A STUDY IN PROGRAMMING SYSTEMS ARCHITECTURE

MUL

A universal semantic language for artificial intelligence agents

Architecture, intermediate representation,
compiler, code transformations and evaluation methodology

Rafal Rabczuk

Independent Researcher, United Kingdom

A design specification laid out as a scientific and technical dissertation.
A concept document — it is neither a doctoral dissertation nor a description of an experimentally verified system.

English edition of the Polish original `MUL_Rozprawa_Naukowa.tex`.

Editorial version 2.3 | 23 November 2026

Abstract

This work presents the concept of the *Modular Universal Language* (MUL), a programming platform designed to work together with artificial intelligence agents. The proposed system separates reasoning about software changes from the textual syntax of source languages. Central to it is a typed intermediate representation (IR), linked to a symbol graph, source maps and an optional vector index. The document defines the compiler architecture, code emission strategies, a mechanism for safe transformations, integration with MDDB, the role of a local AI model and the design of empirical tests. The hypotheses cover fewer tokens spent on external models, fewer mechanical errors and greater portability of programming operations. The benefits described here are not yet confirmed research results. 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; artificial intelligence; program transformation; Code Graph; semantic analysis; MDDB; language models.

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.

Contents

Abstract i

Streszczenie ii

Introduction and scope 1

1 Vision, problem and definitions 2
- 1.1 The problem MUL solves . . . 2
- 1.2 What MUL means exactly . . . 2
- 1.3 Goals and non-goals . . . 2
- 1.4 Three measurable hypotheses . . . 3

2 Product model and usage modes 4
- 2.1 Mode A: existing code → MUL → patch . . . 4
- 2.2 Mode B: MUL as the source language . . . 4
- 2.3 Mode C: the agent as an operations client . . . 4
- 2.4 Division of responsibilities . . . 4

3 System architecture 6
- 3.1 Repository modules . . . 6
- 3.2 Repositories and licensing . . . 6

4 Design of the MUL Source language 8
- 4.1 Assumptions . . . 8
- 4.2 Minimal constructs of v0.1 . . . 8
- 4.3 Memory system and concurrency . . . 9
- 4.4 Semantics and compatibility . . . 9
- 4.5 Objects without inheritance . . . 9
- 4.6 Language reference for AI . . . 9

5 MUL IR, graph and the vector layer 11
- 5.1 Three representations, three tasks . . . 11
- 5.2 Node structure and stable identity . . . 11
- 5.3 IR: operations, types, flow . . . 12
- 5.4 The IR verifier . . . 12
- 5.5 The .mulb format . . . 12

6 Parsers and importing existing code **13**
6.1 Import pipeline . . . 13
6.2 The Go profile as the first implementation . . . 13
6.3 The C++ profile as the second frontend . . . 13
6.4 Round-trip: definition of success . . . 14

7 The operations engine for AI **15**
7.1 Why operations instead of raw IR . . . 15
7.2 The PLAN → REVIEW → APPLY → VERIFY protocol . . . 15
7.3 Core API operations . . . 16
7.4 Token budget and progressive output . . . 16
7.5 Change safety . . . 17

8 The MUL compiler **18**
8.1 Compilation paths . . . 18
8.2 When the compiler is “our own” . . . 18
8.3 Minimal compiler profile v0.1 . . . 19
8.4 Go/C++/Rust generators . . . 19
8.5 Runtime, standard library, ABI . . . 19
8.6 Optimizations and debugging . . . 19
8.7 CLI example . . . 19

9 Local AI model and GGUF **21**
9.1 Role of the model . . . 21
9.2 Training pipeline . . . 21
9.3 Training data . . . 21
9.4 A leak-free dataset . . . 21
9.5 Model routing . . . 22
9.6 Local model metrics . . . 22

10 Integration with MDDB and agents **23**
10.1 Integration contract . . . 23
10.2 Synchronization . . . 23
10.3 Agent tool interface . . . 23
10.4 End-to-end example: bump 2.19.13 → 2.19.14 . . . 23

Methodology and evidentiary rigor **25**

11 Tests and benchmarks **26**
11.1 Measurement principle . . . 26

11.2 Three baselines . . . 26
11.3 Test matrix . . . 26
11.4 KPIs and formulas . . . 27
11.5 Proposed quality gates . . . 27
11.6 Experiment design . . . 28
11.7 An illustration of benefits, not a measurement result . . . 28

12 Security, privacy and reliability 30
12.1 Threat model . . . 30
12.2 Determinism and evidence . . . 30
12.3 Language versioning . . . 30

13 Capabilities and limitations map 31
13.1 Extension: PHP, Java, Ruby, TypeScript, and JavaScript . . . 32
13.2 MUL Language Compatibility Layer (LCL) . . . 33
13.3 Parser contract: analysis is not translation . . . 34
13.4 Semantics: dynamic and static languages in one IR . . . 35
13.5 Backend and compiler for new languages . . . 36
13.6 AI task profiles and token minimization . . . 36
13.7 Tests, KPIs, and a realistic rollout order . . . 37
13.8 Implementation decisions and the compatibility contract . . . 38
13.9 Language adapter as a separate module . . . 39
13.10 Object orientation in foreign languages . . . 40

14 Implementation roadmap 41
14.1 Minimal team and roles . . . 41
14.2 Backlog for the first 12 weeks (indicative) . . . 42

15 Product success criteria and the go/no-go decision 43
15.1 Conditions for continuing the project . . . 43
15.2 Conditions for stopping / changing direction . . . 43

16 Reference examples 44
16.1 Function refactoring without rewriting sources . . . 44
16.2 New MUL code emitted to Go . . . 44
16.3 Unsupported translation . . . 44
16.4 Token report . . . 44

17 Design decisions pending approval 46

18 Glossary **47**

19 Technical sources and inspirations **48**

20 Executive summary **49**

21 Technology decision: implementation of the MUL compiler **50**
21.1 Choice: Rust for the compiler core 50
21.2 Backend strategy: Cranelift → LLVM 51
21.3 Components and module boundaries 51
21.4 MVP scope and acceptance criteria 52
21.5 Longer-term outlook: own CPU backend and self-hosting 53
21.6 Risks and decision 53

22 Implementation status (November 2026) **54**
22.1 Completed 54
22.2 Deviations from the roadmap in Chapter 14 55
22.3 Decisions added in version 2.1 55

23 Pilot study: tokens of the source text **57**
23.1 Aim and scope 57
23.2 Material and method 57
23.3 Results 59
23.4 Control variable: indentation 62
23.5 Second layout: compact MUL 62
23.6 Cost of the language reference and the break-even point 64
23.7 Textual IR against the source 66
23.8 Accuracy of the built-in estimator 66
23.9 Threats to validity 67
23.10 Conclusions and the plan for larger samples 68

Concluding remarks **70**

Declarations **71**

List of Figures

3.1 MUL reference architecture. The structural IR is the source of truth, and vectors serve as an auxiliary index. 7

7.1 Change protocol: planning, review, application, verification. Each phase has a separate contract. 15

8.1 Planned compilation paths: source emitters and two interchangeable native backends. 18

13.1 Compatibility layer. The universal core does not remove semantic differences between languages. 31

23.1 Geometric mean of the token ratio relative to MUL with the 95% confidence interval (t distribution). The horizontal line at 1 marks equality with MUL. 60

23.2 Token count of a program in Go, PHP and C against MUL (tokenizer o200k_base) with regression lines. 61

23.3 Geometric mean of the token ratio relative to compact MUL with the 95% confidence interval (t distribution). The horizontal line at 1 marks equality with MUL; compare figure 23.1. 63

23.4 Total cost of reading code in C and in MUL with the one-off cost of the language reference (tokenizer o200k_base), for pretty and compact MUL. The dots mark V^*. 65

List of Tables

1.1 Goals and non-goals 2
1.1 Goals and non-goals 3

2.1 Division of responsibilities 5

5.1 Three representations, three tasks 11

7.1 Core API operations 16

9.1 Model routing 22

11.1 Three baselines 26
11.2 Test matrix 26
11.2 Test matrix 27
11.3 Proposed quality gates 28
11.4 An illustration of benefits, not a measurement result 29

13.1 Capabilities and limitations map 31
13.1 Capabilities and limitations map 32
13.2 Extension: PHP, Java, Ruby, TypeScript, and JavaScript 32
13.2 Extension: PHP, Java, Ruby, TypeScript, and JavaScript 33
13.3 Parser contract: analysis is not translation 34
13.3 Parser contract: analysis is not translation 35
13.4 Backend and compiler for new languages 36
13.5 AI task profiles and token minimization 37
13.6 Tests, KPIs, and a realistic rollout order 38

14.1 Implementation roadmap 41

17.1 Design decisions pending approval 46

18.1 Glossary 47

21.1 Choice: Rust for the compiler core 50
21.2 Components and module boundaries 51
21.2 Components and module boundaries 52

23.1 Token counts of the comparison programs (tokenizer o200k_base); lines and bytes refer to the MUL version in the pretty layout, the column “compact” to the same program in the compact layout 59
23.2 Token ratio language/MUL: aggregate ratio ρ, geometric mean $\bar{r}_g$, 95% confidence intervals and the Wilcoxon test ($n = 8$) . . . 60
23.3 Linear approximation $T_L \approx \alpha + \beta\, T_{\text{MUL}}$ and paired differences (tokenizer o200k_base) 61
23.4 Share of indentation in the token count and the aggregate ratio ρ before and after removing it (tokenizer o200k_base) 62
23.5 Token ratio language/MUL with MUL in the compact layout: aggregate ratio ρ against pretty and against compact MUL, geometric mean $\bar{r}_g$ and its 95% confidence interval (t distribution, $n = 8$) 64
23.6 Cost of the language reference C_{ref} and the code volume V^* (in tokens of the text in language L) after which it is recovered, with MUL in the pretty and in the compact layout; “–” means no break-even point 65
23.7 Token count of a MUL program and of its textual IR (`mul ir`; tokenizer o200k_base) 66
23.8 Error of the `mul tokens` estimator against reference tokenizers on 32 source files 67

Introduction and scope

Main thesis

MUL is meant to be an independent, AI-oriented programming platform: a program can be represented as a typed, verifiable semantic representation on which agents perform short operations. The MUL engine analyzes dependencies locally, prepares and applies changes, and either emits code in a target language or compiles the program directly. The business goal is to reduce the number of tokens spent on external models while keeping or improving the success rate of programming tasks.

Status and assumptions

This is a design specification, not a description of an existing, finished MUL. The example instructions, formats, interfaces, numeric values and test thresholds are proposals to be validated. There are no benchmark results here yet; every potential benefit is marked as a hypothesis.

The architectural decision in brief

- MUL is built in a separate repository and does not depend on MDDB or on any particular AI model.
- The core is a precise, typed IR and the Code Graph. Embeddings are an auxiliary index, not the source of truth about the program.
- Existing code is modified through safe patches and source maps; new MUL code can be emitted to Go/C++/Rust in full.
- The in-house compiler grows in stages: frontend + IR + LLVM backend, and only later, if at all, a machine code generator.
- A local model may plan MUL operations; correctness is decided by validators, compilers and tests.

1 Vision, problem and definitions

1.1 The problem MUL solves

Today's coding agents often spend tokens on reading the same files repeatedly, browsing grep/sed output, resolving symbols, generating whole blocks of text and analyzing diffs. Semantic indexes speed up locating code, but if they return a similar number of characters to the LLM, the token count stays about the same. MUL moves the mechanical and deterministic steps into a local engine.

