STUDIUM ARCHITEKTURY SYSTEMÓW PROGRAMOWANIA

MUL

Uniwersalny język semantyczny dla agentów sztucznej inteligencji

Architektura, reprezentacja pośrednia,
kompilator, transformacje kodu i metodologia ewaluacji

Rafał Rabczuk

badacz niezależny, Wielka Brytania

Specyfikacja projektowa w układzie rozprawy
naukowo-technicznej.
Dokument koncepcyjny — nie stanowi rozprawy doktorskiej
ani opisu zweryfikowanego eksperymentalnie systemu.

Wersja redakcyjna 2.3 | 23 listopada 2026

Streszczenie

Przedmiotem opracowania jest koncepcja *Modular Universal Language* (MUL), platformy programowania przeznaczonej do współpracy z agentami sztucznej inteligencji. Proponowany system oddziela rozumowanie o zmianach w oprogramowaniu od tekstowej składni języków źródłowych. Centralne znaczenie ma typowana reprezentacja pośrednia (IR), powiązana z grafem symboli, mapami źródeł i opcjonalnym indeksem wektorowym. Dokument definiuje architekturę kompilatora, strategie emisji kodu, mechanizm bezpiecznych transformacji, integrację z MDDB, rolę lokalnego modelu AI oraz projekt testów empirycznych. Hipotezy obejmują zmniejszenie liczby tokenów modeli zewnętrznych, redukcję błędów mechanicznych i zwiększenie przenośności operacji programistycznych. Przedstawione korzyści nie są jeszcze potwierdzonymi wynikami badań. Dokument zawiera jedno badanie pilotażowe na małej próbie (rozdział 23); dotyczy ono wyłącznie liczby tokenów zapisu źródłowego i nie potwierdza hipotez. **Słowa kluczowe:**

kompilatory; reprezentacja pośrednia; sztuczna inteligencja; transformacje programów; Code Graph; analiza semantyczna; MDDB; modele językowe.

Abstract

This document proposes Modular Universal Language (MUL), a programming platform intended for AI-assisted software engineering. The architecture separates code transformation intent from language-specific surface syntax and introduces a typed intermediate representation, symbol graph, source maps, and optional vector search. It outlines compiler backends, safe transactional edits, MDDB integration, local AI model specialization, and an empirical evaluation methodology. Claims concerning token reduction and reliability are research hypotheses, not measured outcomes. The document includes one pilot study on a small sample (chapter 23); it concerns only the token count of source text and does not confirm the hypotheses. **Keywords:**

compilers; intermediate representation; AI agents; program transformation; semantic analysis; programming languages.

Spis treści

Streszczenie i

Abstract ii

Wprowadzenie i zakres 1

1 Wizja, problem i definicje 2
- 1.1 Problem, który rozwiązuje MUL . . . 2
- 1.2 Co dokładnie oznacza MUL . . . 2
- 1.3 Cele i nie-cele . . . 2
- 1.4 Trzy mierzalne hipotezy . . . 3

2 Model produktu i sposoby użycia 4
- 2.1 Tryb A: existing code → MUL → patch . . . 4
- 2.2 Tryb B: MUL jako język źródłowy . . . 4
- 2.3 Tryb C: agent jako klient operacji . . . 4
- 2.4 Podział odpowiedzialności . . . 4

3 Architektura systemu 6
- 3.1 Moduły repozytorium . . . 6
- 3.2 Repozytoria i licencjonowanie . . . 6

4 Projekt języka MUL Source 8
- 4.1 Założenia . . . 8
- 4.2 Minimalne konstrukcje v0.1 . . . 8
- 4.3 System pamięci i współbieżność . . . 8
- 4.4 Semantyka i kompatybilność . . . 9
- 4.5 Obiekty bez dziedziczenia . . . 9
- 4.6 Referencja języka dla AI . . . 9

5 MUL IR, graf i warstwa wektorowa 10
- 5.1 Trzy reprezentacje, trzy zadania . . . 10
- 5.2 Struktura węzła i stabilna tożsamość . . . 10
- 5.3 IR: operacje, typy, przepływ . . . 11
- 5.4 Weryfikator IR . . . 11
- 5.5 Format .mulb . . . 11

6 Parsery i import istniejącego kodu 12
6.1 Pipeline importu . . . 12
6.2 Profil Go jako pierwsza implementacja . . . 12
6.3 Profil C++ jako drugi frontend . . . 12
6.4 Round-trip: definicja sukcesu . . . 12

7 Silnik operacji dla AI 14
7.1 Dlaczego operacje zamiast surowego IR . . . 14
7.2 Protokół PLAN → REVIEW → APPLY → VERIFY . . . 14
7.3 Podstawowe operacje API . . . 15
7.4 Budżet tokenów i wyjście progresywne . . . 15
7.5 Bezpieczeństwo zmian . . . 16

8 Kompilator MUL 17
8.1 Ścieżki kompilacji . . . 17
8.2 Kiedy kompilator jest „własny” . . . 17
8.3 Minimalny profil kompilatora v0.1 . . . 18
8.4 Generatory Go/C++/Rust . . . 18
8.5 Runtime, standard library, ABI . . . 18
8.6 Optymalizacje i debugowanie . . . 18
8.7 Przykład CLI . . . 18

9 Lokalny model AI oraz GGUF 20
9.1 Rola modelu . . . 20
9.2 Pipeline treningowy . . . 20
9.3 Dane treningowe . . . 20
9.4 Dataset bez przecieków . . . 20
9.5 Routing modeli . . . 21
9.6 Metryki lokalnego modelu . . . 21

10 Integracja z MDDB i agentami 22
10.1 Kontrakt integracji . . . 22
10.2 Synchronizacja . . . 22
10.3 Agent tool interface . . . 22
10.4 Przykład end-to-end: bump 2.19.13 → 2.19.14 . . . 22

Metodyka i rygor dowodowy 24

11 Testy i benchmarki 25
11.1 Zasada pomiaru . . . 25

11.2 Trzy baseline'y . . . 25
11.3 Matryca testowa . . . 25
11.4 KPI i formuły . . . 26
11.5 Proponowane bramki jakości . . . 26
11.6 Projekt eksperymentu . . . 27
11.7 Ilustracja korzyści, nie wynik pomiaru . . . 27

12 Bezpieczeństwo, prywatność i niezawodność 29
12.1 Model zagrożeń . . . 29
12.2 Deterministyczność i dowody . . . 29
12.3 Wersjonowanie języka . . . 29

13 Mapa możliwości i ograniczeń 30
13.1 Rozszerzenie: PHP, Java, Ruby, TypeScript i JavaScript . . . 31
13.2 MUL Language Compatibility Layer (LCL) . . . 33
13.3 Kontrakt parserów: analiza to nie translacja . . . 33
13.4 Semantyka: dynamiczne i statyczne języki w jednym IR . . . 34
13.5 Backend i kompilator dla nowych języków . . . 35
13.6 Profile zadań AI i minimalizacja tokenów . . . 36
13.7 Testy, KPI i realistyczna kolejność wdrożenia . . . 37
13.8 Decyzje implementacyjne i kontrakt kompatybilności . . . 38
13.9 Adapter językowy jako osobny moduł . . . 39
13.10 Obiektowość obcych języków . . . 39

14 Roadmapa realizacji 40
14.1 Minimalny zespół i role . . . 40
14.2 Backlog pierwszych 12 tygodni (orientacyjny) . . . 41

15 Kryteria sukcesu produktu i decyzja go/no-go 42
15.1 Warunki kontynuacji projektu . . . 42
15.2 Warunki zatrzymania / zmiany kierunku . . . 42

16 Przykłady referencyjne 43
16.1 Refactoring funkcji bez przepisywania źródeł . . . 43
16.2 Nowy kod MUL emitowany do Go . . . 43
16.3 Nieobsługiwana translacja . . . 43
16.4 Raport tokenów . . . 43

17 Decyzje projektowe do zatwierdzenia 45

18 Słownik pojęć **47**

19 Źródła techniczne i inspiracje **48**

20 Podsumowanie wykonawcze **49**

21 Decyzja technologiczna: implementacja kompilatora MUL **50**
21.1 Wybór: Rust dla rdzenia kompilatora 50
21.2 Strategia backendów: Cranelift → LLVM 51
21.3 Komponenty i granice modułów 51
21.4 Zakres MVP i kryteria odbioru 52
21.5 Dalsza perspektywa: własny backend CPU i self-hosting 53
21.6 Ryzyka i decyzja 53

22 Stan realizacji (listopad 2026) **54**
22.1 Zrealizowane 54
22.2 Odstępstwa od roadmapy z rozdziału 14 55
22.3 Ustalenia dodane w wersji 2.1 55

23 Badanie pilotażowe: tokeny zapisu źródłowego **57**
23.1 Cel i zakres 57
23.2 Materiał i metoda 57
23.3 Wyniki 59
23.4 Zmienna kontrolna: wcięcia 62
23.5 Drugi układ: zwarty MUL 62
23.6 Koszt referencji języka i punkt zwrotu 64
23.7 Tekstowy IR a zapis źródłowy 66
23.8 Trafność estymatora wbudowanego 66
23.9 Zagrożenia dla trafności 67
23.10 Wnioski i plan większych prób 68

Wnioski końcowe **69**

Oświadczenia **70**

Spis rysunków

3.1 Architektura referencyjna MUL. Strukturalny IR jest źródłem prawdy, a wektory stanowią indeks pomocniczy. 7

7.1 Protokół zmian: planowanie, ocena, zastosowanie, weryfikacja. Każda faza ma odrębny kontrakt. 14

8.1 Planowane ścieżki kompilacji: emitery źródłowe oraz dwa wymienne backendy natywne. 17

13.1 Warstwa kompatybilności. Uniwersalny rdzeń nie usuwa różnic semantycznych pomiędzy językami. 30

23.1 Średnia geometryczna ilorazu liczby tokenów względem MUL z przedziałem ufności 95% (rozkład t). Linia pozioma 1 oznacza równość z MUL. 60

23.2 Liczba tokenów programu w Go, PHP i C względem MUL (tokenizer o200k_base) z prostymi regresji. 61

23.3 Średnia geometryczna ilorazu liczby tokenów względem zwartego MUL z przedziałem ufności 95% (rozkład t). Linia pozioma 1 oznacza równość z MUL; por. rysunek 23.1. 63

23.4 Łączny koszt odczytu kodu w C oraz w MUL z jednorazowym kosztem referencji języka (tokenizer o200k_base), dla czytelnego i zwartego MUL. Punkty oznaczają V^*. 65

Spis tabel

1.1 Cele i nie-cele 2
1.1 Cele i nie-cele 3

2.1 Podział odpowiedzialności 5

5.1 Trzy reprezentacje, trzy zadania 10

7.1 Podstawowe operacje API 15

9.1 Routing modeli 21

11.1 Trzy baseline'y 25
11.2 Matryca testowa 26
11.3 Proponowane bramki jakości 27
11.4 Ilustracja korzyści, nie wynik pomiaru 28

13.1 Mapa możliwości i ograniczeń 30
13.1 Mapa możliwości i ograniczeń 31
13.2 Rozszerzenie: PHP, Java, Ruby, TypeScript i JavaScript 31
13.2 Rozszerzenie: PHP, Java, Ruby, TypeScript i JavaScript 32
13.3 Kontrakt parserów: analiza to nie translacja 34
13.4 Backend i kompilator dla nowych języków 35
13.4 Backend i kompilator dla nowych języków 36
13.5 Profile zadań AI i minimalizacja tokenów 36
13.6 Testy, KPI i realistyczna kolejność wdrożenia 37
13.6 Testy, KPI i realistyczna kolejność wdrożenia 38

14.1 Roadmapa realizacji 40

17.1 Decyzje projektowe do zatwierdzenia 45
17.1 Decyzje projektowe do zatwierdzenia 46

18.1 Słownik pojęć 47

21.1 Wybór: Rust dla rdzenia kompilatora 50
21.2 Komponenty i granice modułów 51
21.2 Komponenty i granice modułów 52

23.1 Liczba tokenów programów porównawczych (tokenizer o200k_base); wiersze i bajty dotyczą wersji MUL w układzie czytelnym, kolumna „zwarty” tego samego programu w układzie zwartym 59
23.2 Iloraz liczby tokenów język/MUL: iloraz łączny ρ, średnia geometryczna $\bar{r}_g$, przedziały ufności 95% i test Wilcoxona ($n = 8$) . 60
23.3 Przybliżenie liniowe $T_L \approx \alpha + \beta\, T_{\text{MUL}}$ oraz różnice par (tokenizer o200k_base) 61
23.4 Udział wcięć w liczbie tokenów i iloraz łączny ρ przed usunięciem wcięć i po nim (tokenizer o200k_base) 62
23.5 Iloraz liczby tokenów język/MUL przy MUL w układzie zwartym: iloraz łączny ρ względem czytelnego i zwartego MUL, średnia geometryczna $\bar{r}_g$ i jej przedział ufności 95% (rozkład t, $n = 8$) . 63
23.6 Koszt referencji języka C_{ref} i objętość kodu V^* (w tokenach zapisu w języku L), po której się zwraca, przy MUL w układzie czytelnym i zwartym; „–” oznacza brak punktu zwrotu 64
23.7 Liczba tokenów programu MUL i jego tekstowego IR (`mul ir`; tokenizer o200k_base) 65
23.8 Błąd estymatora `mul tokens` względem tokenizerów referencyjnych na 32 plikach źródłowych 66

Wprowadzenie i zakres

Główna teza

MUL ma być niezależną platformą programowania zorientowaną na AI: program może być przedstawiony jako typowana, weryfikowalna reprezentacja semantyczna, nad którą agenci wykonują krótkie operacje. Silnik MUL lokalnie analizuje zależności, przygotowuje i stosuje zmiany, emituje kod języka docelowego albo kompiluje program bezpośrednio. Celem biznesowym jest redukcja liczby tokenów zewnętrznych modeli przy zachowaniu lub poprawie skuteczności zadań programistycznych.

Status i założenia

To specyfikacja projektowa, nie opis istniejącego, ukończonego MUL. Przykładowe instrukcje, formaty, interfejsy, wartości liczbowe i progi testowe są propozycjami do walidacji. Nie istnieją tu jeszcze wyniki benchmarków; wszystkie potencjalne korzyści oznaczono jako hipotezy.

Decyzja architektoniczna w skrócie

- MUL powstaje w osobnym repozytorium i nie zależy od MDDB ani konkretnego modelu AI.
- Rdzeniem jest dokładny, typowany IR i Code Graph. Embeddingi są pomocniczym indeksem, nie nośnikiem prawdy o programie.
- Istniejący kod jest modyfikowany przez bezpieczne patche i mapy źródeł; nowy kod MUL może być w całości emitowany do Go/C++/Rust.
- Własny kompilator rozwija się etapowo: frontend + IR + backend LLVM, a dopiero ewentualnie generator kodu maszynowego.
- Model lokalny może planować operacje MUL; o poprawności decydują walidatory, kompilatory i testy.

1 Wizja, problem i definicje

1.1 Problem, który rozwiązuje MUL

Dzisiejsi agenci kodujący często zużywają tokeny na wielokrotne czytanie tych samych plików, przeglądanie wyników grep/sed, rozpoznawanie symboli, generowanie całych fragmentów tekstu i analizę diffów. Indeksy semantyczne przyspieszają lokalizowanie kodu, ale jeśli zwrócą podobną liczbę znaków do LLM, liczba tokenów pozostaje zbliżona. MUL przenosi mechaniczne i deterministyczne etapy do lokalnego silnika.

1.2 Co dokładnie oznacza MUL

MUL = Modular Universal Language (nazwa robocza). Jest jednocześnie: (a) językiem źródłowym dla nowych aplikacji, (b) wewnętrzną reprezentacją programu, (c) protokołem krótkich operacji edycji i analizy, (d) platformą kompilacji do wielu targetów. „Vector Language” może być nazwą marketingową, ale embeddingi same nie nadają się do bezstratnego przechowywania kodu.

