem bez aktuacji, wreszcie próby HIL z operatorem. Każdy wynik eksperymentalny powinien być łączony z wersją modelu i konfiguracją sprzętową. Dopiero taka procedura pozwoli sformułować empiryczne twierdzenia o skuteczności, opóźnieniu i niezawodności.

19. Literatura i dokumentacja techniczna

[1] Unitree G1: Overview, development computing unit. https://docs.quadruped.de/projects/g1/html/g1_overview.html

[2] Weston Robot: G1 Development Guide. https://docs.westonrobot.com/tutorial/unitree/g1_dev_guide/

[3] NVIDIA: Jetson Orin NX Series Data Sheet. https://developer.nvidia.com/downloads/jetson-orin-nx-series-data-sheet

[4] Qwen: Qwen3.5-2B, model card. https://huggingface.co/Qwen/Qwen3.5-2B

[5] Qwen: Qwen3.5-4B, model card. https://huggingface.co/Qwen/Qwen3.5-4B

[6] Qwen: Qwen3.5-9B, model card. https://huggingface.co/Qwen/Qwen3.5-9B

[7] Microsoft: Phi-3-mini-4k-instruct. https://huggingface.co/microsoft/Phi-3-mini-4k-instruct

[8] Meta: Llama 3.2 3B Instruct. https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct

[9] Mistral 7B Instruct. https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2

[10] Gemma: Terms of Use. https://ai.google.dev/gemma/terms

[11] NVIDIA: PersonaPlex repo. https://github.com/NVIDIA/personaplex

[12] llama.cpp: runtime i kwantyzacja. https://github.com/ggml-org/llama.cpp

[13] Faster-whisper: ASR. https://github.com/SYSTRAN/faster-whisper

[14] whisper.cpp: ASR. https://github.com/ggml-org/whisper.cpp

[15] Vosk: offline speech recognition. https://alphacephei.com/vosk/

[16] sherpa-onnx: offline ASR/TTS. https://github.com/k2-fsa/sherpa-onnx

[17] Silero VAD. https://github.com/snakers4/silero-vad

[18] Piper: aktualnie rozwijane wydanie. https://github.com/OHF-Voice/piper1-gpl

[19] Piper: repozytorium historyczne. https://github.com/rhasspy/piper

[20] eSpeak NG. https://github.com/espeak-ng/espeak-ng

[21] NVIDIA Riva. https://docs.nvidia.com/deeplearning/riva/

[22] ROS 2: dokumentacja. https://docs.ros.org/