1.2 What MUL means exactly

MUL = Modular Universal Language (working name). It is at once: (a) a source language for new applications, (b) an internal program representation, (c) a protocol of short edit and analysis operations, and (d) a compilation platform for many targets. "Vector Language" may work as a marketing name, but embeddings alone are not suitable for lossless code storage.

1.3 Goals and non-goals

Table 1.1: Goals and non-goals

Goal	Definition / constraint
Token minimization	Measuring total input/output tokens sent to the external model per completed task, including retries.
Correctness	Preserving semantics in a defined subset, plus tests and compilation after a modification.

Table 1.1: Goals and non-goals

Goal	Definition / constraint
Language independence	Common operations + explicit language dialects and backend limitations.
Safety	Transactions, preconditions, sandbox, no silent changes outside the scope.
Non-goal v1	Automatic, lossless translation of arbitrary C++ to Go.
Non-goal v1	A vector model as the sole source of code or a guarantee of a correct program.

1.4 Three measurable hypotheses

1. H1: Agent + MUL uses at least 50% fewer external tokens than the same agent with grep/sed on mechanical tasks, with no loss of correctness.
2. H2: Agent + MUL solves more repository tasks on the first attempt thanks to structural operations, at a comparable execution budget.
3. H3: A small local model reduces the number of escalations to a large model, without degrading quality on the test set.

The thresholds are experimental targets, not a product promise.

2 Product model and usage modes

2.1 Mode A: existing code → MUL → patch

For existing Go/C++/Rust repositories, the source files remain authoritative. The frontend builds the CST/AST, a symbol index, types and an IR with references to specific file ranges. Changes are made in the IR, and then the patch generator modifies only the relevant places. The compiler and tests verify the result. This is the recommended initial mode.

2.2 Mode B: MUL as the source language

New programs are written directly in MUL. Its files are authoritative, and backends generate Go, C++ or other languages. A native backend targeting LLVM IR is also possible. Each target declares a capability profile; code that uses an unsupported feature gets an error, not an approximation.

2.3 Mode C: the agent as an operations client

Claude, GPT, a local model or an IDE sends a short semantic command, e.g. rename_symbol or bump_project_version. MUL returns a summary, a plan identifier, ambiguous cases and validation results, without delivering all the sources to the context window.

2.4 Division of responsibilities

Table 2.1: Division of responsibilities

Layer	Responsibility
AI agent	Intent, strategy, decisions on ambiguities, analysis of business logic.
MUL Command	Typed, short instructions, constraints and preconditions.
MUL Engine	IR, symbol resolution, plan, apply, rollback, validation.
Backend	Emission of source code or LLVM IR.
Build system	Compilation, tests, lint, static analysis.
MDDB (optional)	Semantic search, repository graph and detection of the change scope.

3 System architecture

3.1 Repository modules

The repository is a single monorepo. The core is a Rust workspace split into crates with a single responsibility each; items marked as planned are built in later waves of the backlog, not as parallel versions of existing modules.

```
mul/
crates/mul-syntax/ # lexer, parser, AST, in-source diagnostics
crates/mul-sema/ # symbols, types, effects
crates/mul-ir/ # typed SSA IR, lowering from HIR
crates/mul-verifier/ # IR verification before the backend
crates/mul-interp/ # reference IR interpreter
crates/mul-backend-cranelift/ # JIT and native code
crates/mul-cli/ # the mul binary
frontends/ # language adapters as separate processes (planned)
spec/ # normative RFCs, one per topic
docs/adr/ # architecture decisions
docs/language/ # language reference for humans and AI (planned)
tests/golden/ # golden cases and IR snapshots
examples/ # examples run by make demo
site/ # documentation site and demo (planned)
```

Modules from the original plan (transformations, runtime, `.mulb`, model, integrations) arrive as further crates in the same workspace.

3.2 Repositories and licensing

The engine, compiler, specification, documentation and language adapters are maintained in a single monorepo; shared versioning simplifies compatibility tests between modules. Model training materials may be split out later if they require a different license. The license should be chosen after an audit of the LLVM, parser and model dependencies. Training data requires documented provenance and license checks.

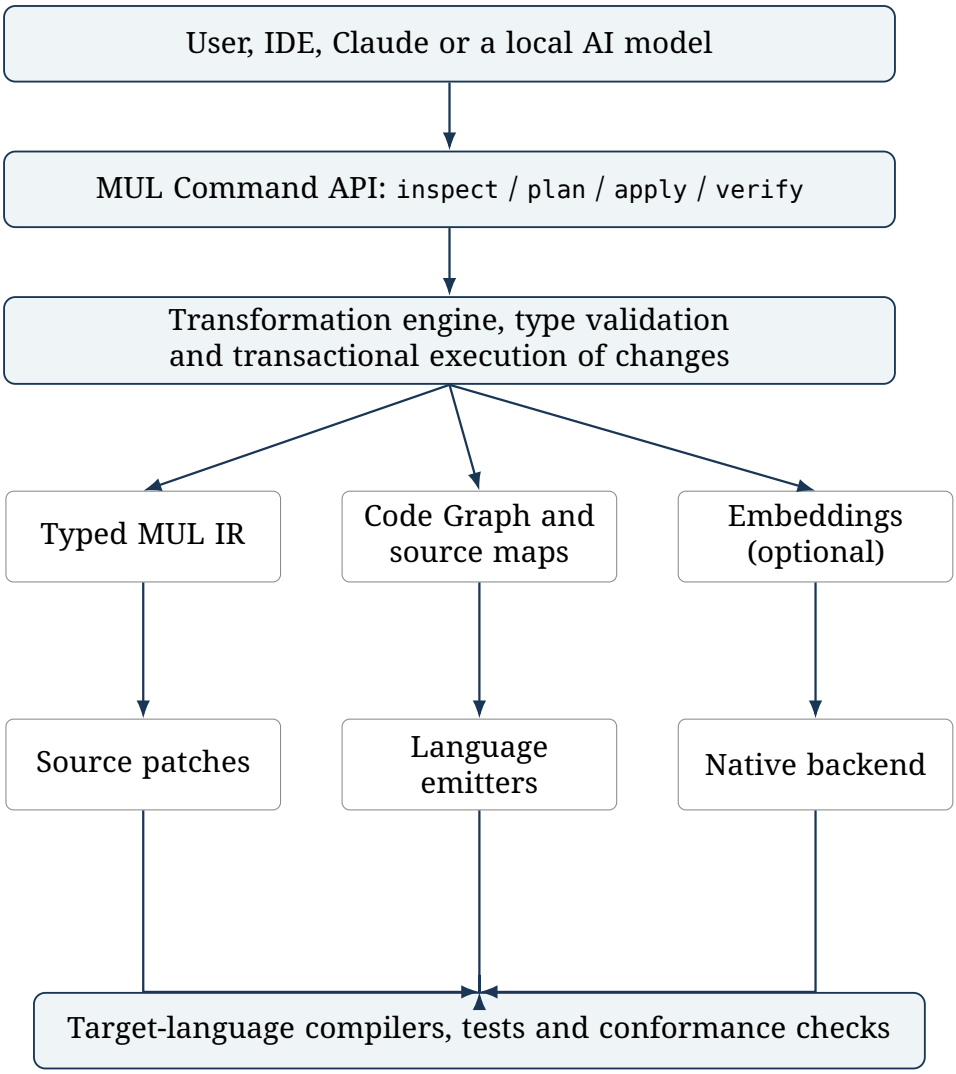

Figure 3.1: MUL reference architecture. The structural IR is the source of truth, and vectors serve as an auxiliary index.

4 Design of the MUL Source language

4.1 Assumptions

MUL Source is a readable, statically typed language with explicit side effects, modules, functions, values, structs and interfaces. Its syntax is meant to be simpler than C++, but not at the cost of ambiguous semantics. Conciseness for AI is achieved primarily through operations on the IR, not through cryptic symbols.

```
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 Minimal constructs of v0.1

- Module/import/export; let/var; primitives bool, i32, i64, f64, string; structs, enum, option/result.
- Functions, blocks, if, match, while, return, calls and basic generics after later stabilization.
- Clear rules for type conversions, errors, overflow and evaluation order.
- Effects: pure, io, alloc, async, unsafe; an early prototype may support the pure/io subset.

4.3 Memory system and concurrency

The contradictory rules of all backends must not be inherited at once. Proposed v1 model: values and references with an explicitly specified lifetime for resources, no implicit sharing of mutable state, and optional managed objects. The Go backend can map managed objects to the GC; the C++ backend will use RAII/smart pointers; the native runtime will provide its own implementation. The concrete semantics must be written down in the specification and backed by tests; this is a separate research program.

4.4 Semantics and compatibility

The word “universal” does not mean full interchangeability of languages. MUL defines a behavior contract. Each frontend emits common operations where they preserve meaning, and extension dialects where a construct is specific. Each backend publishes a capability matrix, e.g. reflection, unsafe pointers, exceptions, async, threads, generics. Unsupported elements block emission.

4.5 Objects without inheritance

MUL Source has no classes or implementation inheritance. Behavior is attached to structs through methods, and polymorphism is expressed with interfaces and composition. A call through an interface is explicitly marked as dynamic in the IR, and a method call on a concrete type as static; both forms have a direct counterpart in emission to Go. Class hierarchies from other languages are not “flattened” into MUL Source; their model is described in Section 13.10.

4.6 Language reference for AI

Before a specialized model exists, agents must be able to write MUL from documentation. Each language feature is introduced together with: a formal grammar (`spec/grammar.ebnf`) consistent with the parser, a description in `docs/language/` (syntax, types, effects, error codes, idioms and an explicit list of constructs the language does not have) and an example. A compact version that fits in the model’s context is available as `mul guide` and `llms.txt`, and later

as an API tool. A test ensures that every example from the documentation passes `mul check` and `mul run`. The same content is the starting point for the training data of the local model.

Normative RFCs cover single topics: an accepted RFC admits only dated errata, and a behavior change requires a new RFC that marks the previous one as superseded. `docs/language/` merges the accepted RFCs into one current description.

5 MUL IR, graph and the vector layer

5.1 Three representations, three tasks

Table 5.1: Three representations, three tasks

Representation	Goal	Is it lossless?
CST + Source Map	Original tokens, comments, formatting, paths and ranges	Yes, in the declared source mode.
Typed Semantic IR + Code Graph	Types, values, symbols, dependencies, CFG, effects	Semantically, for the supported subset.
Embeddings	Similarity of functions, intents, descriptions	No. They are not used to reconstruct code.

5.2 Node structure and stable identity