1.3 Cele i nie-cele

Tabela 1.1: Cele i nie-cele

Cel	Definicja / ograniczenie
Minimalizacja tokenów	Pomiar łącznych tokenów wejścia/wyjścia do modelu zewnętrznego na ukończone zadanie, z retry.
Poprawność	Zachowanie semantyki w zdefiniowanym podzbiorze oraz testy i kompilacja po modyfikacji.
Niezależność językowa	Wspólne operacje + jawne dialekty językowe i ograniczenia backendów.

Tabela 1.1: Cele i nie-cele

Cel	Definicja / ograniczenie
Bezpieczeństwo	Transakcje, preconditions, sandbox, brak cichych zmian poza zakresem.
Nie-cel v1	Automatyczna, bezstratna translacja dowolnego C++ do Go.
Nie-cel v1	Model wektorowy jako jedyne źródło kodu lub gwarancja prawidłowego programu.

1.4 Trzy mierzalne hipotezy

1. H1: Agent + MUL zużywa co najmniej 50% mniej zewnętrznych tokenów niż ten sam agent z grep/sed w zadaniach mechanicznych, przy niegorszej poprawności.
2. H2: Agent + MUL rozwiązuje więcej zadań repozytoryjnych za pierwszym razem dzięki operacjom strukturalnym, przy porównywalnym budżecie wykonania.
3. H3: Mały model lokalny redukuje liczbę eskalacji do modelu dużego, bez pogorszenia jakości na zestawie testowym.

Progi są celami eksperymentalnymi, nie obietnicą produktu.

2 Model produktu i sposoby użycia

2.1 Tryb A: existing code → MUL → patch

Dla istniejących repozytoriów Go/C++/Rust pliki źródłowe pozostają autorytatywne. Frontend buduje CST/AST, indeks symboli, typy i IR z referencjami do konkretnych zakresów plików. Zmiany wykonywane są w IR, następnie generator patchy modyfikuje jedynie odpowiednie miejsca. Kompilator i testy weryfikują wynik. To rekomendowany tryb początkowy.

2.2 Tryb B: MUL jako język źródłowy

Nowe programy są pisane bezpośrednio w MUL. Jego pliki są autorytatywne, a backendy generują Go, C++ lub inne języki. Możliwy jest także backend natywny do LLVM IR. Każdy target deklaruje profil zdolności; kod korzystający z nieobsługiwanej funkcji otrzymuje błąd, a nie aproksymację.

2.3 Tryb C: agent jako klient operacji

Claude, GPT, model lokalny albo IDE wysyła krótką komendę semantyczną, np. rename_symbol albo bump_project_version. MUL zwraca podsumowanie, identyfikator planu, przypadki niejednoznaczne i wyniki walidacji, bez dostarczania wszystkich źródeł do okna kontekstowego.

2.4 Podział odpowiedzialności

Tabela 2.1: Podział odpowiedzialności

Warstwa	**Odpowiedzialność**
Agent AI	Intencja, strategia, decyzje o niejednoznacznościach, analiza logiki biznesowej.
MUL Command	Typowane, krótkie instrukcje, ograniczenia i preconditions.
MUL Engine	IR, symbol resolution, plan, apply, rollback, walidacja.
Backend	Emisja kodu źródłowego lub LLVM IR.
System build	Kompilacja, testy, lint, statyczna analiza.
MDDB (opcjonalny)	Wyszukiwanie semantyczne, graf repozytorium i wykrywanie zakresu zmian.

3 Architektura systemu

3.1 Moduły repozytorium

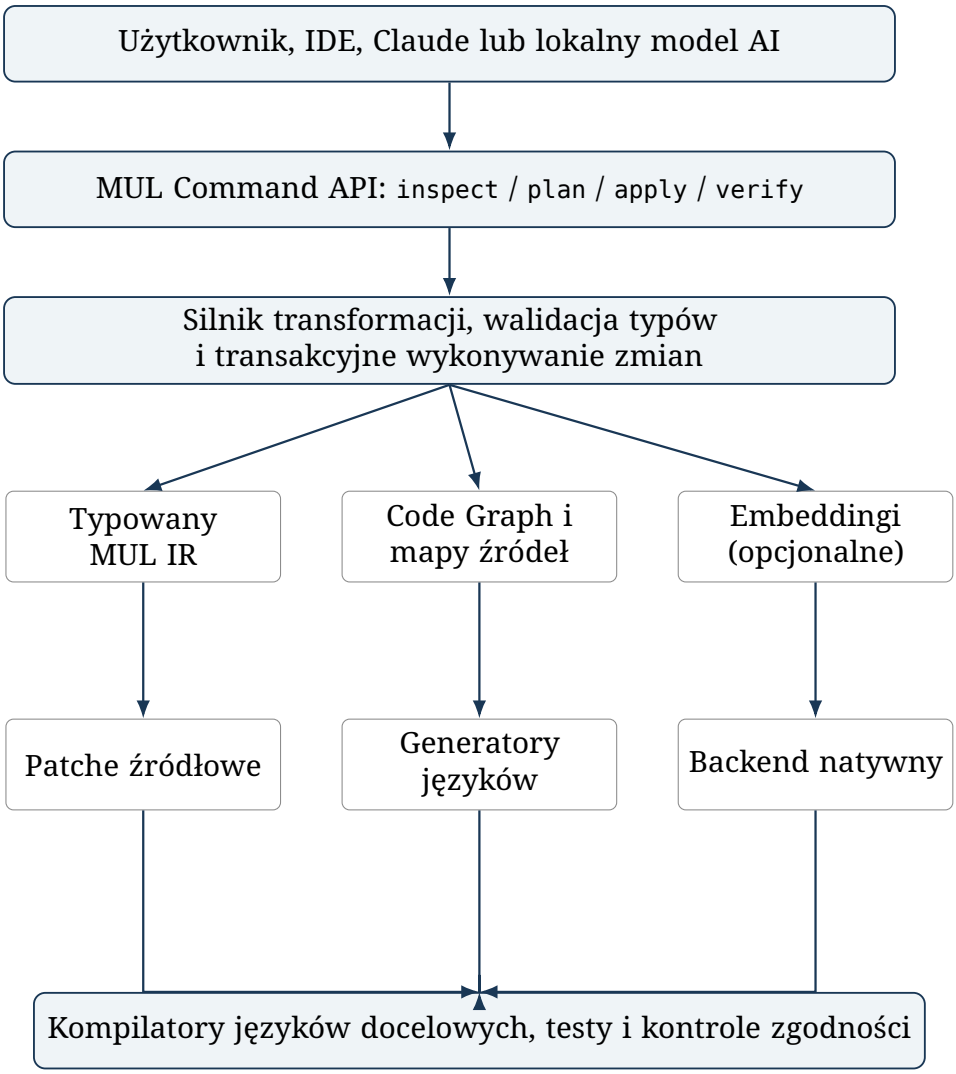
Repozytorium jest jednym monorepo. Rdzeń to workspace Rust podzielony na crate'y o pojedynczej odpowiedzialności; elementy oznaczone jako planowane powstają w kolejnych falach backlogu, a nie jako równoległe wersje istniejących modułów.

```
mul/
crates/mul-syntax/ # lexer, parser, AST, diagnostyka w źródle
crates/mul-sema/ # symbole, typy, efekty
crates/mul-ir/ # typowany IR SSA, lowering z HIR
crates/mul-verifier/ # weryfikacja IR przed backendem
crates/mul-interp/ # interpreter referencyjny IR
crates/mul-backend-cranelift/ # JIT i kod natywny
crates/mul-cli/ # binarka mul
frontends/ # adaptery językowe jako osobne procesy (planowane)
spec/ # normatywne RFC, jedno na temat
docs/adr/ # decyzje architektoniczne
docs/language/ # referencja języka dla ludzi i AI (planowane)
tests/golden/ # przypadki golden i snapshoty IR
examples/ # przykłady uruchamiane przez make demo
site/ # strona dokumentacji i demo (planowane)
```

Moduły z pierwotnego planu (transformacje, runtime, `.mulb`, model, integracje) dochodzą jako kolejne crate'y w tym samym workspace.

3.2 Repozytoria i licencjonowanie

Silnik, kompilator, specyfikacja, dokumentacja i adaptery językowe są utrzymywane w jednym monorepo; wspólne wersjonowanie upraszcza testy zgodności między modułami. Materiały treningowe modelu mogą zostać wydzielone później, jeśli wymagają innej licencji. Licencja powinna być wybrana po audycie zależności LLVM, parserów i modeli. Dane treningowe wymagają udokumentowanego pochodzenia i kontroli licencji.



Rysunek 3.1: Architektura referencyjna MUL. Strukturalny IR jest źródłem prawdy, a wektory stanowią indeks pomocniczy.

4 Projekt języka MUL Source

4.1 Założenia

MUL Source jest czytelnym, statycznie typowanym językiem z jawnymi efektami ubocznymi, modułami, funkcjami, wartościami, strukturami i interfejsami. Jego składnia ma być prostsza niż C++, lecz nie kosztem niejasności semantyki. Zwięzłość dla AI osiąga się przede wszystkim przez operacje na IR, nie przez krypticzne symbole.

```
module demo.version
pub fn next_patch(v: Version) -> Version pure {
    return Version(v.major, v.minor, v.patch + 1)
}
pub fn main() -> i32 io {
    let current = Version(2, 19, 13)
    print(next_patch(current))
    return 0
}
```

4.2 Minimalne konstrukcje v0.1

- Module/import/export; let/var; prymitywy bool, i32, i64, f64, string; struktury, enum, option/result.
- Funkcje, bloki, if, match, while, return, wywołania i podstawowe generyki po późniejszej stabilizacji.
- Jasne reguły konwersji typów, błędów, overflow i kolejności ewaluacji.
- Efekty: pure, io, alloc, async, unsafe; wczesny prototyp może obsługiwać podzbiór pure/io.

4.3 System pamięci i współbieżność

Nie wolno odziedziczyć sprzecznych zasad wszystkich backendów naraz. Proponowany model v1: wartości i referencje z jawnie określonym czasem życia dla zasobów, brak niejawnego współdzielenia mutowalnego stanu, opcjonalne zarządzane obiekty. Backend Go może mapować zarządzane obiekty

na GC; backend C++ wykorzysta RAII/smart pointers; natywny runtime zapewni własną implementację. Konkretna semantyka musi zostać zapisana w specyfikacji i poparta testami; to osobny program prac badawczych.

4.4 Semantyka i kompatybilność

Słowo „uniwersalny” nie oznacza pełnej wymienności języków. MUL definiuje kontrakt zachowania. Każdy frontend emituje wspólne operacje, jeśli zachowują sens, oraz dialekty rozszerzeń, gdy konstrukcja jest specyficzna. Każdy backend publikuje capability matrix, np. reflection, unsafe pointers, exceptions, async, threads, generics. Nieobsługiwane elementy blokują emisję.

4.5 Obiekty bez dziedziczenia

MUL Source nie ma klas ani dziedziczenia implementacji. Zachowanie wiąże się ze strukturami przez metody, a polimorfizm wyraża się interfejsami i kompozycją. Wywołanie przez interfejs jest w IR jawnie oznaczone jako dynamiczne, wywołanie metody konkretnego typu jako statyczne; obie formy mają bezpośredni odpowiednik w emisji do Go. Hierarchie klas z innych języków nie są „spłaszczane” do MUL Source; ich model opisuje sekcja 13.10.

4.6 Referencja języka dla AI

Zanim powstanie wyspecjalizowany model, agenci muszą umieć pisać w MUL na podstawie dokumentacji. Każda cecha języka wprowadzana jest razem z: gramatyką formalną (`spec/grammar.ebnf`) zgodną z parserem, opisem w `docs/language/` (składnia, typy, efekty, kody błędów, idiomy oraz jawna lista konstrukcji, których język nie ma) i przykładem. Zwarta wersja mieszcząca się w kontekście modelu jest dostępna jako `mul guide` i `llms.txt`, a później jako narzędzie API. Test pilnuje, że każdy przykład z dokumentacji przechodzi `mul check` i `mul run`. Ta sama treść jest punktem wyjścia dla danych treningowych modelu lokalnego.

Normatywne RFC dotyczą pojedynczych tematów: przyjęte RFC przyjmuje tylko datowane errata, a zmiana zachowania wymaga nowego RFC, które oznacza poprzednie jako zastąpione. `docs/language/` scala przyjęte RFC w jeden aktualny opis.

5 MUL IR, graf i warstwa wektorowa

5.1 Trzy reprezentacje, trzy zadania

Tabela 5.1: Trzy reprezentacje, trzy zadania

Reprezentacja	Cel	Czy jest bezstratna?
CST + Source Map	Oryginalne tokeny, komentarze, formatowanie, ścieżki i zakresy	Tak, w deklarowanym trybie źródłowym.
Typed Semantic IR + Code Graph	Typy, wartości, symbole, zależności, CFG, efekty	Semantycznie dla wspieranego podzbioru.
Embeddings	Podobieństwo funkcji, intencji, opisów	Nie. Nie służą do odtwarzania kodu.

5.2 Struktura węzła i stabilna tożsamość

```
symbol_id: sym:go:pkg/app:Version#d4e2
kind: function
name: Version
signature: () -> string
origin: {file: "app/version.go", start: 120, end: 173}
effects: [pure]
references: [sym:go:pkg/app:main#2f11]
body_ir_ref: ir:hash:7f9a...
source_hash: sha256:...
embedding_ref: vec:optional:...
```

Identyfikatory muszą być stabilne w ramach zmian, ale system powinien obsługiwać zmianę tożsamości i migrację identyfikatorów. Osobno przechowuje się hash zawartości, identyfikator symbolu i identyfikator wersji migawki. Nie wolno zakładać, że przesunięcie linii nie zmieni żadnego indeksu.

5.3 IR: operacje, typy, przepływ

```
func @get_version() -> !mul.string effects<pure> {
    %v = mul.const_string "2.19.13"
    mul.return %v
}
# Transformacja: replace constant value on referenced node
set_literal(target="ir:node:113", expected="2.19.13", value="2.19.14")
```

Powyższa składnia jest tylko propozycją tekstowego dumpu IR. Należy zdecydować, czy wykorzystujemy MLIR jako host dla dialektów MUL, czy projektujemy niezależny IR. Własny format ma sens tylko jeśli w pomiarach uzasadni koszt utrzymania.

5.4 Weryfikator IR

- Każda wartość ma typ, a każde użycie odnosi się do istniejącej definicji lub parametru.
- Dominacja SSA, poprawność bloków i terminatorów, spójność symboli i widoczności.
- Zgodność efektów, kontraktów i ograniczeń targetu.
- Mapy źródeł powiązane z prawidłowym hashem wejściowych plików.
- Serializacja deterministyczna, zgodna ze schematem i odporna na nieznane pola.

5.5 Format .mulb

Binary IR z nagłówkiem: magic bytes, wersja schematu, flags, target capabilities, content hash i sekcje symbol/type/constant/operation/edge/source-map. Niezbędne: limity wielkości, kontrola integralności, fuzzing parsera i migracje schematu. Zapis binarny oszczędza dysk/transmisję, lecz nie zmniejsza tokenów, jeśli zostanie zamieniony na długi tekst i podany LLM.

6 Parsery i import istniejącego kodu

6.1 Pipeline importu

```
source -> CST -> AST -> name resolution -> typed graph
-> MUL semantic IR -> index / transformations
+ source map to original source bytes
```

Tree-sitter dobrze nadaje się do inkrementalnego CST. Jednak samo CST nie rozwiązuje typów ani referencji. Go frontend powinien używać narzędzi ekosystemu Go do analizy typów, a C++ frontend infrastruktury Clang do semantyki szablonów, makr i przeciążeń.

6.2 Profil Go jako pierwsza implementacja

Obsłużyć: pakiety, importy, deklaracje, funkcje/metody, stałe, struktury, interfejsy, wywołania, podstawowe generyki w kolejnych etapach. Na początku operacje mutujące: zmiana literału, rename symbolu, dodanie parametru tylko w zamkniętym zakresie, aktualizacja wersji. Generować minimalny patch, nie przepisywać całego repozytorium.

6.3 Profil C++ jako drugi frontend

C++ ma makra, warunkową kompilację, szablony, UB i niestandardowe rozszerzenia kompilatorów. Parser bez kompilacyjnej konfiguracji (compile_commands.json) daje niepełny obraz. Początkowo tryb read-only i bardzo ograniczone transformacje. Nie deklarować pełnej round-trip kompatybilności bez oddzielnych testów.

6.4 Round-trip: definicja sukcesu

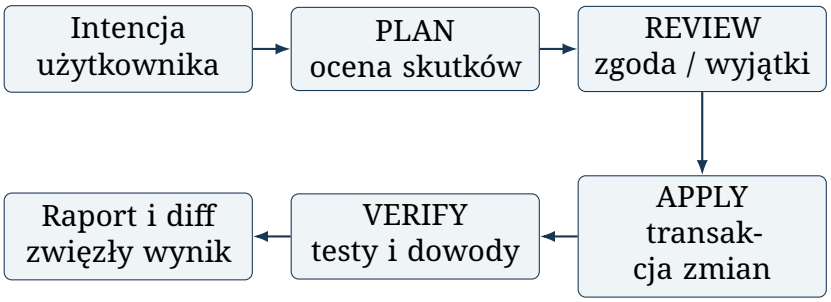
Tryb patch: zachowanie nienaruszonych zakresów bajt po bajcie i semantyki zmienionych zakresów w deklarowanym profilu. Tryb emission: równoważne

zachowanie na określonym zestawie wejść i kontraktach języka, ale nie identyczny kod tekstowy. Komentarze i formatowanie mogą zostać zachowane w patch mode, lecz nie muszą być identyczne w pełnym emitterze.

7 Silnik operacji dla AI

7.1 Dlaczego operacje zamiast surowego IR

Model powinien otrzymywać zwięzłe obiekty i identyfikatory planów, a nie tysiące linii IR. Mechanika obejmuje deduplikację, klasyfikację, planowanie, zastosowanie zmian, walidację i rollback. Model rozstrzyga tylko rzeczy niejednoznaczne. Domyślnie nie ma prawa generować patcha bez sprawdzonych preconditions.



Rysunek 7.1: Protokół zmian: planowanie, ocena, zastosowanie, weryfikacja. Każda faza ma odrębny kontrakt.

7.2 Protokół PLAN → REVIEW → APPLY → VERIFY

```
request:
op: "version.bump"
scope: "repository"
increment: "patch"
base_snapshot: "sha256:abcd..."
dry_run: true
token_budget: 400
response:
plan_id: "plan:8a72"
old_version: "2.19.13"
new_version: "2.19.14"
deterministic_edits: 8
ambiguous_edits: 2
output_truncated: false
```

```
confidence_basis: "typed + manifest rules"
next: "review_ambiguities"
```

7.3 Podstawowe operacje API

Tabela 7.1: Podstawowe operacje API

Operacja	Wynik	Warunki
inspect(scope)	Zwięzła mapa symboli/typów	Budżet wyniku i stronicowanie.
find(query)	Lista identyfikatorów i ranking	Bez gwarancji kompletności przy wektorach.
refs(symbol)	Wyczerpujące użycia w znanym indeksie	Snapshot, pokrycie analizy.
plan(transform)	Plan, diff hash, ryzyka	Bez modyfikacji.
apply(plan_id)	Zmiany transakcyjne	Preconditions i snapshot nadal aktualne.
verify(plan_id)	Testy, typowanie, build, policy	Raport i logi na żądanie.
emit(target)	Kod / artefakt / diagnostyka	Capability profile.
rollback(change_id)	Cofnięcie zmian	Konflikty wykrywane jawnie.

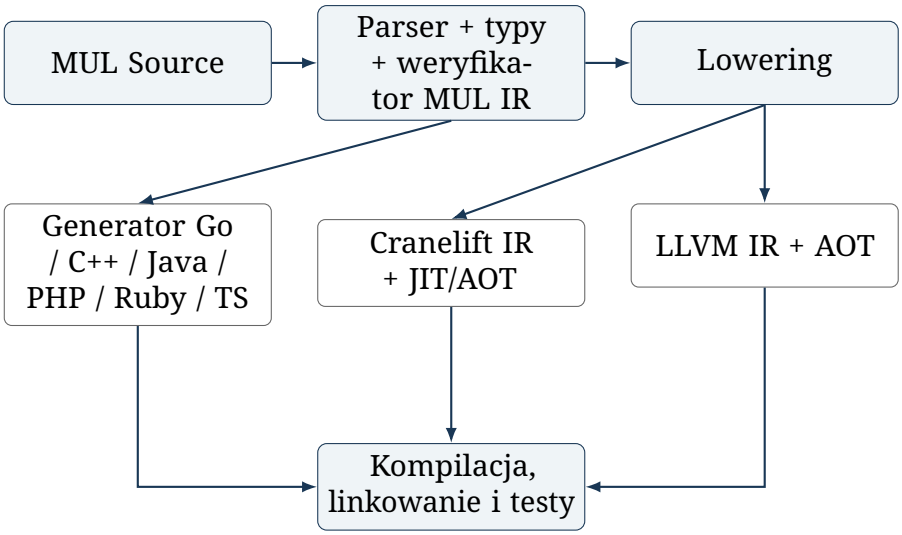
7.4 Budżet tokenów i wyjście progresywne

Każde narzędzie obsługuje detail=summary|references|source, max_items, token_budget, cursor i result_id. Budżet odpowiedzi nie jest budżetem całej rozmowy. Pełny wynik może być przechowywany w artefakcie, do którego agent odwołuje się identyfikatorem. Dla żądania „wszystkie referencje” pełna lista musi być dostępna poza promptem, nawet jeśli do modelu dociera tylko summary.

7.5 Bezpieczeństwo zmian

Każda mutacja ma expected old value, hash bazowego pliku, operację idempotentną lub change_id, ograniczenie zakresu, dry-run, audit log. Przy równoległych zmianach stosować optimistic concurrency; konflikty blokują apply. Przy operacjach wysokiego ryzyka wymagane zatwierdzenie użytkownika. Wtyczka AI nie może uruchamiać dowolnych komend shell bez osobnej, kontrolowanej zgody.

8 Kompilator MUL



Rysunek 8.1: Planowane ścieżki kompilacji: emitery źródłowe oraz dwa wymienne backendy natywne.

8.1 Ścieżki kompilacji

```
MUL Source -> parser -> type checker -> MUL High IR
-> verify / optimize -> MUL Low IR
-> [Go emitter] -> go build
-> [C++ emitter] -> clang++
-> [LLVM lowering] -> LLVM IR -> object -> linker
-> [future native backend] -> x86-64 / AArch64
```

8.2 Kiedy kompilator jest „własny”

Własny parser, checker typów, semantyka, IR i lowering do LLVM tworzą pełnoprawny kompilator MUL. Własny assembler/selector instrukcji nie jest konieczny. LLVM dostarcza sprawdzoną optymalizację i backendy maszynowe; redukuje ryzyko, choć wymaga integracji i utrzymania zgodności wersji.

8.3 Minimalny profil kompilatora v0.1

Programy bez złożonej współbieżności: funkcje, arytmetyka, if/while, struktury, string, IO przez ograniczony runtime, podstawowe błędy. Własne programy MUL mogą być emitowane do Go. Przed dodaniem generyków, async i wyjątków konieczny jest stabilny model semantyczny.

8.4 Generatory Go/C++/Rust

Każdy generator implementuje lowering High IR do konstrukcji języka docelowego, name mangling, import resolution, obsługę bibliotek i mapowanie diagnostyki. Go jest pierwszy; C++ drugi; Rust później, ponieważ ownership/borrowing wymaga rozwiązań semantycznych, których nie da się dopisać samym formatterem.

8.5 Runtime, standard library, ABI

Potrzebne są specyfikacje layoutu danych, String/Array/Map, alokacji, panic/error, ABI FFI, inicjalizacji modułów, concurrency i exceptions. Runtime na Go może korzystać z runtime Go, lecz natywny backend LLVM potrzebuje własnego lub jasno wybranego runtime. Kompatybilność ABI jest niezależnym zestawem testów.

8.6 Optymalizacje i debugowanie

- Przepływ: constant folding, DCE, inlining, CFG simplification, analiza efektów; tylko gdy zachowują semantykę.
- Source maps od MUL do backendowego kodu i z powrotem; błędy wskazują plik i symbol MUL.
- Profile debug/release; deterministyczne buildy, reproducible artifacts i testy różnicowe między targetami.

8.7 Przykład CLI

```
mul check src/
```

```
mul build src/ --target go --out build/go
mul build src/ --target native --triple x86_64-unknown-linux-gnu
mul test ./...
mul inspect --symbol sym:demo:next_patch
mul plan version.bump --increment patch --json
mul apply --plan plan:8a72 --verify build,test
```

9 Lokalny model AI oraz GGUF

9.1 Rola modelu

Lokalny model to planer i klasyfikator operacji, nie substytut systemu typów ani kompilatora. Otrzymuje minimalne summary symboli, wybiera typowaną instrukcję, a przy niepewności eskaluje do dużego LLM. Wynik jest ograniczony schematem/gramatyką. Model może działać offline z GGUF przez llama.cpp; GGUF to format inferencji, nie metoda treningowa.

9.2 Pipeline treningowy

```
licensed source repos + generated MUL programs
-> parser / typed IR / canonical operations
-> programmatic transformations + human-reviewed tasks
-> compile / tests / differential checks
-> dataset: instruction + minimal context -> MUL commands
-> train LoRA/QLoRA on model base
-> evaluate unseen repositories
-> merge / export -> GGUF -> quantize -> runtime
```

9.3 Dane treningowe

Kategorie: (1) transformacje deterministyczne, (2) klasyfikacja referencji, (3) naprawy po błędach kompilacji, (4) zmiany logiczne wymagające testów, (5) wykrywanie braku informacji i odmowa wykonania. Dla trudnych przypadków używać dokumentacji zmian i par PR, ale nie utożsamiać merged PR z bezbłędną prawdą. Przechowywać licencję, commit, hash i wyniki walidacji.

9.4 Dataset bez przecieków

Split według całych repozytoriów oraz czasu, nie pojedynczych funkcji. Usuwać duplikaty i bardzo podobne przykłady. Osobny holdout z projektów niewykorzystywanych w treningu. Mierzyć generalizację między językami i porównać z bazowym niedostrojonym modelem.

9.5 Routing modeli

Tabela 9.1: Routing modeli

Zadanie	Pierwszy wykonawca	Eskalacja
Zmiana wersji, rename, refs	Deterministyczny MUL Engine	Tylko niejednoznaczności.
Transformacja z opisem naturalnym	Lokalny 3B–8B (hipoteza)	Duży model przy niskiej pewności.
Projekt nowego algorytmu	Duży model	Lokalny model może przygotować artefakty.
Weryfikacja poprawności	Kompilator, testy, weryfikator	Model analizuje błędy dopiero po zebraniu dowodów.

9.6 Metryki lokalnego modelu

Schema-valid rate, tool selection accuracy, plan success, pass@1 po kompilacji i testach, liczba eskalacji, tokens/accepted task, zużycie GPU RAM i czas; dla prywatności osobno testy wycieku sekretnych danych z indeksu i prompt injection w komentarzach kodu.

10 Integracja z MDDB i agentami

10.1 Kontrakt integracji

MDDB indeksuje i proponuje identyfikatory symboli oraz potencjalnie powiązane miejsca. MUL weryfikuje precyzyjne referencje w swoim modelu. Interfejs adaptera ma metody ingest, resolve, analyze, plan, apply, verify oraz get_artifact. Gdy brak MDDB, MUL nadal działa na lokalnym repozytorium.

10.2 Synchronizacja

Zdarzenia file changed, branch switched, build flags changed powodują invalidation i reindex. Każdy wynik zawiera snapshot_id. Embeddingi mogą pozostać chwilowo nieaktualne, ale nie wolno używać ich jako dowodu do apply. Utrata spójności indeksów kończy się błędem, nie cichym patchem.

10.3 Agent tool interface

```
mul.inspect mul.refs mul.find
mul.plan mul.apply mul.verify
mul.emit mul.rollback mul.artifact
MUL result envelope:
status | snapshot_id | result_id | warnings
summary | diagnostics_count | capability_flags
```

10.4 Przykład end-to-end: bump 2.19.13 → 2.19.14

4. Agent wysyła intencję version.bump(patch) z zakresem repozytorium.
5. MUL rozpoznaje autorytatywne manifesty, lockfile i zależne referencje; MDDB opcjonalnie proponuje semantyczne kandydaty.

6. Silnik rozdziela zmiany pewne i przypadki historyczne/niejednoznaczne, które mogą nie wymagać aktualizacji.
7. Agent otrzymuje krótkie summary oraz tylko wyjątki; zatwierdza plan.
8. MUL stosuje patche na świeżym snapshot, wykonuje gofmt, build i testy, zwraca result_id i status.

Gdy użytkownik chce zobaczyć wszystkie zmiany, dostaje pełny diff jako artefakt, bez konieczności wstawiania go w kontekst LLM.

Metodyka i rygor dowodowy

Opis MUL ma status propozycji architektonicznej i programu badań, a nie raportu z wdrożenia. Twierdzenia o mniejszej liczbie tokenów, niezawodności i przenośności kodu są *hipotezami*, których nie wolno traktować jako danych eksperymentalnych. Weryfikacja powinna korzystać z kontrolowanych porównań, zadań z niezależnego zbioru testowego, pomiaru wszystkich wywołań modeli oraz powtarzalnych środowisk kompilacji. W szczególności porównywać należy koszt *ukończonego poprawnie* zadania, nie tylko rozmiar pojedynczej odpowiedzi agenta.

11 Testy i benchmarki

11.1 Zasada pomiaru

Wszystkie warianty dostają identyczne zadania, repozytoria, model nadrzędny, limit prób, dostęp do testów i budżet wykonania. Porównywać end-to-end success, a nie pojedyncze udane wywołania. Suma tokenów obejmuje wejście i wyjście dużego modelu, powtórne odczyty, retry, kontekst cache według jego rzeczywistego kosztu. Tokeny lokalnego modelu raportować osobno, nie ukrywać ich.

11.2 Trzy baseline'y

Tabela 11.1: Trzy baseline'y

Wariant	Narzędzia	Cel
A: Agent + shell	grep/rg/sed/read/edit/test	Punkt odniesienia.
B: Agent + MDDB	Wyszukiwanie, graf, wektory + zwykłe edycje	Izoluje wartość indeksowania.
C: Agent + MDDB + MUL	Operacje IR, plan/apply/verify	Izoluje wartość transformacji.
D: C + MUL model lokalny	Routing zadań do małego modelu	Izoluje oszczędność API i koszt lokalny.

11.3 Matryca testowa

Tabela 11.2: Matryca testowa

Kategoria	Przykład	Warunek sukcesu
Mechaniczne	Bump version, rename, import rewrite	Poprawne modyfikacje, brak niepotrzebnych zmian.
Strukturalne	Zmiana sygnatury w całym module	Kompilacja, testy, kompletne referencje.
Logika	Cache z invalidacją, poprawa race condition	Ukryte testy funkcjonalne.
Cross-language	MUL → Go i C++	Równoważne wyniki na korpusie wejść.
Safety	Stary snapshot, symlink, macro ambiguity	Bezpieczna odmowa lub jawny konflikt.
IR	Malformed .mulb, niepoprawne typy	Verifier odrzuca, bez crash.
Model	Nieznany symbol, sprzeczna intencja	Powstrzymanie / eskalacja.

11.4 KPI i formuły

```
external_token_reduction = 1 - tokens_variant / tokens_baseline
success_rate = accepted_tasks / attempted_tasks
valid_patch_rate = verified_applies / all_applies
escalation_rate = LLM_escalations / all_tasks
round_trip_pass_rate = equivalent_rebuilds / supported_cases
cost_per_success = total_model_compute_and_API_cost / accepted_tasks
```

Kluczowy wskaźnik porównawczy: tokeny zewnętrznego modelu na poprawnie rozwiązane zadanie. Sama redukcja tokenów przy spadku jakości nie stanowi sukcesu.