```
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:...
```

Identifiers must be stable across changes, but the system should support identity changes and identifier migration. The content hash, the symbol identifier and the snapshot version identifier are stored separately. It must not be assumed that a line shift will not change any index.

5.3 IR: operations, types, flow

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

The syntax above is only a proposal for a textual IR dump. It must be decided whether to use MLIR as a host for MUL dialects or to design an independent IR. A custom format makes sense only if measurements justify the maintenance cost.

5.4 The IR verifier

- Every value has a type, and every use refers to an existing definition or parameter.
- SSA dominance, correctness of blocks and terminators, consistency of symbols and visibility.
- Consistency of effects, contracts and target constraints.
- Source maps tied to the correct hash of the input files.
- Deterministic serialization, conforming to the schema and tolerant of unknown fields.

5.5 The .mulb format

Binary IR with a header: magic bytes, schema version, flags, target capabilities, content hash and the sections symbol/type/constant/operation/edge/source-map. Required: size limits, integrity checks, parser fuzzing and schema migrations. A binary encoding saves disk/transmission, but it does not reduce tokens if it is converted to long text and given to an LLM.

6 Parsers and importing existing code

6.1 Import pipeline

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

Tree-sitter is well suited to an incremental CST. However, the CST alone does not resolve types or references. The Go frontend should use the Go ecosystem's tooling for type analysis, and the C++ frontend the Clang infrastructure for the semantics of templates, macros and overloads.

6.2 The Go profile as the first implementation

To handle: packages, imports, declarations, functions/methods, constants, structs, interfaces, calls, and basic generics in later stages. Mutating operations at the start: changing a literal, renaming a symbol, adding a parameter only within a closed scope, and updating the version. Generate a minimal patch; do not rewrite the whole repository.

6.3 The C++ profile as the second frontend

C++ has macros, conditional compilation, templates, UB and non-standard compiler extensions. A parser without the compilation configuration (compile_commands.json) gives an incomplete picture. Initially read-only mode and very limited transformations. Do not declare full round-trip compatibility without separate tests.

6.4 Round-trip: definition of success

Patch mode: byte-for-byte preservation of untouched ranges and preservation of the semantics of changed ranges in the declared profile. Emission mode: equivalent behavior on a defined set of inputs and language contracts, but not identical source text. Comments and formatting may be preserved in patch mode, but need not be identical in a full emitter.

7 The operations engine for AI

7.1 Why operations instead of raw IR

The model should receive concise objects and plan identifiers, not thousands of lines of IR. The mechanics cover deduplication, classification, planning, applying changes, validation and rollback. The model resolves only ambiguous matters. By default it has no right to generate a patch without verified preconditions.

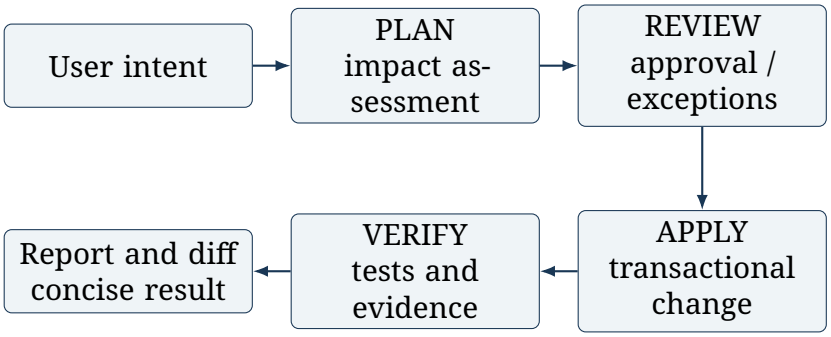

Figure 7.1: Change protocol: planning, review, application, verification. Each phase has a separate contract.

7.2 The PLAN → REVIEW → APPLY → VERIFY protocol

```
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 Core API operations

Table 7.1: Core API operations

Operation	Result	Conditions
inspect(scope)	Concise map of symbols/types	Result budget and pagination.
find(query)	List of identifiers and a ranking	No completeness guarantee when vectors are used.
refs(symbol)	Exhaustive uses in the known index	Snapshot, analysis coverage.
plan(transform)	Plan, diff hash, risks	No modifications.
apply(plan_id)	Transactional changes	Preconditions and snapshot still current.
verify(plan_id)	Tests, typing, build, policy	Report and logs on demand.
emit(target)	Code / artifact / diagnostics	Capability profile.
rollback(change_id)	Reverting changes	Conflicts detected explicitly.

7.4 Token budget and progressive output

Every tool supports detail=summary|references|source, max_items, token_budget, cursor and result_id. A response budget is not a budget for the whole conversation. The full result may be stored in an artifact that the agent refers to by identifier. For a request for “all references”, the full list must be available outside the prompt, even if only a summary reaches the model.

7.5 Change safety

Every mutation has an expected old value, the base file hash, an idempotent operation or a change_id, a scope limit, dry-run and an audit log. For concurrent changes, use optimistic concurrency; conflicts block apply. High-risk operations require user approval. The AI plugin must not run arbitrary shell commands without separate, controlled consent.

8 The MUL compiler

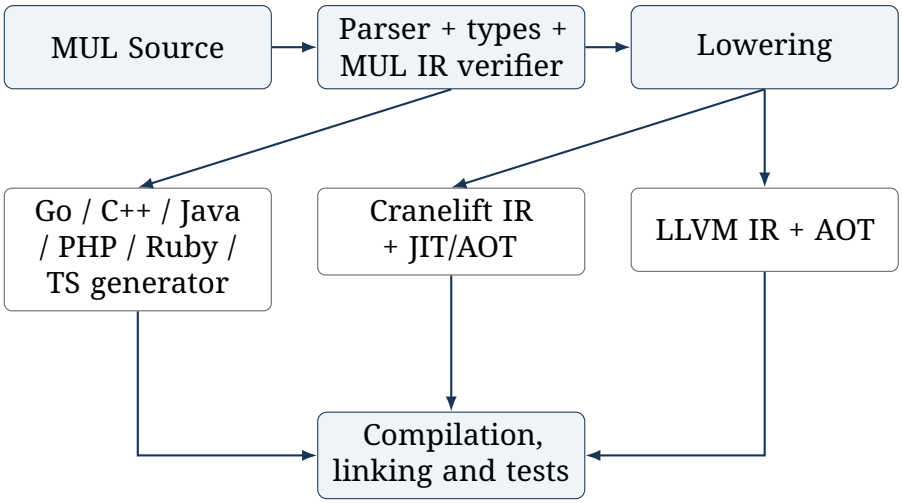

Figure 8.1: Planned compilation paths: source emitters and two interchangeable native backends.

8.1 Compilation paths

```
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 When the compiler is “our own”

A custom parser, type checker, semantics, IR and lowering to LLVM make up a full-fledged MUL compiler. A custom assembler or instruction selector is not required. LLVM provides proven optimization and machine backends; it reduces risk, although it requires integration and maintaining version compatibility.

8.3 Minimal compiler profile v0.1

Programs without complex concurrency: functions, arithmetic, if/while, structs, string, IO through a restricted runtime, basic errors. MUL programs can be emitted to Go. A stable semantic model is required before adding generics, async and exceptions.

8.4 Go/C++/Rust generators

Each generator implements lowering of High IR to constructs of the target language, name mangling, import resolution, library handling and diagnostics mapping. Go comes first; C++ second; Rust later, because ownership/borrowing requires semantic solutions that cannot be added by a formatter alone.

8.5 Runtime, standard library, ABI

Specifications are needed for data layout, String/Array/Map, allocation, panic/error, FFI ABI, module initialization, concurrency and exceptions. A Go-based runtime can use the Go runtime, but the native LLVM backend needs its own or an explicitly chosen runtime. ABI compatibility is an independent test suite.

8.6 Optimizations and debugging

- Flow: constant folding, DCE, inlining, CFG simplification, effect analysis; only when they preserve semantics.
- Source maps from MUL to backend code and back; errors point to the MUL file and symbol.
- Debug/release profiles; deterministic builds, reproducible artifacts and differential tests between targets.

8.7 CLI example

```
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 Local AI model and GGUF

9.1 Role of the model

The local model is a planner and operation classifier, not a substitute for the type system or the compiler. It receives a minimal symbol summary, selects a typed instruction, and escalates to a large LLM when uncertain. Its output is constrained by a schema/grammar. The model can run offline from GGUF through llama.cpp; GGUF is an inference format, not a training method.

9.2 Training pipeline

```
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 Training data

Categories: (1) deterministic transformations, (2) reference classification, (3) fixes after compilation errors, (4) logic changes that require tests, (5) detecting missing information and refusing to execute. For hard cases, use change documentation and PR pairs, but do not equate a merged PR with error-free ground truth. Store the license, commit, hash and validation results.

9.4 A leak-free dataset

Split by whole repositories and by time, not by individual functions. Remove duplicates and near-duplicate examples. Keep a separate holdout from projects not used in training. Measure generalization across languages and compare with the untuned base model.

9.5 Model routing

Table 9.1: Model routing

Task	First executor	Escalation
Version change, rename, refs	Deterministic MUL Engine	Only ambiguities.
Transformation from a natural-language description	Local 3B–8B (hypothesis)	Large model when confidence is low.
Design of a new algorithm	Large model	The local model can prepare artifacts.
Correctness verification	Compiler, tests, verifier	The model analyzes errors only after evidence has been collected.

9.6 Local model metrics

Schema-valid rate, tool selection accuracy, plan success, pass@1 after compilation and tests, number of escalations, tokens/accepted task, GPU RAM usage and time; for privacy, separate tests for leakage of secret data from the index and for prompt injection in code comments.

10 Integration with MDDB and agents

10.1 Integration contract

MDDB indexes and proposes symbol identifiers and potentially related locations. MUL verifies precise references in its own model. The adapter interface has the methods ingest, resolve, analyze, plan, apply, verify and get_artifact. When MDDB is absent, MUL still works on the local repository.

10.2 Synchronization

The events file changed, branch switched and build flags changed cause invalidation and reindex. Every result contains a snapshot_id. Embeddings may remain temporarily stale, but they must not be used as evidence for apply. Loss of index consistency ends in an error, not a silent patch.

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 End-to-end example: bump 2.19.13 → 2.19.14

4. The agent sends the version.bump(patch) intent with the repository scope.
5. MUL identifies authoritative manifests, lockfiles and dependent references; MDDB optionally proposes semantic candidates.

6. The engine separates certain changes from historical/ambiguous cases that may not require an update.
7. The agent receives a short summary and only the exceptions; it approves the plan.
8. MUL applies the patches on a fresh snapshot, runs gofmt, build and tests, and returns result_id and status.

When the user wants to see all changes, they get the full diff as an artifact, without having to put it into the LLM context.

Methodology and evidentiary rigor

The description of MUL has the status of an architectural proposal and a research program, not a deployment report. Claims about fewer tokens, reliability and code portability are *hypotheses* and must not be treated as experimental data. Verification should rely on controlled comparisons, tasks from an independent test set, measurement of all model calls, and reproducible build environments. In particular, what should be compared is the cost of a *correctly completed* task, not just the size of a single agent response.

11 Tests and benchmarks

11.1 Measurement principle

All variants get identical tasks, repositories, parent model, attempt limit, test access and execution budget. Compare end-to-end success, not individual successful calls. The token total includes the large model's input and output, repeated reads, retries, and cached context at its actual cost. Report local model tokens separately; do not hide them.

11.2 Three baselines

Table 11.1: Three baselines

Variant	Tools	Purpose
A: Agent + shell	grep/rg/sed/read/edit/test	Reference point.
B: Agent + MDDB	Search, graph, vectors + plain edits	Isolates the value of indexing.
C: Agent + MDDB + MUL	IR operations, plan/apply/verify	Isolates the value of transformations.
D: C + local MUL model	Routing tasks to a small model	Isolates API savings and local cost.

11.3 Test matrix

Table 11.2: Test matrix

Category	Example	Success condition
Mechanical	Bump version, rename, import rewrite	Correct modifications, no unnecessary changes.

Table 11.2: Test matrix

Category	Example	Success condition
Structural	Signature change across a whole module	Compilation, tests, complete references.
Logic	Cache with invalidation, race condition fix	Hidden functional tests.
Cross-language	MUL → Go and C++	Equivalent results on an input corpus.
Safety	Stale snapshot, symlink, macro ambiguity	Safe refusal or an explicit conflict.
IR	Malformed .mulb, invalid types	Verifier rejects, no crash.
Model	Unknown symbol, contradictory intent	Abstention / escalation.

11.4 KPIs and formulas