11.5 Proponowane bramki jakości

Tabela 11.3: Proponowane bramki jakości

Kamień milowy	Proponowane kryterium zaliczenia
MVP Go	100% przejścia testów verifiera dla obsługiwanego profilu; zero cichych niezgodności.
Refactor benchmark	□95% sukcesu mechanicznych transformacji na holdout (cel, do sprawdzenia).
Token benchmark	□50% redukcji zewnętrznych tokenów na poprawną operację mechaniczną.
Regresja jakości	Nie gorszy end-to-end success niż baseline w tych samych zadaniach.
Native compiler	Programy z profilu v0.1 przechodzą differential tests vs Go backend.
Model local	Co najmniej 30% mniej eskalacji z zachowaniem skuteczności (cel eksperymentalny).

11.6 Projekt eksperymentu

Zacząć od 50 zadań mechanicznych i 50 zadań logicznych na kilku oddzielnych repozytoriach Go, potem rozszerzyć do 300+. Każdy task wykonać z co najmniej trzema seedami dla stochastycznego modelu; publikować medianę i przedziały ufności z bootstrapu. Do trudniejszych zadań wykorzystać podejście SWE-bench i wielojęzyczne zbiory, ale nie traktować testów jako niepodważalnego dowodu poprawności.

11.7 Ilustracja korzyści, nie wynik pomiaru

Tabela 11.4: Ilustracja korzyści, nie wynik pomiaru

Scenariusz przykładowy	Tokeny zewnętrzne	Poprawność
A: shell	20 000	Musi być zmierzona
B: MDDB	12 000	Musi być zmierzona
C: MUL	3 500	Musi być zmierzona
D: MUL + local	900	Musi być zmierzona

Wartości powyżej służą wyłącznie do pokazania mechanizmu. Nie są prognozą ani wynikiem testu.

12 Bezpieczeństwo, prywatność i niezawodność

12.1 Model zagrożeń

- Prompt injection w komentarzach, README, nazwach zmiennych lub zależnościach zewnętrznych nie może uzyskać uprawnień do operacji shell.
- Dane wektorowe i logi nie mogą przypadkowo wysyłać prywatnego kodu do modelu zewnętrznego.
- Emitowany kod jest niezaufanym artefaktem: kompilacja i testy w sandboxie z limitami CPU, pamięci, sieci i czasu.
- Format .mulb musi być odporny na uszkodzone pliki i złośliwe wielkości struktur.
- Operacje apply mają zasadę najmniejszych uprawnień, allowlistę zakresu plików, audyt i rollback.

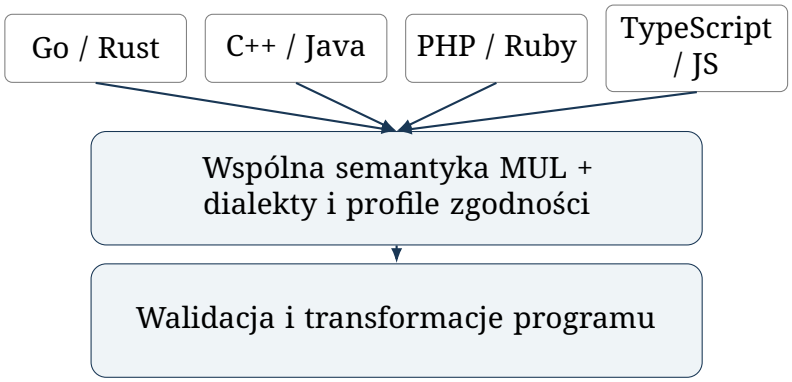
12.2 Deterministyczność i dowody

Każda transformacja zapisuje change_id, wersję silnika, regułę, symbole, input/output hashes, wyniki testów, decyzję użytkownika i zwięzły uzasadniający trace. Agent może prosić o rozwinięcie dowolnego kroku po identyfikatorze, a nie wklejać pełny log.

12.3 Wersjonowanie języka

SemVer dla specyfikacji, oddzielna wersja binarnego IR i protokołu komend. Zmiany breaking wymagają migratora i testów kompatybilności. Format plików powinien odrzucać nieznane wymagane funkcje, a tolerować nieznane opcjonalne metadane.

13 Mapa możliwości i ograniczeń



Rysunek 13.1: Warstwa kompatybilności. Uniwersalny rdzeń nie usuwa różnic semantycznych pomiędzy językami.

Tabela 13.1: Mapa możliwości i ograniczeń

Możliwość	**MVP**	**Pełna wizja**	**Weryfikacja**
Edycja wartości, wersji, rename	Tak (Go)	Wiele języków	Kompilacja + testy + diff.
Refactoring strukturalny	Częściowo	Cross-module/cross-language	Graph consistency + hidden tests.
MUL Source jako język	Podzbiór	Pełna biblioteka i concurrency	Conformance suite.
MUL → Go	Tak	Stabilny pełny backend	Differential tests.
MUL → C++	Nie	Backend	Toolchain matrix.
MUL → Rust	Nie	Backend z ownership	Semantics/borrow checker.
MUL → natywny	Prototyp po Go	LLVM / różne CPU	Binary/ABI/benchmark.

Tabela 13.1: Mapa możliwości i ograniczeń

Możliwość	**MVP**	**Pełna wizja**	**Weryfikacja**
Local GGUF model	Proof of concept	Dostrojony specjalista MUL	Held-out evaluation.
Pełna translacja dowolnego C++	Nie	Nie gwarantować	Językowa niezgodność semantyki.

13.1 Rozszerzenie: PHP, Java, Ruby, TypeScript i JavaScript

Zakres projektu MUL obejmuje docelowo także PHP, Java, Ruby, TypeScript i JavaScript. Żaden z tych języków nie jest automatycznie obsługiwany przez sam fakt istnienia wspólnego IR: potrzebne są frontend, rozpoznawanie semantyki, mapowanie źródła, bezpieczny mechanizm edycji i, niezależnie, backend generowania. Poniższa matryca to plan produktu, nie opis obecnych funkcji.

Tabela 13.2: Rozszerzenie: PHP, Java, Ruby, TypeScript i JavaScript

Język	Frontend / import	Edycja istniejących plików	Generowanie MUL → język	Szczególne ryzyko
PHP	AST/CST + analiza nazw i autoloadingu	Patche wierne źródłu; walidacja PHP	Backend PHP + formatowanie	Dynamiczne include, magic methods, PHPDoc, Composer
Java	AST + symbol solver + classpath	Refaktoryzacje powiązane z typami	Backend Java + javac	Generics/erasure, overload, refleksja, JVM

Tabela 13.2: Rozszerzenie: PHP, Java, Ruby, TypeScript i JavaScript

Język	Frontend / import	Edycja istniejących plików	Generowanie MUL → język	Szczególne ryzyko
Ruby	CST + informacje statyczne i dynamiczne	Ostrożne patche i wyjątki	Backend Ruby + ruby -c	method_missing, reopen classes, eval, metaprogramowanie
TypeScript	Compiler API + tsconfig + project references	Edycje z zachowaniem typów i emit	Backend TS + tsc	Typy warunkowe, dekoratory, erase typów, JSX
JavaScript	AST + resolver zależności	Patche z niepewnością dynamiczną	Backend JS + runtime	Dynamiczne bindingi, eval, ESM/CJS
Go	go/parser, go/types i moduły	Wysoka precyzja przy statycznych symbolach	Backend Go + go build	Generics, channels, concurrency
C++	Clang AST + kompilacja w kontekście projektu	Konserwatywnie przy makrach	Backend C++ + clang++	Preprocessor, ABI, ownership, templates
Rust	Parser/HIR, cargo metadata	Uwzględnienie borrow-checkera	Backend Rust + cargo	Ownership, lifetimes, macros

13.2 MUL Language Compatibility Layer (LCL)

LCL jest wersjonowanym kontraktem między niezależnym rdzeniem MUL IR a dialektami języków. LCL nie zaciera różnic: opisuje, co jest wspólne, co wymaga rozszerzenia, a czego system nie potrafi bezpiecznie reprezentować lub wyemitować.

- Common Core: wartości, typy, wywołania, funkcje, moduły, kontrakty, efekty uboczne, przepływ sterowania, wyjątki i zależności.
- Dialekty: java.jvm (klasy, interfejsy, erasure), php.runtime (autoload, magic methods), ruby.dynamic (open classes, dynamic dispatch), ts.types (typy compile-time, JSX), js.runtime (prototypy i moduły).
- Capabilities: manifest backendu deklaruje obsługiwane konstrukcje oraz gwarancję exact / equivalent / approximate / unsupported. Tryb approximate jest wyłączony dla automatycznego APPLY.
- Source provenance: każdy węzeł IR ma mapowanie do pliku, zakresu bajtów, rewizji, identyfikatora symbolu, konfiguracji kompilacji i języka pochodzenia.
- Semantics preserving: refaktoryzacja wymaga dowodu warunków wstępnych, walidacji typów lub skonfigurowanych testów. Brak dowodu oznacza REVIEW_REQUIRED, nigdy cichy sukces.

Interfejs przykładowy: import(language, project, config) -> module_id; capabilities(language, dialect) -> feature_set; plan(operation, targets) -> plan_id; apply(plan_id, expected_revision) -> patch_id; emit(module_id, target, profile) -> artifacts; verify(artifacts) -> evidence. Operacje muszą być idempotentne tam, gdzie jest to wykonalne, audytowalne i odwracalne.

13.3 Kontrakt parserów: analiza to nie translacja

Tabela 13.3: Kontrakt parserów: analiza to nie translacja

Poziom wsparcia	Gwarancje	Przykładowe testy
L0: Parse	Drzewo składni, zakresy, komentarze, błędy parsera	fixtures składni, fuzz parsera
L1: Resolve	Symbole i odwołania w znanym środowisku	cross-file references, package imports
L2: Edit	Bezpieczne transformacje i kontrola diff	rename, version bump, import update
L3: Emit	Poprawny kod ze wspólnego profilu MUL	MUL $\rightarrow$ source $\rightarrow$ parse/typecheck/build
L4: Translate	Ograniczony transfer między językami z gwarancją semantyki	equivalence properties, integration tests

Przy dynamicznych konstrukcjach wynik L1/L2 może być niepełny. Dla PHP i Ruby poprawne zachowanie oznacza jawne wskazanie niepewności oraz pozostawienie fragmentów źródła bez zmian. Języki te nie mogą mieć deklaracji „pełna analiza referencji” bez sprecyzowania środowiska uruchomieniowego i zakresu dynamicznych wywołań.

13.4 Semantyka: dynamiczne i statyczne języki w jednym IR

- Typy: MUL Core przechowuje zarówno deklarowane typy, jak i ograniczenia/inferencje; dynamiczny typ nie jest zastępowany sztucznym typem statycznym. TypeScript ma typy wymazywane przy emisji JS; Java ma type erasure i overload resolution.
- Wyjątki i efekty: kontrolowany model throws, promise/async, IO, mutations, reflection i external calls. Semantyka async/await TypeScript nie jest równoznaczna z goroutines Go ani wątkami JVM.
- Moduły: Composer/autoload w PHP, gem/Bundler w Ruby, Maven/Gradle w Java oraz npm/tsconfig w TypeScript są częścią kontekstu kompilacji, a

nie tylko listą importów.

- Makra i metaprogramowanie: źródłowe fragmenty nieprzekładalne muszą zachować oryginalną postać + opaque node + ostrzeżenie. Transformacje na takich węzłach wymagają dedykowanego adaptera lub eskalacji do AI.
- ABI i runtime: JVM bytecode, JS runtime, PHP engine, Ruby VM oraz natywny LLVM to odrębne cele. Natywny kompilator MUL nie jest automatycznie zamiennikiem każdej z tych maszyn wykonawczych.

13.5 Backend i kompilator dla nowych języków

Dla kodu pisanego natywnie w MUL: generator Java może tworzyć źródła .java i używać javac; docelowy backend JVM może emitować bytecode. Generator TypeScript tworzy .ts i waliduje przez tsc, a wdrożenie zależy od kompilacji do JavaScript i jego runtime. PHP i Ruby otrzymują generatory źródeł oraz walidację odpowiednich interpreterów. Własny backend LLVM dotyczy natywnego profilu MUL, nie bezpośredniego wykonywania całego PHP, Ruby czy TypeScript.

Oddzielić dwa wymagania: (1) MUL → target: synteza nowego kodu w zakresie gwarantowanego profilu; (2) source → MUL → source: edycja istniejącej aplikacji z zachowaniem stylu, komentarzy, makr, konfiguracji i semantyki. Drugie wymaganie realizować przez source-preserving patches, zamiast przebudowy wszystkich plików.

Tabela 13.4: Backend i kompilator dla nowych języków

Cel	Walidacja minimalna	Walidacja projektowa
PHP	php -l	Composer install/test, PHPUnit lub projektowy runner
Java	javac	Gradle/Maven tests, checkstyle/lint zgodnie z projektem
Ruby	ruby -c	Bundler, RSpec/Minitest, RuboCop jeśli używany

Tabela 13.4: Backend i kompilator dla nowych języków

Cel	Walidacja minimalna	Walidacja projektowa
TypeScript	tsc --noEmit	Test runner + ESLint i build bundlera
JavaScript	Parser + test runner	Node runtime, bundler, ESM/CJS checks

13.6 Profile zadań AI i minimalizacja tokenów

Mały model MUL lub Claude nie musi otrzymywać całego drzewa. Wybór operacji jest wąski, a rozbudowane IR, AST, graf i pliki pozostają po stronie silnika. Dla zadań z warstwami dynamicznymi wyniki zawierają uncertainty_code i konkretne wyjątki do przeglądu.

Tabela 13.5: Profile zadań AI i minimalizacja tokenów

Przykładowe zadanie	Praca silnika MUL	Kiedy konieczny jest AI
Bump wersji w Composer/package.json/pom.xml/gemspec	Rozpoznanie manifestów, wersji projektu, bezpieczne edycje, test	Niejasne wersje dokumentacyjne i wersje niezależnych modułów
Rename symbolu	Resolve references, patch, format, typecheck	Refleksja, dynamic dispatch, nazwy budowane w runtime
Dodaj pole/parametr	Modyfikacje struktury, wywołań, typów i test	Zmiana publicznego API i decyzje semantyczne
Przenieś funkcję	Aktualizacja importów i zależności	Granice modułów, side effects, framework hooks
MUL → Java/TS/PHP/Ruby	Generator z ograniczonym profilem i walidacją	Brak odpowiednika operacji, zależności runtime

Przykład kompaktowego raportu przedstawiono poniżej. Jego pełna struktura jest przechowywana poza kontekstem modelu.

```
{p̈ lan_id:p̈ l-412,¨
c̈ hanges:Ï8,v̈ alidated:Ï6,
r̈ eview_required:2,
ärtifact_id:p̈ atch-412}¨
```

Szczegółowe patche pozostają lokalnie i są pobierane na żądanie; token_budget ogranicza tylko odpowiedź, nie weryfikację ani kompletność przeszukiwania.

13.7 Testy, KPI i realistyczna kolejność wdrożenia

Każdy język otrzymuje niezależną bramkę jakości. Wynik testów „MUL działa na Java” nie wynika z tego, że działa na Go. Publiczne deklaracje obsługi muszą wymieniać poziom L0–L4 i profil semantyczny.

Tabela 13.6: Testy, KPI i realistyczna kolejność wdrożenia

Test	Metoda	Kryterium startowe (proponowane, nie zmierzone)
Parse / print stability	Losowe próbki projektów + round-trip bez edycji	Brak niezamierzonych zmian źródła dla L2
Rename correctness	Fixture z dynamic dispatch i shadowing	100% testów pozytywnych, 0 niejawnych niebezpiecznych edycji
Build after transform	Natywne toolchainy każdego języka	Nie pogorszyć build/test w zadaniach obsługiwanych
Behavioral equivalence	Testy integracyjne i property-based	Brak wykrytych regresji w zadeklarowanym profilu

Tabela 13.6: Testy, KPI i realistyczna kolejność wdrożenia

Test	Metoda	Kryterium startowe (proponowane, nie zmierzone)
Token efficiency	Równe zadania: agent+grep vs agent+MUL	Cel badawczy □50% tokenów mniej przy porównywalnym sukcesie
Full-task success	Wymagania + build + test + diff review	Mierzyć pass@1 i liczbę eskalacji

Fazy wdrożenia: A: Go L2 i benchmark; B: TypeScript L2 + Java L1/L2; C: PHP L2 + Java L3; D: Ruby L1/L2 + TypeScript L3; E: PHP/Ruby L3 dla podzbiorów MUL; F: transfer międzyjęzykowy L4 po testach zgodności. Priorytety zmieniać na podstawie zapotrzebowania i benchmarków, nie obietnicy „każdy język od ręki”.

13.8 Decyzje implementacyjne i kontrakt kompatybilności

- Specyfikacja capabilities.json: każdy adapter deklaruje nazwę języka, wersję, poziom L0–L4, dialekty, niezawodność resolve, obsługę emisji, komendę weryfikacji i ograniczenia.
- Versionowanie: spec MUL Core, dialect schema i adapter API mają osobne numery wersji; odmowa importu/transformacji przy niekompatybilnym kontrakcie.
- Bramka wykonania: parser → resolve → plan → preconditions → apply atomic patch → format/lint/typecheck → build/tests → raport; rollback przy niepowodzeniu według polityki projektu.
- Bezpieczeństwo: przy uruchamianiu npm/composer/bundle/gradle/maven stosować sandbox i wyraźne uprawnienia. Zewnętrzne skrypty instalacyjne i build hooks mogą wykonywać arbitralny kod.
- Otwarta rozszerzalność: third-party adapter SDK; test suite zgodności; plugin nie może deklarować pełnej obsługi bez zaliczenia publicznych

testów.

Konkluzja rozszerzenia: PHP, Java, Ruby, TypeScript i JavaScript są pełnoprawnymi celami architektury MUL. Każdy ma osobny frontend, dialekt semantyczny i ewentualny backend. Wsparcie analizy, transformacji i generowania jest niezależnie mierzone, dzięki czemu język AI nie sprowadza wszystkich runtime do fikcyjnie identycznej maszyny.

13.9 Adapter językowy jako osobny moduł

Każdy język źródłowy jest obsługiwany przez osobny adapter: niezależny proces, napisany w dowolnym języku i komunikujący się z rdzeniem przez JSON na standardowym wejściu i wyjściu. Rdzeń nie zna żadnego języka z nazwy; frontend Go jest pierwszym adapterem na tym kontrakcie, a nie przypadkiem szczególnym. Pozwala to korzystać z narzędzi analizy z ekosystemu danego języka, izolować awarie i limity zasobów adaptera oraz dodawać język bez zmian w rdzeniu. Adapter deklaruje w `capabilities.json` poziom L0–L4, dialekty i komendę weryfikacji, a jego deklaracja jest potwierdzana wspólnym zestawem zgodności. Kandydatami na adaptery tylko do odczytu (L0/L1) są także języki o dużej bazie starszego kodu, np. Fortran i COBOL; ich mechanizmy preprocesora i `COPY`/`REPLACE` traktuje się jak makra (węzły opaque).

13.10 Obiektowość obcych języków

Klasy, dziedziczenie, traity, nadpisywanie metod i magic methods są w Code Graph reprezentowane jako węzły i krawędzie dialektu (`extends`, `implements`, `overrides`), z odwołaniem do oryginalnych zakresów bajtów, a nie tłumaczone na MUL Source. Wywołania rozstrzygane w czasie wykonania (dynamic dispatch, callbacki przekazywane jako nazwa w łańcuchu znaków, np. system hooków w aplikacjach PHP, `call_user_func`, `include` ze ścieżką obliczaną dynamicznie) są krawędziami z `uncertainty_code` i wymagają przeglądu przed APPLY. Duży projekt PHP o rozbudowanym systemie hooków jest przewidziany jako korpus testowy adaptera PHP; edycje trafiają do oryginalnych plików PHP.

14 Roadmapa realizacji

Tabela 14.1: Roadmapa realizacji

Faza	Zakres	Artefakt / bramka
0 — Spec & research	Formalna semantyka, definicje IR, capability matrix, testy	MUL RFC 001–010 + 30 golden cases.
1 — Core	Parser MUL, typy, serializer, verifier, symbol IDs	CLI check/inspect; fuzz tests.
2 — Go frontend/patch	Parser i resolver Go, source maps, 4 transformacje	Bezpieczny Go round-trip; benchmark A/B/C.
3 — Go backend	Emisja programów MUL do Go, runtime basic	Build/test end-to-end.
4 — AI API + MDDB	Plan/apply/verify, token budgets, adapter	Automatyczne zadania repozytoryjne.
5 — Local model	Dataset, LoRA, GGUF, routing	Benchmark D i raport kosztów.
6 — LLVM	Lowering i natywna kompilacja	Differential test Go vs native.
7 — Multi-language	C++ frontend i backend; Rust później	Zdefiniowane coverage i ograniczenia.

14.1 Minimalny zespół i role

Rdzeń: lider kompilatora/semantyki, inżynier Go frontend, inżynier transformacji i API, inżynier testów/benchmarków; AI/ML dołącza po pojawieniu się dobrego korpusu. Dodatkowo potrzebny audyt bezpieczeństwa przed uruchomieniem obcego kodu. Terminy zależą od zakresu i zespołu; nie należy deklarować pełnego MUL w kilku sprintach.

14.2 Backlog pierwszych 12 tygodni (orientacyjny)

1. Tydzień 1–2: normatywne definicje operacji, typów i source maps; benchmark wejściowy grep vs MDDB.
2. Tydzień 3–4: IR/verifier/serializacja i mała implementacja MUL Source.
3. Tydzień 5–7: Go ingest, symbole i pierwsze bezpieczne edycje wersji/literału/rename.
4. Tydzień 8–9: plan/apply/verify, snapshot, rollback, poprawny report CLI.
5. Tydzień 10–11: benchmark 100 zadań; analiza niepowodzeń, poprawki.
6. Tydzień 12: decyzja go/no-go dla dalszego Go backendu i modelu lokalnego.

To plan proponowany, nie obietnica ukończenia. Pełny język i kompilator natywny to odrębne kolejne etapy.

15 Kryteria sukcesu produktu i decyzja go/no-go

15.1 Warunki kontynuacji projektu

- MUL wykonuje zadania mechaniczne co najmniej równie niezawodnie co baseline, z mierzalnie mniejszym zużyciem tokenów.
- Nie istnieje klasa nieobsługiwanych transformacji, które MUL wykonuje po cichu ze zmianą znaczenia programu.
- System odtwarza istniejący kod źródłowy w deklarowanym zakresie i daje audytowalny diff.
- Interfejs narzędzi jest użyteczny także bez specjalnie dostrojonego modelu.
- Własny kompilator ma rzeczywistą wartość: portability, deterministyczne operacje, kontrola runtime, potencjalna optymalizacja; nie jest tworzony tylko dla prestiżu.

15.2 Warunki zatrzymania / zmiany kierunku

Jeśli wyniki pokażą podobne zużycie tokenów lub wysoką liczbę błędów, najpierw uprościć produkt do deterministycznego engine operacji na AST/Code Graph. Jeśli pełny MUL Source wymaga nieproporcjonalnie dużo pracy, utrzymać wyłącznie MUL IR i protokół transformacji. Jeśli model lokalny nie poprawia kosztu na poprawne zadanie, pozostawić go opcjonalnym.

16 Przykłady referencyjne

16.1 Refactoring funkcji bez przepisywania źródeł

```
mul plan rename --symbol sym:go:app:GetVersion \
    --new-name ReadVersion --scope repo
# PLAN: 17 references, 4 files, 0 unresolved
mul apply --plan plan:29fa --verify build,test
# APPLIED: 18 edits, BUILD: pass, TESTS: pass
```

16.2 Nowy kod MUL emitowany do Go

```
// MUL Source
module math
pub fn add(a: i32, b: i32) -> i32 pure { return a + b }
// wynik generatora Go - przykład
package math
func Add(a int32, b int32) int32 { return a + b }
```

Przykład ilustruje przypadek zgodny semantycznie. Dla pełnej zgodności język musi zdefiniować m.in. overflow liczb całkowitych i reprezentację string.

16.3 Nieobsługiwana translacja

```
mul emit --target go complex_cpp.mulb
ERROR MUL-CAP-014: target "go"does not support
cpp.volatile_atomic_ordering at sym:cpp:pkg:sync_impl
No output emitted. Source unchanged.
```

16.4 Raport tokenów

```
task_id: bump-001
```

```
baseline_external_tokens: 20430
mul_external_tokens: 3760
local_model_tokens: 0
pass: true
verified_build: true
verified_tests: true
reduction_external: 81.6% # example only
```

Pomocniczo polecenie `mul tokens <plik> --compare <plik>...` szacuje liczbę tokenów programu MUL i programu o tym samym zachowaniu w innym języku oraz podaje różnicę. Wynik zależy od tokenizera, dlatego tokenizer jest wymienny, domyślny estymator działa lokalnie, a kod nie jest wysyłany na zewnątrz bez jawnej konfiguracji. Ten sam licznik służy w harnessie benchmarku; porównanie rozmiaru źródeł nie zastępuje pomiaru tokenów na poprawnie wykonane zadanie.

17 Decyzje projektowe do zatwierdzenia

Tabela 17.1: Decyzje projektowe do zatwierdzenia

ID	Decyzja	Rekomendacja
ADR-001	MUL oddzielony od MDDB	Tak; jawny adapter.
ADR-002	Kanoniczna struktura	Typed IR + source maps; embeddingi opcjonalne.
ADR-003	Pierwszy język	Go; najpierw patch mode.
ADR-004	IR framework	Własny IR w Rust; MLIR dopiero gdy pomiar wykaże potrzebę (przyjęte warunkowo).
ADR-005	Runtime i pamięć	Ustalić profil v0.1 przed backendami.
ADR-006	Model AI	Nieobowiązkowy; LoRA→GGUF po zebraniu danych.
ADR-007	Native compiler	Zastąpione przez ADR-011.
ADR-008	Metryka kluczowa	Tokeny zewnętrzne na poprawnie wykonane zadanie.
ADR-009	Bezpieczeństwo	Transakcje, sandbox, audit, brak silent fallback.
ADR-010	Zakres „uniwersalności”	Capability profiles; żadnych obietnic any→any.

Tabela 17.1: Decyzje projektowe do zatwierdzenia

ID	Decyzja	Rekomendacja
ADR-011	Implementacja kompilatora	Rust jako rdzeń; Cranelift pierwszy, LLVM później.

18 Słownik pojęć

Tabela 18.1: Słownik pojęć

Termin	Definicja
IR	Intermediate Representation: formalny, typowany model programu.
CST/AST	Drzewo konkretnej / abstrakcyjnej składni.
Source map	Mapowanie elementów IR na pozycje źródła i odwrotnie.
Code Graph	Graf symboli, typów, definicji, użyć i zależności.
Embedding	Wektor przybliżający podobieństwo semantyczne; nie jest odwracalnym kodem.
Dialect	Zbiór operacji z osobną semantyką lub obsługą targetu.
Lowering	Przekształcenie z wyższego IR do niższego lub do języka docelowego.
GGUF	Format dystrybucji modeli do inferencji, stosowany m.in. w llama.cpp.
Snapshot	Spójny stan źródeł i indeksów, do którego odnosi się plan zmian.
Differential testing	Porównanie zachowania równoważnych implementacji na tych samych wejściach.

19 Źródła techniczne i inspiracje

Poniższe źródła opisują istniejące technologie i metodologie. Nie są dowodem, że hipotetyczny MUL osiągnie deklarowane korzyści.

1. MLIR Language Reference — https://mlir.llvm.org/docs/LangRef/ (operacje, dialekty, typy, regiony i verifier).

2. MLIR Tutorials — https://mlir.llvm.org/docs/Tutorials/Toy/Ch-2/ (emisja własnego IR i rozbudowa języka).

3. LLVM: My First Language Frontend — https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/ (ścieżka budowy kompilatora).

4. Tree-sitter — https://tree-sitter.github.io/tree-sitter/ (inkrementalny parser i CST).

5. llama.cpp model documentation — https://github.com/ggml-org/llama.cpp/blob/master/docs/models.md (GGUF i modele lokalne).

6. SWE-bench (ICLR 2024) — https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html (metodyka ewaluacji napraw kodu).

7. Multi-SWE-bench — https://arxiv.org/abs/2504.02605 (benchmark wielojęzykowy).

20 Podsumowanie wykonawcze

MUL należy budować jako osobną platformę programowania dla AI: formalny język źródłowy, dokładny IR, Code Graph, bezstratne mapy źródeł dla trybu edycji, silnik transformacji, backendy Go/C++/Rust oraz własny kompilator oparty najpierw na LLVM. Model GGUF może zostać dodany jako wyspecjalizowany lokalny planer, ale nie stanowi fundamentu poprawności. Wartość pierwszej wersji musi zostać udowodniona na zadaniach repozytoryjnych i rzeczywistych kosztach tokenów. Bez tego MUL ryzykuje stanie się kosztowną abstrakcją, która przepisuje Go na własny język po to, by z powrotem przepisać go na Go.

Rekomendowany pierwszy produkt: MUL Core + Go Frontend + plan/apply/verify + benchmarki. Drugi: MUL Source $\rightarrow$ Go. Trzeci: LLVM native. Czwarty: modele lokalne i kolejne języki. Każdy etap powinien mieć niezależne kryteria poprawności, oszczędności i możliwości wycofania.

21 Decyzja technologiczna: implementacja kompilatora MUL

Status: rekomendowana architektura implementacyjna (23 listopada 2026). Ten rozdział uzupełnia rozdziały 3, 8, 14 i 17. Decyzja dotyczy języka implementacji MUL, nie ogranicza zestawu języków obsługiwanych przez MUL.

21.1 Wybór: Rust dla rdzenia kompilatora

Rekomendacja: rdzeń MUL Compiler napisać w Rust. Dotyczy to parsera MUL Source, analizy semantycznej, systemu typów, MUL IR, weryfikatora IR, grafu symboli, zarządzania zmianami, optymalizatora oraz koordynacji backendów. Rust umożliwia precyzyjną kontrolę pamięci i bezpieczną współbieżność przy rozbudowanej strukturze danych; nie eliminuje jednak potrzeby testów i przeglądów bezpieczeństwa.

Tabela 21.1: Wybór: Rust dla rdzenia kompilatora

Kryterium	Rust	Go	C / assembler
Bezpieczeństwo pamięci	Silny system typów, ownership	GC i proste API	Ręczna kontrola, większe ryzyko błędów
IR / grafy / transformacje	Bardzo dobry wybór	Dobry, prosty w utrzymaniu	Możliwe, wyższy koszt utrzymania
Backend natywny	Dojrzałe biblioteki i FFI	Możliwy przez integracje	Wysoka kontrola, duży koszt implementacji
Praktyczne zastosowanie	Główny rdzeń MUL	Usługi sieciowe / MDDB	Niskopoziomowe fragmenty, ABI

Assembler nie jest rekomendowany jako język implementacji całego kompilatora. Pisanie parsera, systemu typów i grafu semantycznego w assemblerze

podwyższa koszty bez gwarancji przyspieszenia. C pozostaje opcją dla bardzo wąskich warstw integracji; Go nadaje się do usług, CLI i otoczenia infrastrukturalnego.

21.2 Strategia backendów: Cranelift → LLVM

Etap prototypowy: Cranelift jako backend do szybkiego uzyskania natywnych programów/JIT dla ograniczonego podzbioru MUL, bez budowania od zera generatora instrukcji CPU. Etap docelowy: LLVM jako opcjonalny backend silnie optymalizujący dla wydań produkcyjnych. Oba backendy powinny konsumować tę samą zweryfikowaną semantykę MUL IR poprzez jawny kontrakt lowering. Cranelift i LLVM nie muszą obsługiwać identycznych funkcji ani platform od pierwszego wydania.