```
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
```

The key comparison metric: external model tokens per correctly solved task. A token reduction alone, with a drop in quality, is not a success.

11.5 Proposed quality gates

Table 11.3: Proposed quality gates

Milestone	Proposed pass criterion
MVP Go	100% of verifier tests pass for the supported profile; zero silent mismatches.
Refactor benchmark	95% success on mechanical transformations on the holdout (target, to be verified).
Token benchmark	50% reduction in external tokens per correct mechanical operation.
Quality regression	End-to-end success no worse than the baseline on the same tasks.
Native compiler	Programs from the v0.1 profile pass differential tests vs the Go backend.
Model local	At least 30% fewer escalations while preserving effectiveness (experimental target).

11.6 Experiment design

Start with 50 mechanical tasks and 50 logic tasks on several separate Go repositories, then expand to 300+. Run each task with at least three seeds for a stochastic model; publish the median and bootstrap confidence intervals. For harder tasks, use the SWE-bench approach and multilingual datasets, but do not treat tests as irrefutable proof of correctness.

11.7 An illustration of benefits, not a measurement result

Table 11.4: An illustration of benefits, not a measurement result

Example scenario	External tokens	Correctness
A: shell	20 000	Must be measured
B: MDDB	12 000	Must be measured
C: MUL	3 500	Must be measured
D: MUL + local	900	Must be measured

The values above serve only to show the mechanism. They are neither a forecast nor a test result.

12 Security, privacy and reliability

12.1 Threat model

- Prompt injection in comments, README files, variable names or external dependencies must not be able to gain permission for shell operations.
- Vector data and logs must not accidentally send private code to an external model.
- Emitted code is an untrusted artifact: compilation and tests run in a sandbox with CPU, memory, network and time limits.
- The .mulb format must be robust against corrupted files and malicious structure sizes.
- Apply operations follow the principle of least privilege, with an allowlist of file scope, an audit and rollback.

12.2 Determinism and evidence

Every transformation records change_id, the engine version, the rule, symbols, input/output hashes, test results, the user's decision and a concise justifying trace. The agent can ask for any step to be expanded by its identifier, instead of pasting the full log.

12.3 Language versioning

SemVer for the specification, with separate versions for the binary IR and the command protocol. Breaking changes require a migrator and compatibility tests. The file format should reject unknown required features and tolerate unknown optional metadata.

13 Capabilities and limitations map

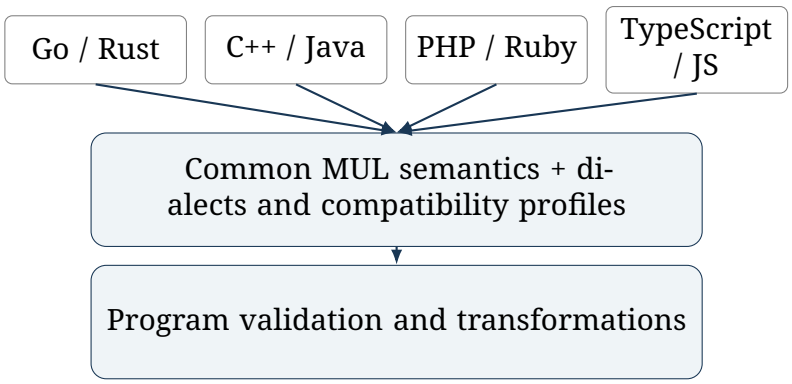

Figure 13.1: Compatibility layer. The universal core does not remove semantic differences between languages.

Table 13.1: Capabilities and limitations map

Capability	MVP	Full vision	Verification
Editing values, versions, rename	Yes (Go)	Multiple languages	Compilation + tests + diff.
Structural refactoring	Partial	Cross-module/cross-language	Graph consistency + hidden tests.
MUL Source as a language	Subset	Full library and concurrency	Conformance suite.
MUL → Go	Yes	Stable full backend	Differential tests.
MUL → C++	No	Backend	Toolchain matrix.
MUL → Rust	No	Backend with ownership	Semantics/borrow checker.
MUL → native	Prototype after Go	LLVM / various CPUs	Binary/ABI/benchmark.

Table 13.1: Capabilities and limitations map

Capability	MVP	Full vision	Verification
Local GGUF model	Proof of concept	Fine-tuned MUL specialist	Held-out evaluation.
Full translation of arbitrary C++	No	Not to be guaranteed	Language semantics mismatch.

13.1 Extension: PHP, Java, Ruby, TypeScript, and JavaScript

The scope of the MUL project eventually also covers PHP, Java, Ruby, TypeScript, and JavaScript. None of these languages is supported automatically merely because a shared IR exists: a frontend, semantics recognition, source mapping, a safe editing mechanism, and, independently, a generation backend are all required. The matrix below is a product plan, not a description of current features.

Table 13.2: Extension: PHP, Java, Ruby, TypeScript, and JavaScript

Language	Frontend / import	Editing existing files	Generating MUL → language	Specific risk
PHP	AST/CST + name and autoloading analysis	Source-faithful patches; PHP validation	PHP backend + formatting	Dynamic include, magic methods, PHPDoc, Composer
Java	AST + symbol solver + classpath	Type-aware refactorings	Java backend + javac	Generics/erasure, overload, reflection, JVM

Table 13.2: Extension: PHP, Java, Ruby, TypeScript, and JavaScript

Language	Frontend / import	Editing existing files	Generating MUL → language	Specific risk
Ruby	CST + static and dynamic information	Careful patches and exceptions	Ruby backend + ruby -c	method_missing, reopened classes, eval, metaprogramming
TypeScript	Compiler API + tsconfig + project references	Edits preserving types and emit	TS backend + tsc	Conditional types, decorators, type erasure, JSX
JavaScript	AST + dependency resolver	Patches with dynamic uncertainty	JS backend + runtime	Dynamic bindings, eval, ESM/CJS
Go	go/parser, go/types, and modules	High precision with static symbols	Go backend + go build	Generics, channels, concurrency
C++	Clang AST + compilation in the project context	Conservative around macros	C++ backend + clang++	Preprocessor, ABI, ownership, templates
Rust	Parser/HIR, cargo metadata	Borrow-checker awareness	Rust backend + cargo	Ownership, lifetimes, macros

13.2 MUL Language Compatibility Layer (LCL)

LCL is a versioned contract between the independent MUL IR core and the language dialects. LCL does not blur differences: it describes what is shared,

what requires an extension, and what the system cannot safely represent or emit.

- Common Core: values, types, calls, functions, modules, contracts, side effects, control flow, exceptions, and dependencies.
- Dialects: java.jvm (classes, interfaces, erasure), php.runtime (autoload, magic methods), ruby.dynamic (open classes, dynamic dispatch), ts.types (compile-time types, JSX), js.runtime (prototypes and modules).
- Capabilities: the backend manifest declares the supported constructs and a guarantee of exact / equivalent / approximate / unsupported. The approximate mode is disabled for automatic APPLY.
- Source provenance: every IR node has a mapping to a file, byte range, revision, symbol identifier, build configuration, and source language.
- Semantics preserving: a refactoring requires proof of preconditions, type validation, or configured tests. Missing proof means REVIEW_REQUIRED, never a silent success.

Example interface: 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. Operations must be idempotent where feasible, auditable, and reversible.

13.3 Parser contract: analysis is not translation

Table 13.3: Parser contract: analysis is not translation

Support level	Guarantees	Example tests
L0: Parse	Syntax tree, ranges, comments, parser errors	syntax fixtures, parser fuzzing
L1: Resolve	Symbols and references in a known environment	cross-file references, package imports

Table 13.3: Parser contract: analysis is not translation

Support level	Guarantees	Example tests
L2: Edit	Safe transformations and diff control	rename, version bump, import update
L3: Emit	Correct code from the common MUL profile	MUL → source → parse/typecheck/build
L4: Translate	Limited cross-language transfer with a semantics guarantee	equivalence properties, integration tests

For dynamic constructs, the L1/L2 result may be incomplete. For PHP and Ruby, correct behavior means explicitly flagging the uncertainty and leaving the affected source fragments unchanged. These languages must not have a declaration of "full reference analysis" without specifying the runtime environment and the scope of dynamic calls.

13.4 Semantics: dynamic and static languages in one IR

- Types: MUL Core stores both declared types and constraints/inferences; a dynamic type is not replaced by an artificial static type. TypeScript has types erased on JS emit; Java has type erasure and overload resolution.
- Exceptions and effects: a controlled model of throws, promise/async, IO, mutations, reflection, and external calls. TypeScript async/await semantics are not equivalent to Go goroutines or JVM threads.
- Modules: Composer/autoload in PHP, gem/Bundler in Ruby, Maven/Gradle in Java, and npm/tsconfig in TypeScript are part of the compilation context, not merely a list of imports.
- Macros and metaprogramming: untranslatable source fragments must retain their original form + opaque node + warning. Transformations on such nodes require a dedicated adapter or escalation to AI.
- ABI and runtime: JVM bytecode, the JS runtime, the PHP engine, the Ruby VM, and native LLVM are distinct targets. The native MUL compiler is not automatically a replacement for each of these execution environments.

13.5 Backend and compiler for new languages

For code written natively in MUL: the Java generator can produce .java sources and use javac; the target JVM backend can emit bytecode. The TypeScript generator produces .ts and validates through tsc, and deployment depends on compilation to JavaScript and its runtime. PHP and Ruby get source generators and validation by the respective interpreters. A custom LLVM backend concerns the native MUL profile, not direct execution of PHP, Ruby, or TypeScript as a whole.