Pipeline: MUL Source → parser → analizator typów/efektów → zweryfikowany MUL IR → lowering → Cranelift IR lub LLVM IR → kod maszynowy → linker → program. Równoległe backendy źródłowe: MUL IR → generator Go/C++/Rust/Java/PHP/Ruby/TypeScript, zgodnie z profilami kompatybilności.

21.3 Komponenty i granice modułów

Tabela 21.2: Komponenty i granice modułów

Moduł	Implementacja	Kontrakt
mul-syntax	Rust	Tokeny, CST/AST, błędy lokalizowane w źródle
mul-sema	Rust	Symbole, typy, efekty, diagnostyka
mul-ir	Rust	Typowany IR, stabilne identyfikatory, wersjonowanie
mul-verifier	Rust	Weryfikacja typów, CFG i efektów przed backendem

Tabela 21.2: Komponenty i granice modułów

Moduł	Implementacja	Kontrakt
mul-interp	Rust	Interpreter referencyjny; wzorzec dla testów różnicowych
mul-transform	Rust	Plan/Apply/Verify, transakcje, odwracalne zmiany
mul-backend-cranelift	Rust	Lowering + emit/JIT; profil v0.x
mul-backend-llvm	Rust + LLVM API	Lowering + emit AOT; profil produkcyjny
mul-emit-*	Rust / adapter językowy	Generowanie źródła, source maps, walidacja
mul-cli / mul-api	Rust; protokół niezależny	CLI, JSON-RPC/gRPC; integracja MDDB

21.4 Zakres MVP i kryteria odbioru

- Wersja MVP: zmienne lokalne, wartości i32/bool/string, funkcje, wywołania, warunki, proste pętle; jawnie zdefiniowana semantyka i diagnostyka.
- Pierwszy test end-to-end: kod MUL → zweryfikowany IR → Cranelift → program uruchamialny; porównanie wyników z interpreterem referencyjnym lub testami semantycznymi.
- Test backendów: identyczny zestaw przypadków dla Cranelift i LLVM, porównanie rezultatów, wykrywanie błędów lowering i niezgodności ABI.
- Próba integracyjna: import ograniczonego kodu Go → MUL → kontrolowana transformacja → patch Go → gofmt/go test; źródło pozostaje punktem odniesienia dla trybu round-trip.
- Pomiar AI: tokeny wejściowe i wyjściowe, liczba tur, odsetek zaliczonych testów, nieudane poprawki, koszt eskalacji do dużego modelu; porównanie do Claude + grep i Claude + MDDB.

- Bezpieczeństwo: fuzzing parsera/serializacji, weryfikacja IR przed backendem, izolacja procesu kompilacji i limit zasobów, kontrola wersji API.

21.5 Dalsza perspektywa: własny backend CPU i self-hosting

Własny generator x86-64/ARM64 bez LLVM i Cranelift nie jest wymagany do uznania MUL za własny kompilator: własny frontend, semantyka, IR i lowering już stanowią własny kompilator wykorzystujący zewnętrzny backend. Samodzielny backend maszynowy i kompilacja kompilatora MUL napisanego w MUL (self-hosting) to osobne, długoterminowe cele badawcze, po stabilizacji języka, runtime, ABI i biblioteki standardowej.

21.6 Ryzyka i decyzja

Ryzyka: koszt utrzymania dwóch backendów, rozbieżności semantyczne, zależności systemowe LLVM, ograniczenia targetów i potrzeba własnego runtime. Minimalizacja: backend trait z testami kontraktowymi, mały wspólny podzbiór języka, macierz kompatybilności i automatyczne testy różnicowe. Decyzja: Rust jako rdzeń; Cranelift pierwszy; LLVM później; assembler tylko tam, gdzie wykazana korzyść. Ta kolejność nie obiecuje sama w sobie niższego zużycia tokenów AI: to wynik zwięzłego protokołu operacji oraz lokalnego wykonywania transformacji.

Źródła techniczne do weryfikacji podczas implementacji: `https://cranelift.dev/`; `https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html`; `https://doc.rust-lang.org/book/`.

22 Stan realizacji (listopad 2026)

Rozdział opisuje stan prototypu w chwili wydania wersji 2.1 dokumentu. Szczegółowy plan prac jest prowadzony w pliku `BACKLOG.md` repozytorium jako kolejne fale zakończone bramkami.

22.1 Zrealizowane

- MVP-0 (RFC-001): funkcje bez parametrów zwracające `i32`, arytmetyka ze sprawdzaniem przepełnienia i dzielenia przez zero, pełny pipeline parser → sema → IR → verifier → interpreter.
- Język MVP (fala 2): parametry funkcji i wywołania z rekurencją, zmienne `let`/`var` i zakresy bloków, `bool`, porównania, operatory logiczne, `if`/`else`, `while`, efekty `pure`/`io`, typ `string` i `print` (program Hello world), wartość `main` jako kod wyjścia procesu, sprawdzanie typów w weryfikatorze IR; każda cecha ma własne RFC.
- Referencja języka dla AI: gramatyka formalna, `docs/language/`, `mul guide` i `llms.txt`.
- Backend Cranelift: uruchamianie JIT, budowa pliku wykonywalnego oraz testy różnicowe interpreter vs kod natywny na całym zestawie golden.
- Parametry funkcji i wywołania z rekurencją ograniczoną limitem głębokości (RFC-002), zmienne `let`/`var`, przypisanie i zasięgi bloków bez przesłaniania nazw (RFC-005) oraz dokumentacja języka dla AI (`docs/language/`, `mul guide`, `llms.txt`).
- Typ `bool`, porównania, operatory logiczne ze skróconym wartościowaniem i instrukcja `if`/`else` (RFC-006); IR z wieloma blokami bazowymi i parametrami bloków zamiast funkcji phi, weryfikator sprawdzający dominację SSA.
- Pętla `while` (RFC-007): blok nagłówka pętli przyjmuje jako parametry zmienne przypisywane w ciele, krawędź powrotna przekazuje ich nowe wartości; interpreter i Cranelift wykonują pętle bez zmian w konwencji wywołań.

- Efekty `pure` i `io` w sygnaturach (RFC-009): brak adnotacji oznacza `pure`, funkcja `pure` nie może wywołać funkcji `io` (`MUL-T013`); funkcja IR niesie swój efekt, a weryfikator odrzuca takie wywołanie (`MUL-V011`).
- Bramka jakości uruchamiana lokalnie przed każdym commitem (`make check`: formatowanie, linter, testy, pokrycie co najmniej 98% na plik, audyt zależności); projekt nie korzysta z zewnętrznego CI.

22.2 Odstępstwa od roadmapy z rozdziału 14

- Kolejność: najpierw kompilator na minimalnym podzbiorze języka (sekcja 21.4), następnie tryb patch dla Go i benchmark z decyzją go/no-go, dopiero potem emisja do Go, pełny język v0.1, LLVM, model lokalny i kolejne języki.
- Backend natywny: Cranelift jako pierwszy (ADR-011), LLVM później.
- Kontrakt adaptera językowego powstaje razem z pierwszym frontendem (sekcja 13.9), a nie po nim.

22.3 Ustalenia dodane w wersji 2.1

- Brak dziedziczenia w MUL Source; metody, interfejsy i kompozycja (sekcja 4.5).
- Model obiektowości obcych języków w Code Graph bez translacji na MUL (sekcja 13.10).
- Referencja języka dla AI i zasada jednego RFC na temat (sekcja 4.6).
- Szacowanie tokenów `mul tokens` (sekcja 16.4).
- Strona dokumentacji z galerią przykładów generowaną przez `mul` (pierwszy przykład: Hello world) oraz demo w przeglądarce, w którym interpreter skompilowany do WebAssembly wykonuje kod lokalnie, bez serwera.
- Języki jako moduły: każdy adapter ma własny katalog, wersję, testy i manifest możliwości; poprawka jednego adaptera ani dodanie nowego nie wymaga zmian w rdzeniu ani w innych adapterach. Nowy adapter powstaje z generatora szkieletu.
- Ewolucja języków: adapter deklaruje obsługiwany zakres wersji języka i poza nim działa tylko do odczytu lub odmawia; nowe wydanie języka

przechodzi procedurę sprawdzenia na korpusie (round-trip bez zmian, porównanie odsetka węzłów opaque); nowe konstrukcje trafiają najpierw do dialektu, a do Common Core dopiero, gdy pojęcie występuje w co najmniej dwóch językach; nieutrzymywany adapter otrzymuje status zamrożonego na konkretnej wersji.

- Oznaczanie wersji: adapter odczytuje zadeklarowaną wersję języka projektu i prowadzi tabelę konstrukcji (wprowadzone, przestarzałe, usunięte w wersji); każdy węzeł grafu ma minimalną wymaganą wersję języka, a symbole z zależności mają wersję pakietu z pliku lock. Umożliwia to planowanie aktualizacji języka i zależności jako operacji (lista blokad, poprawki deterministyczne, pozycje do przeglądu) oraz pilnowanie, by agent nie generował składni nowszej niż wersja projektu.
- Pascal (dialekty Object Pascal) jako kolejny język adaptera.
- Python jako kolejny język docelowy adaptera (poziom L2 z jawną niepewnością dla konstrukcji dynamicznych).
- Dwa eksperymenty przed decyzją go/no-go: gęstość składni (słowa kluczowe, skróty, symbole) mierzona tokenami na poprawnie wykonane zadanie oraz krótkie uchwyty sesyjne zamiast pełnych identyfikatorów w API agenta.
- Językiem repozytorium (kod, komunikaty, dokumentacja techniczna) jest angielski. Niniejsze opracowanie powstało po polsku i pozostaje wersją wiodącą; wydanie angielskie (`MUL_Design_Paper_EN.tex`) ma tę samą numerację rozdziałów i sekcji.

23 Badanie pilotażowe: tokeny zapisu źródłowego

23.1 Cel i zakres

Rozdział przedstawia pierwszy pomiar wykonany na działającym prototypie. Pytanie badawcze jest wąskie: ile tokenów zajmuje ten sam program zapisany w MUL Source, Go, PHP i C oraz od czego ta liczba zależy. Pomiar dotyczy rozmiaru tekstu, a nie pracy agenta. Nie weryfikuje hipotezy H1 (sekcja 1.4), która mówi o tokenach na poprawnie wykonane zadanie i wymaga eksperymentu z rozdziału 11. Próba jest mała (8 programów, 32 plików), dlatego wyniki należy traktować jako rozpoznanie i kalibrację metody przed większym badaniem. W zapisie liczb użyto kropki dziesiętnej, ponieważ tabele i wykresy są generowane bezpośrednio z danych.

Motywacją jest wcześniejsza praca autora o automatycznym przetwarzaniu życiorysów[1]: na 100 syntetycznych dokumentach modele językowe poprawnie rozpoznawały pola w około 20% przypadków, gdy dostawały tekst wydobyty z PDF, i w około 95%, gdy te same dane miały postać ustrukturyzowaną (JSON-LD). Tamten wynik dotyczy innej dziedziny i nie jest dowodem na hipotezy MUL; uzasadnia jedynie pytanie, czy jawna struktura programu daje podobny efekt w pracy z kodem.

23.2 Materiał i metoda

Materiał. Osiem programów (`collatz`, `digits`, `fib`, `fizzbuzz`, `gcd`, `hello`, `power`, `primes`) zapisano w czterech językach w katalogu `examples/compare/`. Wersje jednego programu mają tę samą strukturę funkcji, te same nazwy i ten sam styl formatowania właściwy dla języka. Równoważność zachowania jest sprawdzana automatycznie: każda wersja musi wypisać to samo i zakończyć się tym samym kodem wyjścia co program MUL (`make compare`). Programy mieszczą się w zakresie języka MVP (sekcja 21.4): liczby `i32`, wartości logiczne, napisy, funkcje, rekurencja, warunki i pętle.

Układy zapisu MUL. Każdy program MUL policzono w dwóch układach,

[1] R. Rabczuk, *The Resume Parsing Crisis of 2025: How Applicant Tracking Systems Fail to Identify Qualified Candidates*, praca niezależna, 20 października 2025, SSRN nr 6061595.

które zawierają te same tokeny języka (ADR-013): w układzie czytelnym dla ludzi (`mul fmt --pretty`: jedna instrukcja w wierszu, wcięcie czterema spacjami, spacje wokół operatorów), na którym opierają się główne tabele, oraz w układzie zwartym, czyli postaci kanonicznej, w której pliki są przechowywane i którą agenci czytają i piszą (spacja tylko między dwoma tokenami słownymi, bez wcięć i bez znaków nowej linii). Go, PHP i C liczono w postaci, w jakiej je zapisano, w formatowaniu zwyczajowym dla każdego z nich. Sekcja 23.5 porównuje je ze zwartym MUL. Listingi w tym dokumencie zachowują układ czytelny.

Tokenizery. Każdy plik policzono pięcioma metodami: (1) estymatorem wbudowanym w `mul tokens`; (2) i (3) słownikami BPE `cl100k_base` i `o200k_base` opublikowanymi przez OpenAI, uruchomionymi lokalnie; (4) i (5) tokenizerami dwóch modeli uruchamianych lokalnie, `qwen2.5-coder:7b` i `gemma3:12b`, przez odczyt długości promptu zgłaszanej przez lokalne środowisko uruchomieniowe. Tokenizera modeli Claude nie zmierzono, ponieważ nie jest dostępny do uruchomienia lokalnego; pomiar ten należy do większego badania. Jako tokenizer główny w tabelach szczegółowych przyjęto o200k_base. Żaden plik nie został wysłany poza komputer, na którym wykonano pomiar.

Miary. Niech $T_L(i)$ oznacza liczbę tokenów programu i w języku L. Dla każdej pary język–MUL obliczono iloraz na program, iloraz łączny i średnią geometryczną ilorazów:

$$r_i = \frac{T_L(i)}{T_{\mathrm{MUL}}(i)}, \qquad \rho = \frac{\sum_i T_L(i)}{\sum_i T_{\mathrm{MUL}}(i)}, \qquad \bar{r}_g = \exp\Big(\frac{1}{n}\sum_{i=1}^{n} \ln r_i\Big). \tag{23.1}$$

Wartość większa od 1 oznacza, że zapis w języku L jest dłuższy niż w MUL. Przedział ufności 95% dla $\bar{r}_g$ wyznaczono dwiema metodami: z rozkładu t Studenta dla logarytmów ilorazów,

$$\exp\Big(\overline{\ln r} \pm t_{0.975,\,n-1}\,\frac{s_{\ln r}}{\sqrt{n}}\Big), \qquad t_{0.975,\,7} = 2.365, \tag{23.2}$$

oraz metodą bootstrap (percentyle, 10 000 losowań ze zwracaniem, ustalone ziarno generatora). Różnice $d_i = T_L(i) - T_{\mathrm{MUL}}(i)$ zbadano dokładnym testem znakowanych rang Wilcoxona; rozkład statystyki W^+ wyznaczono przez przeliczenie wszystkich 2^n układów znaków, więc najmniejsza osiągalna dwustronna wartość p wynosi $2/2^8 = 0.0078$. Wielkość efektu podano jako $d_z = \bar{d}/s_d$. Zależność między językami przybliżono regresją liniową metodą najmniejszych kwadratów,

$$T_L(i) \approx \alpha + \beta\, T_{\mathrm{MUL}}(i), \tag{23.3}$$

z współczynnikiem determinacji R^2. Wszystkie obliczenia wykonuje skrypt `research/token-sample/measure.py`; funkcje statystyczne mają test własny na przypadkach o znanym wyniku.