Two requirements must be kept separate: (1) MUL → target: synthesis of new code within the guaranteed profile; (2) source → MUL → source: editing an existing application while preserving style, comments, macros, configuration, and semantics. The second requirement should be implemented through source-preserving patches, instead of rebuilding all files.

Table 13.4: Backend and compiler for new languages

Target	Minimal validation	Project validation
PHP	php -l	Composer install/test, PHPUnit or the project runner
Java	javac	Gradle/Maven tests, checkstyle/lint according to the project
Ruby	ruby -c	Bundler, RSpec/Minitest, RuboCop if used
TypeScript	tsc --noEmit	Test runner + ESLint and bundler build
JavaScript	Parser + test runner	Node runtime, bundler, ESM/CJS checks

13.6 AI task profiles and token minimization

A small MUL model or Claude does not need to receive the whole tree. The choice of operation is narrow, and the extensive IR, AST, graph, and files

remain on the engine side. For tasks involving dynamic layers, the results contain uncertainty_code and specific exceptions for review.

Table 13.5: AI task profiles and token minimization

Example task	MUL engine work	When AI is necessary
Version bump in Composer/package.json/pom.xml/gemspec	Recognizing manifests, project versions, safe edits, test	Unclear documentation versions and versions of independent modules
Symbol rename	Resolve references, patch, format, typecheck	Reflection, dynamic dispatch, names built at run time
Add field/parameter	Modifying structure, calls, types, and test	Public API change and semantic decisions
Move function	Updating imports and dependencies	Module boundaries, side effects, framework hooks
MUL → Java/TS/PHP/Ruby	Generator with a limited profile and validation	No equivalent for the operation, runtime dependencies

An example of a compact report is shown below. Its full structure is stored outside the model context.

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

Detailed patches remain local and are fetched on demand; token_budget limits only the response, not verification or the completeness of the search.

13.7 Tests, KPIs, and a realistic rollout order

Each language gets an independent quality gate. A test result of “MUL works on Java” does not follow from it working on Go. Public support declarations must name the L0–L4 level and the semantic profile.

Table 13.6: Tests, KPIs, and a realistic rollout order

Test	Method	Initial criterion (proposed, not measured)
Parse / print stability	Random project samples + round-trip without edits	No unintended source changes for L2
Rename correctness	Fixture with dynamic dispatch and shadowing	100% of positive tests, 0 implicit unsafe edits
Build after transform	Native toolchains of each language	Do not degrade build/test in supported tasks
Behavioral equivalence	Integration and property-based tests	No detected regressions in the declared profile
Token efficiency	Equal tasks: agent+grep vs agent+MUL	Research goal: 50% fewer tokens at comparable success
Full-task success	Requirements + build + test + diff review	Measure pass@1 and the number of escalations

Rollout phases: A: Go L2 and benchmark; B: TypeScript L2 + Java L1/L2; C: PHP L2 + Java L3; D: Ruby L1/L2 + TypeScript L3; E: PHP/Ruby L3 for MUL subsets; F: cross-language L4 transfer after conformance tests. Priorities should be changed based on demand and benchmarks, not on the promise of "every language out of the box".

13.8 Implementation decisions and the compatibility contract

- capabilities.json specification: each adapter declares the language name, version, L0–L4 level, dialects, resolve reliability, emit support, verification command, and limitations.
- Versioning: the MUL Core spec, dialect schema, and adapter API have

separate version numbers; import/transformation is refused when the contract is incompatible.

- Execution gate: parser → resolve → plan → preconditions → apply atomic patch → format/lint/typecheck → build/tests → report; rollback on failure according to the project policy.
- Security: when running npm/composer/bundle/gradle/maven, use a sandbox and explicit permissions. External install scripts and build hooks can execute arbitrary code.
- Open extensibility: a third-party adapter SDK; a conformance test suite; a plugin must not declare full support without passing the public tests.

Conclusion of the extension: PHP, Java, Ruby, TypeScript, and JavaScript are full-fledged targets of the MUL architecture. Each has a separate frontend, semantic dialect, and possibly a backend. Support for analysis, transformation, and generation is measured independently, so the AI language does not reduce all runtimes to a fictitiously identical machine.

13.9 Language adapter as a separate module

Each source language is served by a separate adapter: an independent process, written in any language and communicating with the core via JSON on standard input and output. The core does not know any language by name; the Go frontend is the first adapter on this contract, not a special case. This makes it possible to use the analysis tools from the given language's ecosystem, to isolate adapter failures and resource limits, and to add a language without changes in the core. An adapter declares in `capabilities.json` its L0–L4 level, dialects, and verification command, and its declaration is confirmed by a shared conformance suite. Candidates for read-only adapters (L0/L1) also include languages with a large base of legacy code, e.g., Fortran and COBOL; their preprocessor mechanisms and `COPY`/`REPLACE` are treated as macros (opaque nodes).

13.10 Object orientation in foreign languages

Classes, inheritance, traits, method overriding, and magic methods are represented in the Code Graph as dialect nodes and edges (`extends`, `implements`, `overrides`), referring to the original byte ranges, rather than being translated into MUL Source. Calls resolved at run time (dynamic dispatch, callbacks passed as a name in a string, e.g., the hook system in PHP applications, `call_user_func`, `include` with a dynamically computed path) are edges with `uncertainty_code` and require review before APPLY. A large PHP project with an extensive hook system is planned as a test corpus for the PHP adapter; edits go to the original PHP files.

14 Implementation roadmap

Table 14.1: Implementation roadmap

Phase	Scope	Artifact / gate
0 — Spec & research	Formal semantics, IR definitions, capability matrix, tests	MUL RFC 001–010 + 30 golden cases.
1 — Core	MUL parser, types, serializer, verifier, symbol IDs	CLI check/inspect; fuzz tests.
2 — Go frontend/patch	Go parser and resolver, source maps, 4 transformations	Safe Go round-trip; benchmark A/B/C.
3 — Go backend	Emission of MUL programs to Go, basic runtime	End-to-end build/test.
4 — AI API + MDDB	Plan/apply/verify, token budgets, adapter	Automated repository tasks.
5 — Local model	Dataset, LoRA, GGUF, routing	Benchmark D and cost report.
6 — LLVM	Lowering and native compilation	Differential test Go vs native.
7 — Multi-language	C++ frontend and backend; Rust later	Defined coverage and limitations.

14.1 Minimal team and roles

Core: compiler/semantics lead, Go frontend engineer, transformation and API engineer, test/benchmark engineer; AI/ML joins once a good corpus exists. A security audit is also needed before running foreign code. Timelines depend on scope and team; the full MUL should not be promised within a few sprints.

14.2 Backlog for the first 12 weeks (indicative)

1. Weeks 1–2: normative definitions of operations, types, and source maps; entry benchmark grep vs MDDB.
2. Weeks 3–4: IR/verifier/serialization and a small MUL Source implementation.
3. Weeks 5–7: Go ingest, symbols, and the first safe edits of version/literal/rename.
4. Weeks 8–9: plan/apply/verify, snapshot, rollback, correct CLI report.
5. Weeks 10–11: 100-task benchmark; failure analysis, fixes.
6. Week 12: go/no-go decision on further work on the Go backend and the local model.

This is a proposed plan, not a promise of completion. The full language and the native compiler are separate, later stages.

15 Product success criteria and the go/no-go decision

15.1 Conditions for continuing the project

- MUL performs mechanical tasks at least as reliably as the baseline, with measurably lower token consumption.
- There is no class of unsupported transformations that MUL performs silently while changing the meaning of the program.
- The system reproduces existing source code within the declared scope and yields an auditable diff.
- The tool interface is useful even without a specially tuned model.
- A dedicated compiler has real value: portability, deterministic operations, runtime control, potential optimization; it is not built merely for prestige.

15.2 Conditions for stopping / changing direction

If the results show similar token consumption or a high error count, first simplify the product to a deterministic engine of operations on the AST/Code Graph. If the full MUL Source requires a disproportionate amount of work, keep only MUL IR and the transformation protocol. If the local model does not improve the cost per correct task, leave it optional.

16 Reference examples

16.1 Function refactoring without rewriting sources