23.3 Wyniki

Tabela 23.1: Liczba tokenów programów porównawczych (tokenizer o200k_base); wiersze i bajty dotyczą wersji MUL w układzie czytelnym, kolumna „zwarty” tego samego programu w układzie zwartym

Program	Wiersze	Bajty	MUL	zwarty	Go	PHP	C
`collatz`	23	382	125	89	114	120	135
`digits`	24	405	137	98	134	135	149
`fib`	13	187	68	50	66	59	71
`fizzbuzz`	29	501	173	117	166	171	191
`gcd`	17	257	96	67	91	92	105
`hello`	6	90	27	22	19	15	23
`power`	20	439	132	103	128	142	144
`primes`	29	475	151	102	147	155	169
Razem	161	2736	909	648	865	889	987

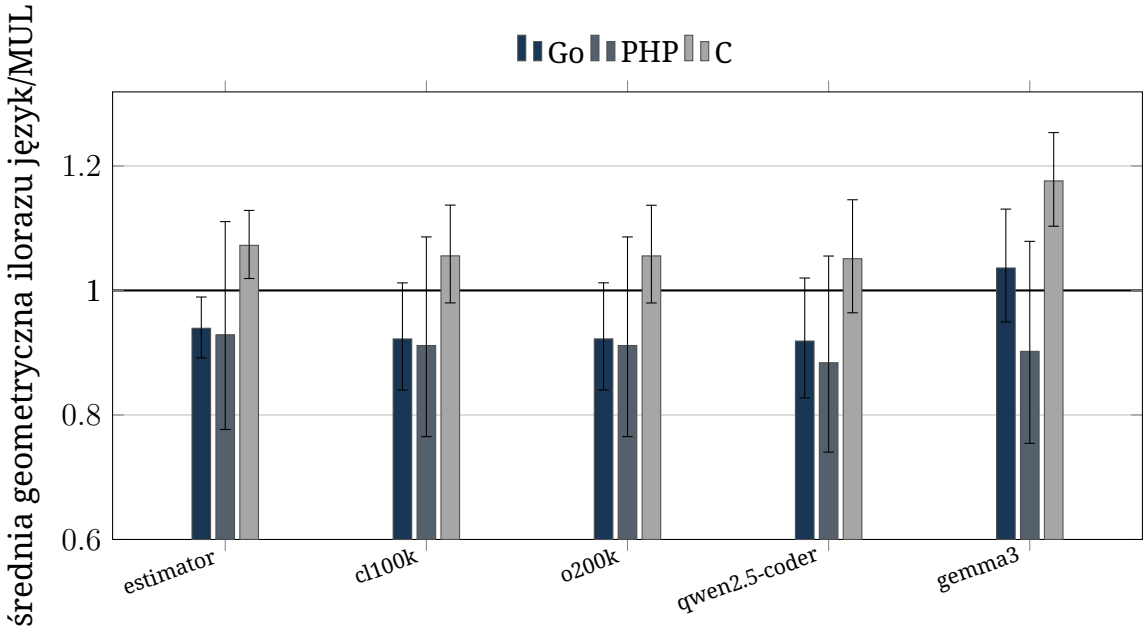
Tabela 23.1 zawiera dane surowe. Łącznie programy MUL mają 909 tokenów w układzie czytelnym (648 w zwartym), Go 865, PHP 889, C 987. Tabela 23.2 podaje ilorazy dla wszystkich tokenizerów, a rysunek 23.1 przedstawia je graficznie.

Dla tokenizera głównego zapis w Go zajmuje średnio $\bar{r}_g = 0.922$ zapisu MUL (PU 95%: 0.840–1.012), w PHP 0.912 (0.765–1.086), a w C 1.055 (0.980–1.137). Zapis MUL nie jest więc w tej próbie krótszy od Go ani PHP; jest o kilka procent krótszy od C. Wszystkie trzy przedziały t obejmują 1. Test Wilcoxona daje dla Go $p = 0.0078$, ponieważ wersja Go jest krótsza we wszystkich ośmiu programach ($W^+ = 0$), lecz różnica jest mała: średnio -5.5 tokena na program. Rozbieżność obu metod wynika z programu `hello`, w którym stały narzut pliku dominuje nad treścią i który zwiększa wariancję logarytmów ilorazów.

Kierunek różnicy względem Go zależy od tokenizera: dla `gemma3:12b` $\bar{r}_g = 1.036$, czyli Go jest dłuższe od MUL. Wynik porównania składni dwóch języków nie jest zatem własnością samych języków, lecz pary język–tokenizer. Przeprowadzono 15 porównań bez poprawki na wielokrotne testowanie; przy poprawce Bonferroniego próg $0.05/15 = 0.0033$ jest niższy od najmniejszej osią-

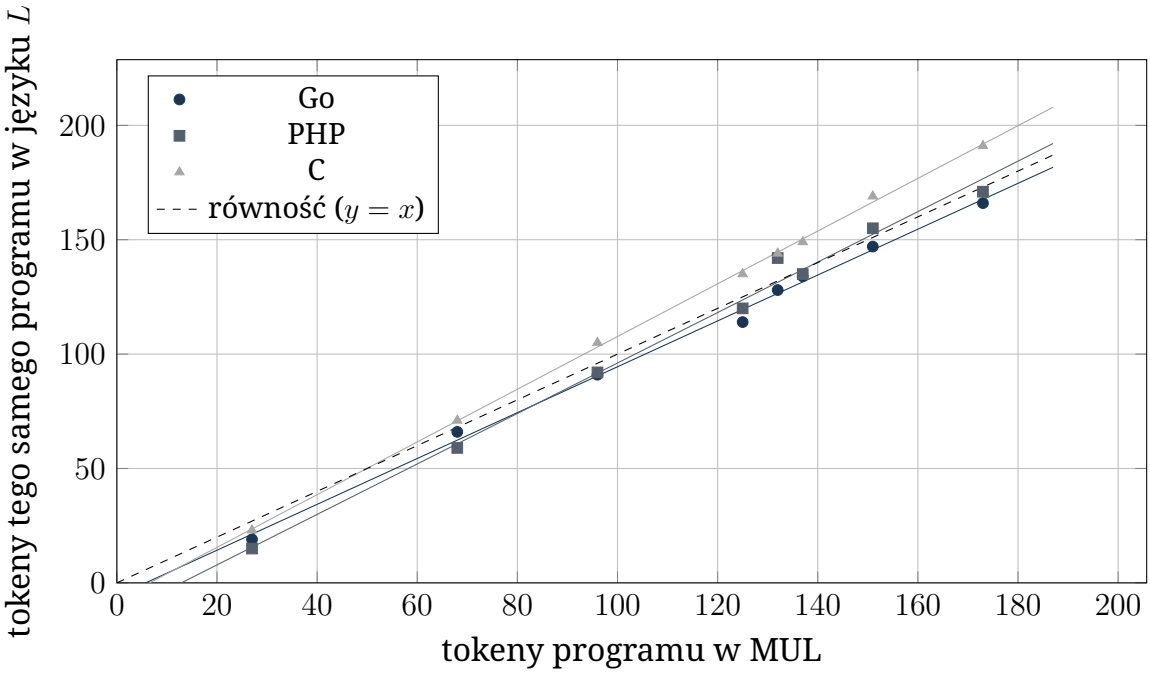
Tabela 23.2: Iloraz liczby tokenów język/MUL: iloraz łączny ρ, średnia geometryczna $\bar{r}_g$, przedziały ufności 95% i test Wilcoxona ($n = 8$)

Tokenizer	Język	ρ	$\bar{r}_g$	PU 95% (t)	PU 95% (bootstrap)	W^+	p
estimate	Go	0.953	0.939	[0.89; 0.99]	[0.90; 0.97]	0	0.0156
estimate	PHP	0.998	0.929	[0.78; 1.11]	[0.80; 1.02]	14.5	0.9688
estimate	C	1.091	1.072	[1.02; 1.13]	[1.03; 1.10]	35	0.0156
cl100k_base	Go	0.952	0.922	[0.84; 1.01]	[0.85; 0.97]	0	0.0078
cl100k_base	PHP	0.978	0.912	[0.77; 1.09]	[0.78; 1.01]	10.5	0.3203
cl100k_base	C	1.086	1.056	[0.98; 1.14]	[0.99; 1.10]	34	0.0234
o200k_base	Go	0.952	0.922	[0.84; 1.01]	[0.85; 0.97]	0	0.0078
o200k_base	PHP	0.978	0.912	[0.77; 1.09]	[0.78; 1.01]	10.5	0.3203
o200k_base	C	1.086	1.055	[0.98; 1.14]	[0.99; 1.10]	34	0.0234
qwen2.5-coder:7b	Go	0.954	0.919	[0.83; 1.02]	[0.84; 0.97]	0	0.0078
qwen2.5-coder:7b	PHP	0.950	0.884	[0.74; 1.06]	[0.76; 0.98]	1.5	0.0469
qwen2.5-coder:7b	C	1.087	1.051	[0.96; 1.15]	[0.97; 1.10]	34	0.0234
gemma3:12b	Go	1.064	1.036	[0.95; 1.13]	[0.96; 1.09]	34	0.0234
gemma3:12b	PHP	0.970	0.902	[0.75; 1.08]	[0.77; 1.00]	9	0.2422
gemma3:12b	C	1.202	1.176	[1.10; 1.25]	[1.11; 1.23]	28	0.0156



Rysunek 23.1: Średnia geometryczna ilorazu liczby tokenów względem MUL z przedziałem ufności 95% (rozkład t). Linia pozioma 1 oznacza równość z MUL.

galnej wartości p, więc żadnej z różnic nie należy uznawać za potwierdzoną statystycznie.



Rysunek 23.2: Liczba tokenów programu w Go, PHP i C względem MUL (tokenizer o200k_base) z prostymi regresji.

Tabela 23.3: Przybliżenie liniowe $T_L \approx \alpha + \beta\, T_{\text{MUL}}$ oraz różnice par (tokenizer o200k_base)

Język	α	β	R^2	$\bar{d}$	s_d	d_z
Go	-5.8	1.002	0.996	-5.5	3.0	-1.85
PHP	-14.2	1.103	0.991	-2.5	7.0	-0.36
C	-7.5	1.152	0.999	9.8	7.4	1.32

Zależność jest w badanym zakresie niemal liniowa (rysunek 23.2, tabela 23.3; R^2 od 0.991 do 0.999). Dla Go nachylenie $\beta = 1.002$ przy ujemnym wyrazie wolnym $\alpha = -5.8$ oznacza, że oba języki rosną w tym samym tempie, a MUL ma stały narzut kilku tokenów na plik (deklaracja modułu, sygnatura `main` z typem wyniku i efektem). Dla PHP i C nachylenie jest większe od 1 (1.103 i 1.152), a wyraz wolny ujemny: krótkie programy są w tych językach krótsze niż w MUL, a dłuższe stają się dłuższe. Punkt zrównania $T^* = -\alpha/(\beta - 1)$ wypada w okolicach 138 tokenów dla PHP i 50 tokenów dla C. Ekstrapolacja poza zakres próby (27–173 tokeny) jest nieuprawniona; przybliżenie służy do zaplanowania większego badania, a nie do prognozy.

23.4 Zmienna kontrolna: wcięcia

Języki różnią się konwencją wcięć (tabulator w Go, cztery spacje w pozostałych), co mogłoby tłumaczyć część różnic. Pomiar powtórzono na plikach pozbawionych wcięć początkowych.

Tabela 23.4: Udział wcięć w liczbie tokenów i iloraz łączny ρ przed usunięciem wcięć i po nim (tokenizer o200k_base)

Język	Wcięcia [%]	ρ	ρ bez wcięć	PU 95% $\bar{r}_g$ bez wcięć
MUL	10.6	1.000	1.000	–
Go	8.3	0.952	0.975	[0.88; 1.03]
PHP	9.6	0.978	0.989	[0.80; 1.09]
C	9.7	1.086	1.096	[0.98; 1.15]

Wcięcia stanowią od 8.3% (Go) do 10.6% (MUL) tokenów. Po ich usunięciu iloraz dla Go zmienia się z 0.952 na 0.975, a dla C z 1.086 na 1.096 (tabela 23.4). Konwencja wcięć odpowiada więc za około połowę różnicy między MUL a Go; kolejność języków pozostaje bez zmian.

23.5 Drugi układ: zwarty MUL

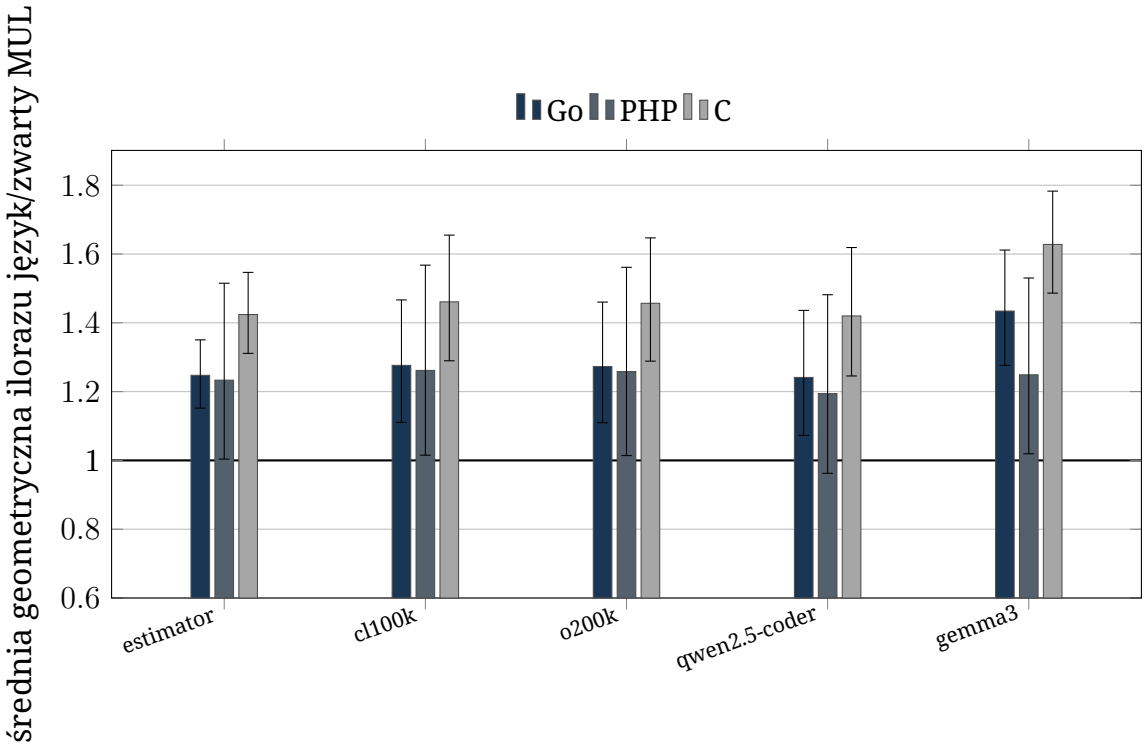
Powyższe porównania używają czytelnego układu MUL, w którym język czytają ludzie. Agenci czytają i piszą układ zwarty (ADR-013), dlatego pomiar powtórzono ze zwartym MUL w mianowniku; zapisy w Go, PHP i C to te same pliki co wcześniej.

Układ zwarty zajmuje o 28.7% mniej tokenów niż czytelny dla tokenizera głównego (od 25.3% do 29.0% w zależności od tokenizera). Względem zwartego MUL Go zajmuje $\bar{r}_g = 1.273$ (PU 95%: 1.110–1.460), PHP 1.258 (1.014–1.561), a C 1.457 (1.289–1.647) (tabela 23.5, rysunek 23.3). Kierunek przestaje zależeć od tokenizera: każdy język jest dłuższy od zwartego MUL dla każdego tokenizera, a 1 obejmuje tylko przedział PHP dla `qwen2.5-coder:7b`.