```
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 New MUL code emitted to Go

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

The example illustrates a semantically compatible case. For full compatibility the language must define, among other things, integer overflow and the string representation.

16.3 Unsupported translation

```
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 Token report

```
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
```

As a helper, the command `mul tokens <file> --compare <file>...` estimates the token count of a MUL program and of a program with the same behavior in another language, and reports the difference. The result depends on the tokenizer, so the tokenizer is swappable, the default estimator runs locally, and code is not sent externally without explicit configuration. The same counter is used in the benchmark harness; comparing source size does not replace measuring tokens per correctly completed task.

17 Design decisions pending approval

Table 17.1: Design decisions pending approval

ID	Decision	Recommendation
ADR-001	MUL separated from MDDB	Yes; explicit adapter.
ADR-002	Canonical structure	Typed IR + source maps; embeddings optional.
ADR-003	First language	Go; patch mode first.
ADR-004	IR framework	Custom IR in Rust; MLIR only if measurement shows a need (accepted conditionally).
ADR-005	Runtime and memory	Fix the v0.1 profile before the backends.
ADR-006	AI model	Optional; LoRA→GGUF after data is collected.
ADR-007	Native compiler	Superseded by ADR-011.
ADR-008	Key metric	External tokens per correctly completed task.
ADR-009	Security	Transactions, sandbox, audit, no silent fallback.
ADR-010	Scope of “universality”	Capability profiles; no any→any promises.
ADR-011	Compiler implementation	Rust as the core; Cranelift first, LLVM later.

18 Glossary

Table 18.1: Glossary

Term	Definition
IR	Intermediate Representation: a formal, typed model of a program.
CST/AST	Concrete / abstract syntax tree.
Source map	Mapping of IR elements to source positions and back.
Code Graph	Graph of symbols, types, definitions, uses, and dependencies.
Embedding	A vector approximating semantic similarity; not a reversible code.
Dialect	A set of operations with separate semantics or target support.
Lowering	Transformation from a higher IR to a lower one or to the target language.
GGUF	A model distribution format for inference, used in, among others, llama.cpp.
Snapshot	A consistent state of sources and indexes to which a change plan refers.
Differential testing	Comparing the behavior of equivalent implementations on the same inputs.

19 Technical sources and inspirations

The sources below describe existing technologies and methodologies. They are not evidence that a hypothetical MUL will achieve the stated benefits.

1. MLIR Language Reference — https://mlir.llvm.org/docs/LangRef/ (operations, dialects, types, regions, and the verifier).

2. MLIR Tutorials — https://mlir.llvm.org/docs/Tutorials/Toy/Ch-2/ (emitting a custom IR and extending the language).

3. LLVM: My First Language Frontend — https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/ (the path to building a compiler).

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

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

6. SWE-bench (ICLR 2024) — https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html (evaluation methodology for code repair).

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

20 Executive summary

MUL should be built as a separate programming platform for AI: a formal source language, a precise IR, a Code Graph, lossless source maps for edit mode, a transformation engine, Go/C++/Rust backends, and a dedicated compiler based first on LLVM. A GGUF model may be added as a specialized local planner, but it is not the foundation of correctness. The value of the first version must be proven on repository tasks and real token costs. Without that, MUL risks becoming an expensive abstraction that rewrites Go into its own language only to rewrite it back into Go.

Recommended first product: MUL Core + Go Frontend + plan/apply/verify + benchmarks. Second: MUL Source $\rightarrow$ Go. Third: LLVM native. Fourth: local models and further languages. Each stage should have independent criteria for correctness, savings, and the ability to roll back.

21 Technology decision: implementation of the MUL compiler

Status: recommended implementation architecture (November 23, 2026). This chapter supplements Chapters 3, 8, 14, and 17. The decision concerns the implementation language of MUL; it does not limit the set of languages supported by MUL.

21.1 Choice: Rust for the compiler core

Recommendation: write the core of the MUL Compiler in Rust. This covers the MUL Source parser, semantic analysis, the type system, MUL IR, the IR verifier, the symbol graph, change management, the optimizer, and backend coordination. Rust allows precise memory control and safe concurrency with a complex data structure; it does not, however, remove the need for tests and security reviews.

Table 21.1: Choice: Rust for the compiler core

Criterion	Rust	Go	C / assembler
Memory safety	Strong type system, ownership	GC and a simple API	Manual control, higher risk of errors
IR / graphs / transformations	Very good choice	Good, simple to maintain	Possible, higher maintenance cost
Native backend	Mature libraries and FFI	Possible via integrations	High control, high implementation cost
Practical use	Main MUL core	Network services / MDDB	Low-level fragments, ABI

Assembly is not recommended as the implementation language of the whole compiler. Writing the parser, type system, and semantic graph in assembly raises costs without a guaranteed speedup. C remains an option for very narrow integration layers; Go is suitable for services, the CLI, and infrastructure tooling.

21.2 Backend strategy: Cranelift → LLVM

Prototype stage: Cranelift as the backend for quickly obtaining native programs/JIT for a limited subset of MUL, without building a CPU instruction generator from scratch. Target stage: LLVM as an optional, heavily optimizing backend for production releases. Both backends should consume the same verified MUL IR semantics through an explicit lowering contract. Cranelift and LLVM need not support identical features or platforms from the first release.

Pipeline: MUL Source → parser → type/effect analyzer → verified MUL IR → lowering → Cranelift IR or LLVM IR → machine code → linker → program. Parallel source backends: MUL IR → Go/C++/Rust/Java/PHP/Ruby/TypeScript generator, according to the compatibility profiles.

21.3 Components and module boundaries

Table 21.2: Components and module boundaries

Module	Implementation	Contract
mul-syntax	Rust	Tokens, CST/AST, errors located in the source
mul-sema	Rust	Symbols, types, effects, diagnostics
mul-ir	Rust	Typed IR, stable identifiers, versioning
mul-verifier	Rust	Verification of types, CFG, and effects before the backend

Table 21.2: Components and module boundaries

Module	Implementation	Contract
mul-interp	Rust	Reference interpreter; oracle for differential tests
mul-transform	Rust	Plan/Apply/Verify, transactions, reversible changes
mul-backend-cranelift	Rust	Lowering + emit/JIT; v0.x profile
mul-backend-llvm	Rust + LLVM API	Lowering + AOT emit; production profile
mul-emit-*	Rust / language adapter	Source generation, source maps, validation
mul-cli / mul-api	Rust; independent protocol	CLI, JSON-RPC/gRPC; MDDB integration

21.4 MVP scope and acceptance criteria

- MVP version: local variables, i32/bool/string values, functions, calls, conditions, simple loops; explicitly defined semantics and diagnostics.
- First end-to-end test: MUL code → verified IR → Cranelift → runnable program; comparison of results with the reference interpreter or semantic tests.
- Backend test: an identical set of cases for Cranelift and LLVM, comparison of results, detection of lowering errors and ABI mismatches.
- Integration trial: import of a limited amount of Go code → MUL → controlled transformation → Go patch → gofmt/go test; the source remains the reference point for round-trip mode.
- AI measurement: input and output tokens, number of turns, share of passed tests, failed fixes, cost of escalation to a large model; comparison with Claude + grep and Claude + MDDB.
- Security: fuzzing of the parser/serialization, IR verification before the

backend, isolation of the compilation process and resource limits, API version control.

21.5 Longer-term outlook: own CPU backend and self-hosting

A custom x86-64/ARM64 generator without LLVM and Cranelift is not required for MUL to count as a compiler of its own: its own frontend, semantics, IR, and lowering already constitute a compiler of its own that uses an external backend. A standalone machine backend and compiling the MUL compiler written in MUL (self-hosting) are separate, long-term research goals, to be pursued after the language, runtime, ABI, and standard library have stabilized.

21.6 Risks and decision

Risks: the maintenance cost of two backends, semantic divergences, LLVM system dependencies, target limitations, and the need for a dedicated runtime. Mitigation: a backend trait with contract tests, a small common subset of the language, a compatibility matrix, and automated differential tests. Decision: Rust as the core; Cranelift first; LLVM later; assembly only where a benefit has been demonstrated. This order does not by itself promise lower AI token consumption: that is the result of a concise operation protocol and local execution of transformations.

Technical sources to verify during implementation: `https://cranelift.dev/`; `https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html`; `https://doc.rust-lang.org/book/`.

22 Implementation status (November 2026)

This chapter describes the state of the prototype at the time version 2.1 of the document was released. The detailed work plan is kept in the repository's `BACKLOG.md` file as successive waves, each ending in a gate.

22.1 Completed

- MVP-0 (RFC-001): functions without parameters returning `i32`, arithmetic with overflow and division-by-zero checks, full pipeline parser → sema → IR → verifier → interpreter.
- MVP language (wave 2): function parameters and recursive calls, `let`/`var` variables and block scopes, `bool`, comparisons, logical operators, `if`/`else`, `while`, `pure`/`io` effects, the `string` type and `print` (the Hello world program), the value of `main` as the process exit code, type checking in the IR verifier; each feature has its own RFC.
- Language reference for AI: formal grammar, `docs/language/`, `mul guide` and `llms.txt`.
- Cranelift backend: JIT execution, executable build, and differential tests of interpreter vs native code on the whole golden suite.
- Function parameters and calls with recursion bounded by a depth limit (RFC-002), `let`/`var` variables, assignment, and block scopes without name shadowing (RFC-005), and language documentation for AI (`docs/language/`, `mul guide`, `llms.txt`).
- The `bool` type, comparisons, short-circuit logical operators, and the `if`/`else` statement (RFC-006); IR with multiple basic blocks and block parameters instead of phi functions, with a verifier that checks SSA dominance.
- The `while` loop (RFC-007): the loop header block takes the variables assigned in the body as parameters, and the back edge passes their new values; the interpreter and Cranelift execute loops without changes to the calling convention.

- `pure` and `io` effects in signatures (RFC-009): no annotation means `pure`, and a `pure` function cannot call an `io` function (`MUL-T013`); an IR function carries its effect, and the verifier rejects such a call (`MUL-V011`).
- A quality gate run locally before every commit (`make check`: formatting, linter, tests, coverage of at least 98% per file, dependency audit); the project does not use external CI.

22.2 Deviations from the roadmap in Chapter 14

- Order: first the compiler on a minimal subset of the language (Section 21.4), then patch mode for Go and the benchmark with the go/no-go decision, and only after that emission to Go, the full v0.1 language, LLVM, the local model, and further languages.
- Native backend: Cranelift first (ADR-011), LLVM later.
- The language adapter contract is created together with the first frontend (Section 13.9), not after it.

22.3 Decisions added in version 2.1

- No inheritance in MUL Source; methods, interfaces, and composition (Section 4.5).
- Object model of foreign languages in the Code Graph, without translation to MUL (Section 13.10).
- Language reference for AI and the one-RFC-per-topic rule (Section 4.6).
- Token estimation with `mul tokens` (Section 16.4).
- A documentation site with an example gallery generated by `mul` (first example: Hello world) and a browser demo in which the interpreter, compiled to WebAssembly, runs code locally, without a server.
- Languages as modules: each adapter has its own directory, version, tests, and capabilities manifest; neither a fix to one adapter nor the addition of a new one requires changes in the core or in other adapters. A new adapter is created from a scaffold generator.

- Language evolution: an adapter declares the range of language versions it supports and outside that range works read-only or refuses; a new language release goes through a corpus check procedure (round-trip without changes, comparison of the share of opaque nodes); new constructs go first into a dialect, and into the Common Core only when the concept occurs in at least two languages; an unmaintained adapter is given the status of frozen at a specific version.
- Version tagging: the adapter reads the project's declared language version and maintains a table of constructs (introduced, deprecated, removed in a given version); each graph node has a minimum required language version, and symbols from dependencies carry the package version from the lock file. This makes it possible to plan language and dependency upgrades as operations (list of blockers, deterministic fixes, items for review) and to ensure the agent does not generate syntax newer than the project's version.
- Pascal (Object Pascal dialects) as a further adapter language.
- Python as a further target language of an adapter (level L2 with explicit uncertainty for dynamic constructs).
- Two experiments before the go/no-go decision: syntax density (keywords, abbreviations, symbols) measured in tokens per correctly completed task, and short session handles instead of full identifiers in the agent API.
- The repository language (code, messages, technical documentation) is English. This dissertation was written in Polish (`MUL_Rozprawa_Naukowa.tex`), which stays authoritative; this English edition keeps the same chapter and section numbering.

23 Pilot study: tokens of the source text

23.1 Aim and scope

This chapter reports the first measurement made on the working prototype. The research question is narrow: how many tokens does the same program take when written in MUL Source, Go, PHP and C, and what does that number depend on. The measurement concerns the size of text, not the work of an agent. It does not test hypothesis H1 (section 1.4), which is about tokens per correctly completed task and needs the experiment of chapter 11. The sample is small (8 programs, 32 files), so the results are a reconnaissance and a calibration of the method before a larger study.

The motivation is the author's earlier work on automated processing of curricula vitae[1]: on 100 synthetic documents, language models identified fields correctly in about 20% of cases when given text extracted from PDF, and in about 95% when the same data was structured (JSON-LD). That result belongs to a different domain and is not evidence for the MUL hypotheses; it only motivates the question whether an explicit program structure has a similar effect in work with code.

23.2 Material and method

Material. Eight programs (`collatz`, `digits`, `fib`, `fizzbuzz`, `gcd`, `hello`, `power`, `primes`) were written in four languages in `examples/compare/`. The versions of one program have the same function structure, the same names and the formatting style usual for the language. Equal behaviour is checked automatically: every version must print the same output and exit with the same code as the MUL program (`make compare`). The programs stay within the MVP language (section 21.4): `i32` numbers, booleans, strings, functions, recursion, conditions and loops.

Layouts of MUL. Every MUL program was counted in two layouts that hold the same tokens of the language (ADR-013): the pretty layout for people

[1] R. Rabczuk, *The Resume Parsing Crisis of 2025: How Applicant Tracking Systems Fail to Identify Qualified Candidates*, independent research, 20 October 2025, SSRN no. 6061595.

(`mul fmt --pretty`: one statement per line, four-space indentation, spaces around operators), on which the main tables are based, and the compact layout, the canonical form in which the files are stored and which agents read and write (a space only between two word tokens, no indentation and no line feeds). Go, PHP and C are counted as written, in the formatting usual for each. Section 23.5 compares them with compact MUL. Listings in this document keep the pretty layout.

Tokenizers. Each file was counted in five ways: (1) the estimator built into `mul tokens`; (2) and (3) the BPE vocabularies `cl100k_base` and `o200k_base` published by OpenAI, run locally; (4) and (5) the tokenizers of two locally run models, `qwen2.5-coder:7b` and `gemma3:12b`, read as the prompt length reported by the local runtime. The tokenizer of the Claude models was not measured because it cannot be run locally; that measurement belongs to the larger study. The primary tokenizer in the detailed tables is o200k_base. No file left the machine on which the measurement was made.

Measures. Let $T_L(i)$ be the number of tokens of program i in language L. For every language–MUL pair we computed the per-program ratio, the aggregate ratio and the geometric mean of the ratios:

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

A value above 1 means the text in language L is longer than in MUL. The 95% confidence interval of $\bar{r}_g$ was obtained in two ways: from Student's t distribution of the log-ratios,

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

and by the bootstrap (percentiles, 10 000 resamples with replacement, fixed generator seed). The differences $d_i = T_L(i) - T_{\mathrm{MUL}}(i)$ were tested with the exact Wilcoxon signed-rank test; the distribution of W^+ was obtained by enumerating all 2^n sign assignments, so the smallest attainable two-sided p is $2/2^8 = 0.0078$. Effect size is given as $d_z = \bar{d}/s_d$. The relation between languages was approximated by ordinary least squares,

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

with the coefficient of determination R^2. All computations are done by the script `research/token-sample/measure.py`; its statistical functions have a self-

test on cases with a known result.

23.3 Results

Table 23.1: Token counts of the comparison programs (tokenizer o200k_base); lines and bytes refer to the MUL version in the pretty layout, the column “compact” to the same program in the compact layout

Program	Lines	Bytes	MUL	compact	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
Total	161	2736	909	648	865	889	987

Table 23.1 holds the raw data. In total the MUL programs take 909 tokens in the pretty layout (648 in the compact one), Go 865, PHP 889 and C 987. Table 23.2 gives the ratios for all tokenizers and figure 23.1 shows them graphically.

With the primary tokenizer, the Go text takes on average $\bar{r}_g = 0.922$ of the MUL text (95% CI: 0.840–1.012), PHP 0.912 (0.765–1.086) and C 1.055 (0.980–1.137). In this sample the MUL text is therefore not shorter than Go or PHP; it is a few per cent shorter than C. All three t intervals include 1. The Wilcoxon test gives $p = 0.0078$ for Go, because the Go version is shorter in all eight programs ($W^+ = 0$), but the difference is small: -5.5 tokens per program on average. The two methods disagree because of the `hello` program, where the fixed overhead of a file dominates its content and inflates the variance of the log-ratios.

The direction of the difference against Go depends on the tokenizer: for `gemma3:12b` $\bar{r}_g = 1.036$, that is, Go is longer than MUL. The outcome of comparing the syntax of two languages is thus a property of the language–tokenizer pair, not of the languages alone. Fifteen comparisons were made without correction for multiple testing; with the Bonferroni correction the threshold $0.05/15 = 0.0033$ is below the smallest attainable p, so none of the differences

Table 23.2: Token ratio language/MUL: aggregate ratio ρ, geometric mean $\bar{r}_g$, 95% confidence intervals and the Wilcoxon test ($n = 8$)

Tokenizer	Language	ρ	$\bar{r}_g$	95% CI (t)	95% CI (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

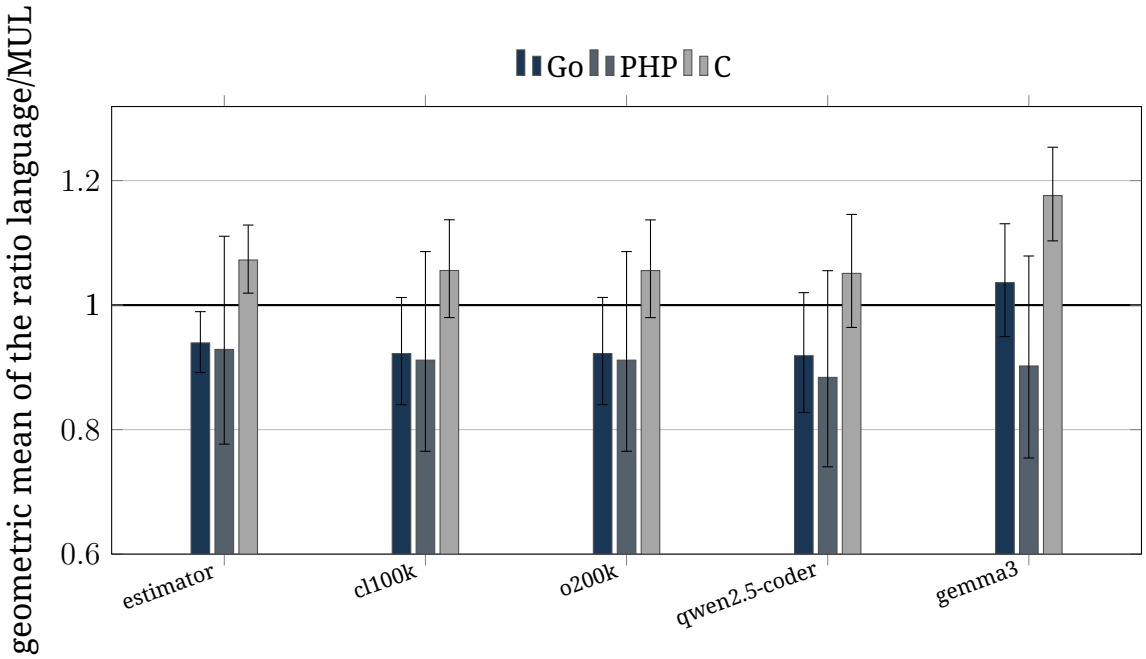

Figure 23.1: Geometric mean of the token ratio relative to MUL with the 95% confidence interval (t distribution). The horizontal line at 1 marks equality with MUL.

should be read as statistically confirmed.

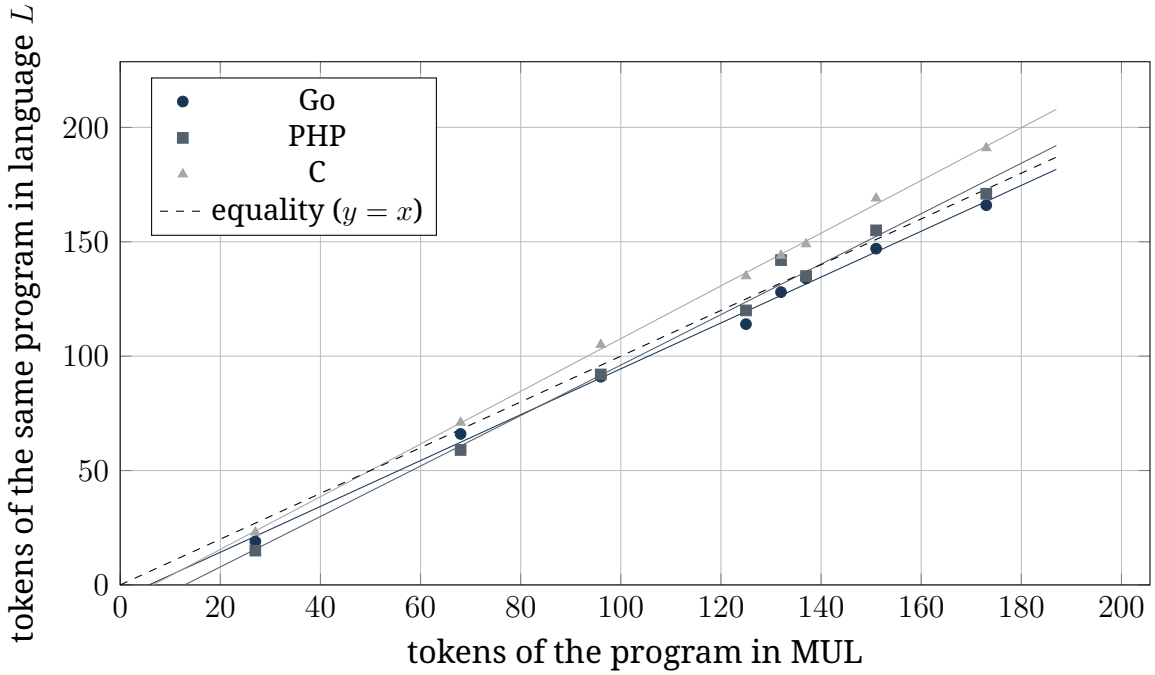

Figure 23.2: Token count of a program in Go, PHP and C against MUL (tokenizer o200k_base) with regression lines.

Table 23.3: Linear approximation $T_L \approx \alpha + \beta\, T_{\text{MUL}}$ and paired differences (tokenizer o200k_base)

Language	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

Within the measured range the relation is close to linear (figure 23.2, table 23.3; R^2 from 0.991 to 0.999). For Go the slope $\beta = 1.002$ with a negative intercept $\alpha = -5.8$ means that both languages grow at the same rate and MUL carries a fixed overhead of a few tokens per file (the module declaration, the signature of `main` with its result type and effect). For PHP and C the slope is above 1 (1.103 and 1.152) and the intercept negative: short programs are shorter in those languages than in MUL and longer ones become longer. The crossover $T^* = -\alpha/(\beta - 1)$ falls near 138 tokens for PHP and 50 tokens for C. Extrapolation beyond the sample range (27–173 tokens) is not justified; the approximation serves to plan the larger study, not to forecast.

23.4 Control variable: indentation

The languages differ in indentation convention (a tab in Go, four spaces in the others), which could explain part of the differences. The measurement was repeated on files stripped of leading indentation.

Table 23.4: Share of indentation in the token count and the aggregate ratio ρ before and after removing it (tokenizer o200k_base)

Language	Indentation [%]	ρ	ρ without indentation	95% CI of $\bar{r}_g$ without indentation
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]

Indentation accounts for 8.3% (Go) to 10.6% (MUL) of the tokens. Without it the ratio for Go moves from 0.952 to 0.975 and for C from 1.086 to 1.096 (table 23.4). The indentation convention thus accounts for about half of the difference between MUL and Go; the order of the languages does not change.

23.5 Second layout: compact MUL

The comparisons above use pretty MUL, which is how people read the language. Agents read and write the compact layout (ADR-013), so the measurement was repeated with compact MUL as the denominator; the texts in Go, PHP and C are the same files as before.