Porównanie nie jest symetryczne. Zwarty MUL nie ma żadnych opcjonalnych białych znaków, a pozostałe języki zachowują swoje zwyczajowe formatowanie; same wcięcia to 8.3–9.7% ich tokenów (tabela 23.4), a Go, PHP i C pozbawione ich w ten sam sposób również by się skróciły. Wynik mierzy więc decyzję o układzie z ADR-013, czyli postać, w której agenci wymieniają kod MUL, a nie przewagę składni MUL. Nie zmierzono, czy model pisze i edytuje kod w układzie zwartym równie poprawnie jak w czytelnym; to wiersz

Tabela 23.5: Iloraz liczby tokenów język/MUL przy MUL w układzie zwartym: iloraz łączny ρ względem czytelnego i zwartego MUL, średnia geometryczna $\bar{r}_g$ i jej przedział ufności 95% (rozkład t, $n = 8$)

Tokenizer	Język	ρ czytelny	ρ zwarty	$\bar{r}_g$ zwarty	PU 95% (t)
estimate	Go	0.953	1.277	1.247	[1.15; 1.35]
estimate	PHP	0.998	1.336	1.233	[1.00; 1.52]
estimate	C	1.091	1.461	1.424	[1.31; 1.55]
cl100k_base	Go	0.952	1.340	1.276	[1.11; 1.47]
cl100k_base	PHP	0.978	1.377	1.262	[1.02; 1.57]
cl100k_base	C	1.086	1.529	1.461	[1.29; 1.65]
o200k_base	Go	0.952	1.335	1.273	[1.11; 1.46]
o200k_base	PHP	0.978	1.372	1.258	[1.01; 1.56]
o200k_base	C	1.086	1.523	1.457	[1.29; 1.65]
qwen2.5-coder:7b	Go	0.954	1.309	1.241	[1.07; 1.44]
qwen2.5-coder:7b	PHP	0.950	1.305	1.194	[0.96; 1.48]
qwen2.5-coder:7b	C	1.087	1.492	1.420	[1.25; 1.62]
gemma3:12b	Go	1.064	1.490	1.434	[1.28; 1.61]
gemma3:12b	PHP	0.970	1.359	1.249	[1.02; 1.53]
gemma3:12b	C	1.202	1.684	1.628	[1.49; 1.78]



Rysunek 23.3: Średnia geometryczna ilorazu liczby tokenów względem zwartego MUL z przedziałem ufności 95% (rozkład t). Linia pozioma 1 oznacza równość z MUL; por. rysunek 23.1.

7.3 planu prac, a ADR-013 pozostaje do tego czasu warunkowy.

23.6 Koszt referencji języka i punkt zwrotu

Model, który nie zna MUL, musi otrzymać opis języka. Zwarta referencja (`mul guide`) zajmuje 2172 tokenów. Jeżeli kod w języku L jest ρ razy dłuższy niż w MUL, to koszt referencji C_{ref} zwraca się po przeczytaniu przez model kodu o objętości

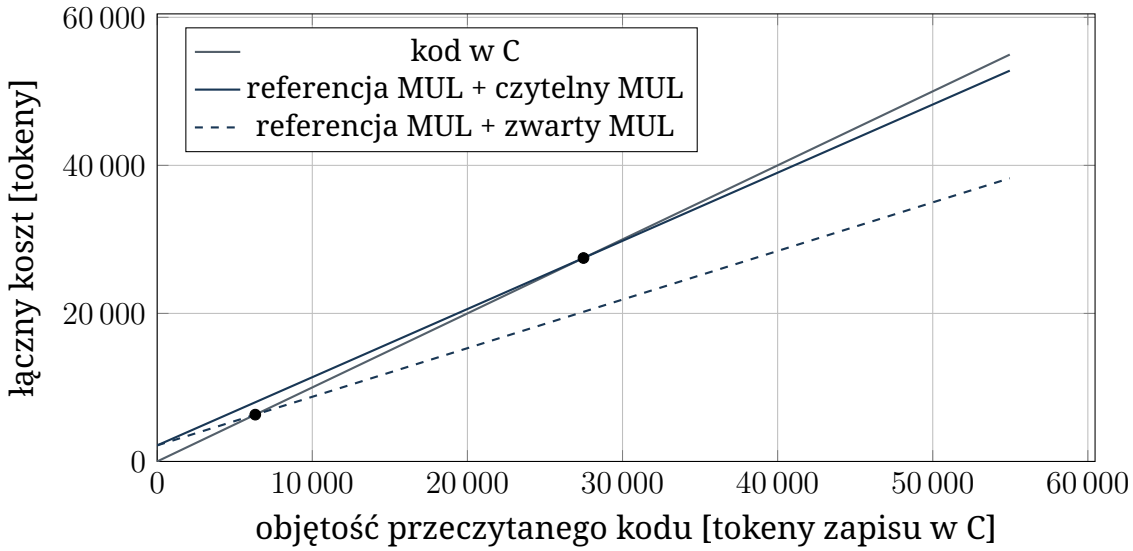
$$V^* = \frac{C_{\text{ref}}}{1 - 1/\rho} \qquad (\rho > 1), \tag{23.4}$$

mierzonej w tokenach zapisu w języku L. Dla $\rho \leq 1$ punkt zwrotu nie istnieje.

Tabela 23.6: Koszt referencji języka C_{ref} i objętość kodu V^* (w tokenach zapisu w języku L), po której się zwraca, przy MUL w układzie czytelnym i zwartym; „–” oznacza brak punktu zwrotu

		MUL czytelny			MUL zwarty		
Tokenizer	C_{ref}	V^* Go	V^* PHP	V^* C	V^* Go	V^* PHP	V^* C
estimate	2253	–	–	27 000	10 400	9 000	7 100
cl100k_base	2166	–	–	27 400	8 500	7 900	6 300
o200k_base	2172	–	–	27 500	8 700	8 000	6 300
qwen2.5-coder:7b	2304	–	–	28 900	9 800	9 900	7 000
gemma3:12b	2408	40 300	–	14 300	7 300	9 100	5 900

Referencja sama jest pisana dla agentów: jej przykłady są w układzie zwartym, a od dodania formatera urosła do 2172 tokenów, co przesunęło w górę każde V^*. Przy czytelnym MUL koszt referencji zwraca się względem C po około 27 500 tokenów kodu, czyli po około 223 programach wielkości średniego programu z próby; względem Go i PHP dla tokenizera głównego nie zwraca się wcale. Przy zwartym MUL zwraca się względem wszystkich trzech języków: po około 8 700 tokenów Go (80 programów), 8 000 PHP (72) i 6 300 C (51; tabela 23.6, rysunek 23.4). Model nie uwzględnia buforowania promptu ani tego, że model wytrenowany na MUL referencji nie potrzebuje. W obecnym stanie sama składnia MUL nie stanowi źródła oszczędności tokenów; stanowi je układ, w którym agenci wymieniają kod, z zastrzeżeniem z sekcji 23.5.



Rysunek 23.4: Łączny koszt odczytu kodu w C oraz w MUL z jednorazowym kosztem referencji języka (tokenizer o200k_base), dla czytelnego i zwartego MUL. Punkty oznaczają V^*.

Tabela 23.7: Liczba tokenów programu MUL i jego tekstowego IR (`mul ir`; tokenizer o200k_base)

Program	Źródło MUL	IR	IR/źródło
`collatz`	125	445	3.56
`digits`	137	373	2.72
`fib`	68	219	3.22
`fizzbuzz`	173	647	3.74
`gcd`	96	196	2.04
`hello`	27	49	1.81
`power`	132	394	2.98
`primes`	151	472	3.13

23.7 Tekstowy IR a zapis źródłowy

Tekstowa postać IR jest średnio 2.82 razy dłuższa od źródła (średnia geometryczna; zakres 1.81–3.74, tabela 23.7). Potwierdza to liczbowo założenie z sekcji 7.1: modelowi nie należy przekazywać surowego IR, lecz wyniki operacji i krótkie identyfikatory.

23.8 Trafność estymatora wbudowanego

Tabela 23.8: Błąd estymatora `mul tokens` względem tokenizerów referencyjnych na 32 plikach źródłowych

Tokenizer referencyjny	Średni błąd bezwzględny [%]	Obciążenie [%]	Błąd największy [%]
cl100k_base	3.9	-1.5	15.8
o200k_base	4.0	-1.6	15.8
qwen2.5-coder:7b	8.2	-6.7	16.8
gemma3:12b	21.8	-21.8	32.3

Estymator działa bez słownika. Po pierwszej rundzie tego badania został skalibrowany na danych z `results.csv`: ciągi interpunkcji liczy po trzy znaki na token, części słów po pięć, znak nowej linii tuż po interpunkcji nic nie kosztuje, wcięcie na początku wiersza jest osobnym tokenem, a jeden ze znaków `( . : # {` tuż przed słowem łączy się z nim, jak w `(n` czy `{return`, co jest częste w zwartym MUL. Pierwsza wersja zawyżała liczbę tokenów względem o200k_base średnio o 11.8% przy średnim błędzie bezwzględnym 13.3% i oceniała Go jako dłuższe od MUL ($\rho = 1.115$), podczas gdy tokenizery BPE oceniały je jako krótsze. Po kalibracji obciążenie wynosi -1.6%, a średni błąd bezwzględny 4.0% (tabela 23.8). Estymator porządkuje teraz języki tak samo jak tokenizery BPE: Go $\rho = 0.953$ wobec 0.952, PHP 0.998 wobec 0.978, C 1.091 wobec 1.086. Na ośmiu programach w układzie zwartym średni błąd bezwzględny wynosi 2.5% (obciążenie $+0.9$%, błąd największy 4.9%), a estymator przypisuje układowi zwartemu oszczędność 25.3% wobec 28.7% dla o200k_base.

Pozostają trzy ograniczenia. Największy błąd, 15.8%, przypada na najkrótsze pliki (`hello` w C i Go, około 20 tokenów), gdzie jeden token to kilka procent. Estymator naśladuje słowniki OpenAI i model `qwen2.5-coder:7b`; dla `gemma3:12b` zaniża wynik o 21.8%, bo ten tokenizer dzieli ten sam tekst na więcej części. Reguły dopasowano też na tych samych plikach, na których je

sprawdzono (32 plikach z tabeli 23.8 i ośmiu plikach zwartego MUL), więc podane błędy są błędami w próbie. Estymator nadaje się do porównywania języków na słownikach BPE, do których go dopasowano; dla innego modelu porównanie powinno używać tokenizera tego modelu (opcja `--tokenizer-cmd`), a `mul tokens` oznacza każdą liczbę szacowaną jako szacunek.

23.9 Zagrożenia dla trafności

- **Wielkość i dobór próby.** Osiem krótkich programów napisanych przez autora projektu; brak losowania i brak kodu z rzeczywistych repozytoriów. Programy są ograniczone do konstrukcji, które MUL już obsługuje, co pomija struktury, kolekcje, obsługę błędów i biblioteki, czyli obszary, w których języki różnią się najbardziej. Wersje programów w języku Rust dodano do repozytorium po wykonaniu pomiaru; wejdą do następnej rundy.
- **Styl zapisu.** Wersje pisano równolegle z zachowaniem tych samych nazw i struktury. Idiomatyczny kod w każdym języku mógłby dać inne ilorazy (np. `for` z licznikiem w Go i C, którego MUL nie ma).
- **Układ zapisu.** Zwarty MUL porównano z pozostałymi językami w postaci, w jakiej je zapisano, a nie z ich postacią zminifikowaną, więc ilorazy z sekcji 23.5 łączą język z konwencją formatowania. Nie zmierzono poprawności kodu pisanego przez model w układzie zwartym.
- **Tokenizery.** `cl100k_base` i `o200k_base` dają na tej próbie niemal identyczne wyniki, więc nie są niezależnymi obserwacjami. Liczby dla modeli lokalnych pochodzą z długości promptu zgłaszanej przez środowisko uruchomieniowe, pomniejszonej o narzut wyznaczony na prompcie jednoznakowym. Tokenizer modeli Claude nie został zmierzony.
- **Kalibracja estymatora.** Reguły estymatora wbudowanego dopasowano do tej próby, więc jego błąd jest tu błędem w próbie. Większe badanie mierzy go na plikach, które nie brały udziału w dopasowaniu.
- **Zakres wniosków.** Mierzono rozmiar tekstu. Nie mierzono poprawności kodu pisanego przez model, liczby tur, kosztu poprawek ani kosztu odczytu kontekstu, które rozstrzygają o hipotezie H1.
- **Niezależność badacza.** Autor pomiaru jest autorem badanego języka. Dane, skrypt i programy są w repozytorium, aby pomiar można było powtórzyć.

23.10 Wnioski i plan większych prób

1. W badanej próbie zapis MUL Source nie jest krótszy w tokenach od Go ani PHP i jest nieznacznie krótszy od C; różnice są rzędu kilku procent, a ich kierunek dla Go zależy od tokenizera.
2. Koszt referencji języka (około 2 tys. tokenów) nie zwraca się samą zwięzłością składni.
3. Zwarty układ MUL zajmuje o 28.7% mniej tokenów niż czytelny; względem niego Go, PHP i C są o 26–46% dłuższe (średnia geometryczna, tokenizer o200k_base), a referencja zwraca się po kilku tysiącach tokenów kodu. Jest to zysk z układu zapisu, porównanego z pozostałymi językami w ich zwyczajowej postaci, i liczy się tylko wtedy, gdy poprawność kodu nie spada (wiersz 7.3).
4. Tekstowy IR jest blisko trzykrotnie dłuższy od źródła.
5. Po kalibracji estymator wbudowany mieści się w kilku procentach od słowników BPE OpenAI i porządkuje języki tak jak one; dla innych modeli porównanie wymaga tokenizera danego modelu.

Wyniki są zgodne z założeniem przyjętym w rozdziale 4, że zwięzłość dla AI ma wynikać z operacji na IR, a nie ze składni, i nie dostarczają argumentu za skracaniem słów kluczowych. Nie mówią nic o hipotezie H1. Kolejne etapy: (1) próba co najmniej 100 programów, w tym fragmenty rzeczywistych repozytoriów i konstrukcje pełnego języka v0.1; (2) tokenizery wszystkich modeli używanych w eksperymencie, w tym modeli Claude; (3) eksperyment gęstości składni z pomiarem poprawności kodu pisanego przez model; (4) pomiar tokenów na poprawnie wykonane zadanie w wariantach A, B i C z rozdziału 11, z progami decyzji ustalonymi przed pomiarem.

Wnioski końcowe

Rozwiązanie MUL ma uzasadnienie badawcze jako próba przesunięcia pracy agentów z interpretacji tekstu kodu na precyzyjne operacje na programie. Krytycznym warunkiem sukcesu nie jest sam zapis wektorowy, lecz weryfikowalny IR, semantyczne kontrakty adapterów i skuteczna walidacja zmian. W pierwszym etapie należy zbudować prototyp Go–MUL–Go i eksperymentalnie porównać skuteczność, zużycie tokenów oraz ryzyko regresji względem standardowej pracy agentów. Dopiero po pozytywnych wynikach uzasadnione będzie rozszerzanie zestawu języków, trenowanie wyspecjalizowanego modelu oraz inwestowanie w backend natywny.

Oświadczenia

Autorstwo

Autorem opracowania, projektu MUL i badania z rozdziału 23 jest Rafał Rabczuk, badacz niezależny (Wielka Brytania). Autor odpowiada za treść, decyzje projektowe i wnioski.

Wykorzystanie narzędzi AI

Praca powstała z pomocą narzędzi sztucznej inteligencji. Przy projektowaniu, pisaniu kodu prototypu, redakcji tekstu i przeglądzie korzystano z kilku modeli językowych: modeli uruchamianych lokalnie, modeli OpenAI oraz modeli Claude firmy Anthropic. Modele nie są autorami. Autor wyznaczał cele i zakres, podejmował decyzje, sprawdzał wyniki i przyjmował lub odrzucał propozycje modeli. Liczby w rozdziale 23 pochodzą z deterministycznych obliczeń skryptu na plikach z repozytorium, a nie z odpowiedzi modeli; modele lokalne posłużyły w pomiarze wyłącznie jako źródło tokenizera.

Dostępność danych

Programy porównawcze (`examples/compare/`), skrypt pomiarowy, dane surowe i wyniki pośrednie (`research/token-sample/`) znajdują się w repozytorium projektu. Pomiar można powtórzyć poleceniem `make token-study`; tabele i wykresy rozdziału 23 są generowane z tych danych.

Konflikt interesów

Autor jest twórcą MUL oraz rozwija MDDB, który w planie eksperymentu (rozdział 11) występuje jako jeden z wariantów porównawczych.