The compact layout takes 28.7% fewer tokens than the pretty one with the primary tokenizer (from 25.3% to 29.0% across tokenizers). Against compact MUL, Go takes $\bar{r}_g = 1.273$ (95% CI: 1.110–1.460), PHP 1.258 (1.014–1.561) and C 1.457 (1.289–1.647) (table 23.5, figure 23.3). The direction no longer depends on the tokenizer: every language is longer than compact MUL with every tokenizer, and only the PHP interval of `qwen2.5-coder:7b` includes 1.

The comparison is not symmetric. Compact MUL drops all optional whitespace, while the other languages keep the formatting usual for them; indentation alone is 8.3–9.7% of their tokens (table 23.4), and Go, PHP and C stripped in the same way would also shrink. The result therefore measures the layout decision of ADR-013, the form in which agents exchange MUL, not an advantage of MUL syntax. Whether a model writes and edits code in the compact layout as correctly as in the pretty one was not measured; that is backlog row

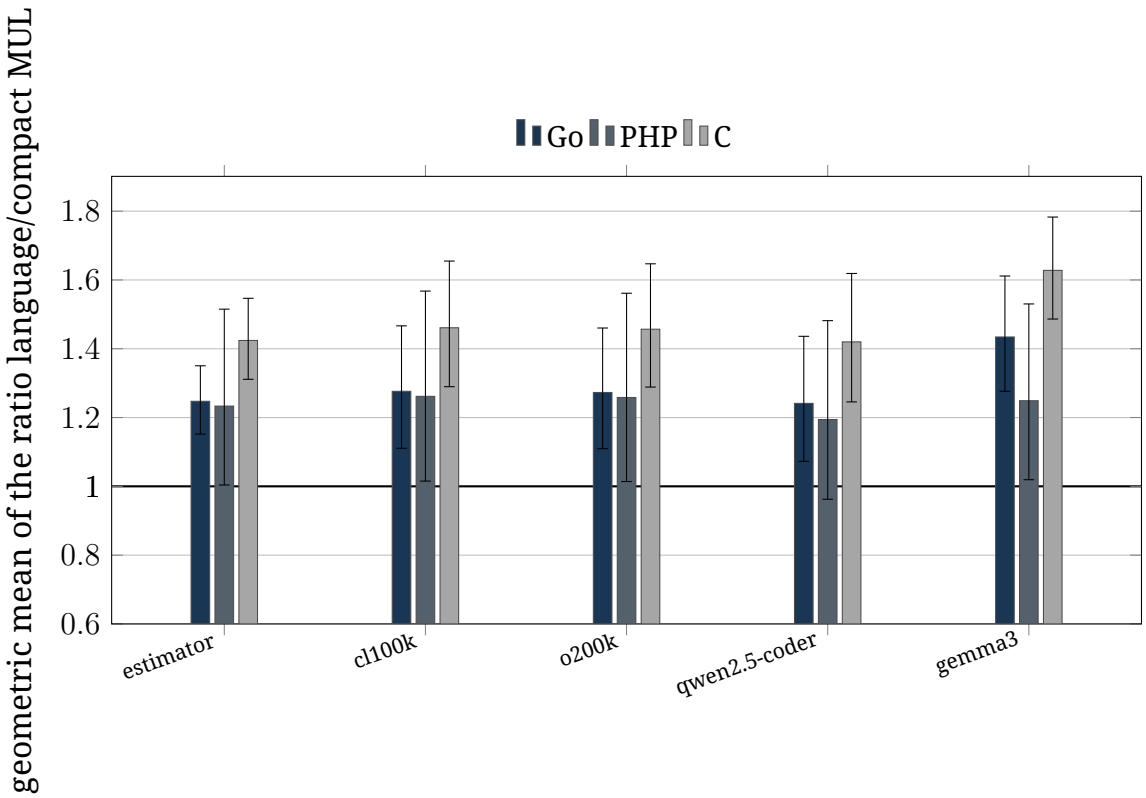

Figure 23.3: Geometric mean of the token ratio relative to compact MUL with the 95% confidence interval (t distribution). The horizontal line at 1 marks equality with MUL; compare figure 23.1.

Table 23.5: Token ratio language/MUL with MUL in the compact layout: aggregate ratio ρ against pretty and against compact MUL, geometric mean $\bar{r}_g$ and its 95% confidence interval (t distribution, $n = 8$)

Tokenizer	Language	ρ pretty	ρ compact	$\bar{r}_g$ compact	95% CI (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]

7.3, and ADR-013 stays conditional until then.

23.6 Cost of the language reference and the break-even point

A model that does not know MUL must be given a description of the language. The compact reference (`mul guide`) takes 2172 tokens. If code in language L is ρ times longer than in MUL, the reference cost C_{ref} is recovered after the model has read a code volume of

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

measured in tokens of the text in language L. For $\rho \leq 1$ there is no break-even point.

The reference is itself written for agents: its examples are in the compact layout, and since the formatter was added it has grown to 2172 tokens, which moved every V^* up. With pretty MUL, the reference cost is recovered against C after about 27 500 tokens of code, that is, after about 223 programs the size

Table 23.6: Cost of the language reference C_{ref} and the code volume V^* (in tokens of the text in language L) after which it is recovered, with MUL in the pretty and in the compact layout; "–" means no break-even point

		pretty MUL			compact MUL		
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

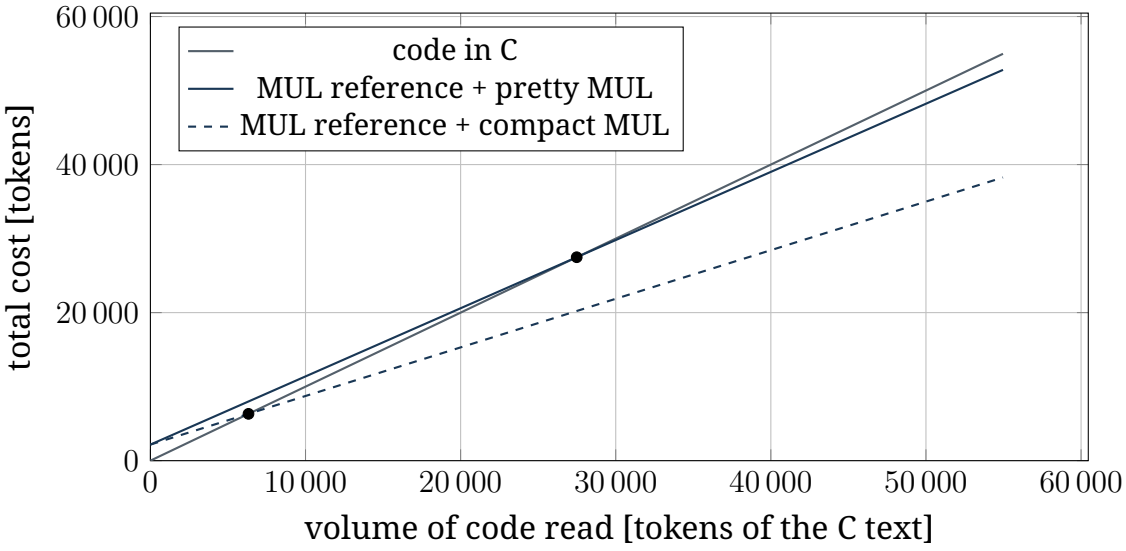

Figure 23.4: Total cost of reading code in C and in MUL with the one-off cost of the language reference (tokenizer o200k_base), for pretty and compact MUL. The dots mark V^*.

of the mean program in the sample; against Go and PHP, with the primary tokenizer, it is never recovered. With compact MUL it is recovered against all three languages: after about 8 700 tokens of Go (80 programs), 8 000 of PHP (72) and 6 300 of C (51; table 23.6, figure 23.4). The model ignores prompt caching and the fact that a model trained on MUL needs no reference. As things stand, MUL syntax alone is not a source of token savings; the layout in which agents exchange the code is, with the reservation of section 23.5.

23.7 Textual IR against the source

Table 23.7: Token count of a MUL program and of its textual IR (`mul ir`; tokenizer o200k_base)

Program	MUL source	IR	IR/source
`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

The textual form of the IR is on average 2.82 times longer than the source (geometric mean; range 1.81–3.74, table 23.7). This supports with numbers the assumption of section 7.1: a model should be given the results of operations and short identifiers, not raw IR.

23.8 Accuracy of the built-in estimator

The estimator works without a vocabulary. After the first round of this study it was recalibrated on `results.csv`: it counts runs of punctuation per three characters and parts of words per five, a line break directly after punctuation costs nothing, leading indentation is one token of its own, and one of the characters `( . : # {` directly before a word merges into it, as in `(n` or `{return`, which is frequent in compact MUL. The first version overstated the count relative to o200k_base by 11.8% on average, with a mean absolute error of 13.3%, and rated Go as longer than MUL ($\rho = 1.115$) while the BPE tokenizers

Table 23.8: Error of the `mul tokens` estimator against reference tokenizers on 32 source files

Reference tokenizer	Mean absolute error [%]	Bias [%]	Largest error [%]
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

rated it as shorter. After the recalibration the bias is -1.6% and the mean absolute error 4.0% (table 23.8). The estimator now orders the languages as the BPE tokenizers do: Go $\rho = 0.953$ against 0.952, PHP 0.998 against 0.978, C 1.091 against 1.086. On the eight programs in the compact layout the mean absolute error is 2.5% (bias $+0.9$%, largest error 4.9%), and the estimator gives the compact layout a saving of 25.3% against 28.7% with o200k_base.

Three limits remain. The largest error, 15.8%, falls on the shortest files (`hello` in C and Go, about 20 tokens), where one token is several per cent. The estimator follows the OpenAI vocabularies and the model `qwen2.5-coder:7b`; it undercounts `gemma3:12b` by 21.8%, because that tokenizer splits the same text into more pieces. And the rules were fitted on the same files they are tested on (the 32 files of table 23.8 and the eight compact MUL files), so the errors above are in-sample errors. The estimator is suitable for comparing languages on the BPE vocabularies it was fitted to; for another model the comparison should use that model's tokenizer (option `--tokenizer-cmd`), and `mul tokens` marks every estimated number as an estimate.

23.9 Threats to validity

- **Sample size and selection.** Eight short programs written by the author of the project; no random selection and no code from real repositories. The programs are limited to constructs MUL already supports, which leaves out structs, collections, error handling and libraries, the areas where languages differ most. Rust versions of the programs were added to the repository after the measurement; they enter the next round.
- **Writing style.** The versions were written in parallel with the same names and structure. Idiomatic code in each language could give different ratios (for example a counted `for` in Go and C, which MUL does not have).

- **Layout.** Compact MUL is compared with the other languages as written, not with their minified forms, so the ratios of section 23.5 mix the language with the formatting convention. The correctness of code that a model writes in the compact layout was not measured.
- **Tokenizers.** `cl100k_base` and `o200k_base` give nearly identical results on this sample, so they are not independent observations. The numbers for local models come from the prompt length reported by the runtime, less the overhead measured on a one-character prompt. The tokenizer of the Claude models was not measured.
- **Estimator calibration.** The rules of the built-in estimator were fitted to this sample, so its error here is an in-sample error. The larger study measures it on files that took no part in the fitting.
- **Scope of the conclusions.** The size of text was measured. The correctness of code written by a model, the number of turns, the cost of repairs and the cost of reading context, which decide hypothesis H1, were not.
- **Independence of the researcher.** The author of the measurement is the author of the language under study. The data, the script and the programs are in the repository so that the measurement can be repeated.

23.10 Conclusions and the plan for larger samples

1. In the sample studied, MUL Source is not shorter in tokens than Go or PHP and is slightly shorter than C; the differences are a few per cent and their direction for Go depends on the tokenizer.
2. The cost of the language reference (about 2 thousand tokens) is not recovered by concise syntax alone.
3. The compact layout of MUL takes 28.7% fewer tokens than the pretty one; against it Go, PHP and C are 26–46% longer (geometric mean, tokenizer o200k_base) and the reference is recovered after several thousand tokens of code. This is a gain of the layout, compared with the other languages as written, and it counts only if the correctness of the code does not drop (row 7.3).
4. The textual IR is close to three times longer than the source.

5. After the recalibration the built-in estimator is within a few per cent of the OpenAI BPE vocabularies and orders the languages as they do; for other models the comparison needs the model's own tokenizer.

The results agree with the assumption made in chapter 4 that conciseness for AI is to come from operations on the IR, not from syntax, and they give no argument for shortening keywords. They say nothing about hypothesis H1. Next steps: (1) a sample of at least 100 programs, including fragments of real repositories and constructs of the full v0.1 language; (2) the tokenizers of all models used in the experiment, including the Claude models; (3) the syntax-density experiment with a measurement of the correctness of code written by a model; (4) a measurement of tokens per correctly completed task in variants A, B and C of chapter 11, with decision thresholds fixed before the measurement.

Concluding remarks

The MUL solution has a research justification as an attempt to shift the work of agents from interpreting the text of code to precise operations on the program. The critical condition for success is not the vector representation itself, but a verifiable IR, semantic adapter contracts and effective validation of changes. The first stage should build a Go–MUL–Go prototype and experimentally compare success rate, token usage and regression risk against standard agent work. Only after positive results will it be justified to extend the set of languages, train a specialized model and invest in a native backend.

Declarations

Authorship

The author of this document, of the MUL project and of the study in chapter 23 is Rafal Rabczuk, independent researcher (United Kingdom). The author is responsible for the content, the design decisions and the conclusions.

Use of AI tools

This work was prepared with the assistance of artificial intelligence tools. Several language models were used in design, in writing the prototype code, in editing the text and in review: locally run models, OpenAI models and Anthropic's Claude models. The models are not authors. The author set the goals and scope, made the decisions, checked the results and accepted or rejected the models' proposals. The numbers in chapter 23 come from deterministic computations of a script on files in the repository, not from model responses; local models served in the measurement only as a source of a tokenizer.

Data availability

The comparison programs (`examples/compare/`), the measurement script, the raw data and the intermediate results (`research/token-sample/`) are in the project repository. The measurement can be repeated with `make token-study`; the tables and figures of chapter 23 are generated from that data.

Conflict of interest

The author is the creator of MUL and develops MDDB, which appears in the experiment design (chapter 11) as one of the comparison variants.