

# Glagol: A Manifest-First Compiler Architecture for Slovo

ꞑꞑꞑꞑꞑꞑꞑꞑ

Sanjin Gumbarevic

[hermeticum\\_lab@protonmail.com](mailto:hermeticum_lab@protonmail.com)

Publication release: 1.0.0-beta.2

Technical behavior baseline: compiler and language support through 1.0.0-beta ; tooling and install workflow through 1.0.0-beta.1 ; runtime/resource foundation through 1.0.0-beta.2

Date: 2026-05-22

Evidence source: paired local Slovo/Glagol monorepo verification and benchmark reruns from a local checkout; beta.2 release-gate verification from the public monorepo

Maturity: beta

## Abstract

Glagol (ꞑꞑꞑꞑꞑꞑꞑꞑ) is the first compiler for Slovo. It exists to make the language support boundary inspectable: tokens, S-expression tree, AST, typed AST, LLVM IR, hosted native executable, tests, diagnostics, and release documents should agree.

The current publication release, 1.0.0-beta.2 , keeps the first real general-purpose beta toolchain baseline from 1.0.0-beta and records the first post-beta tooling/install hardening update plus the first runtime/resource foundation update. The beta baseline includes the completed u32 / u64 unsigned compiler and stdlib breadth scope alongside the current nine-kernel benchmark suite. This paper records the current beta implementation surface, the benchmark method and results, the distinction between Glagol and Lisp-family implementations, the beta.1 tooling update, the beta.2 runtime/resource foundation, and the compiler path from beta to stable.

## 1. Compiler Thesis

The name Glagol is rendered in Glagolitic as ꞑꞑꞑꞑꞑꞑꞑ. The publication pipeline embeds a Glagolitic-capable font so this identity marker survives PDF rendering.

Glagol's compiler motto is:

```
make the tree visible
```

The current pipeline is:

```
.slo source
-> tokens
-> S-expression tree
-> AST
-> typed AST
-> LLVM IR text
-> Clang + runtime/runtime.c
-> native executable
```

The engineering point is not only native output. It is traceability. Source structure, types, spans, diagnostics, formatter behavior, and generated code should stay connected enough that a support claim can be audited.

## 2. Relationship To Lisp Implementations

Glagol compiles a Lisp-shaped language, but it is not a Lisp implementation in the usual technical sense.

Common Lisp and Scheme implementations typically center a runtime evaluation model, symbolic data, macro expansion, and language-defined execution semantics. Clojure centers hosted execution on the JVM, namespaces, immutable persistent data structures, dynamic vars, and runtime sequence abstractions.

Glagol instead centers:

- manifest-first language contracts
- explicit AST and typed AST stages before backend emission
- static checking before native code generation
- canonical formatting and structured diagnostics as release artifacts
- explicit `option` and `result` flow instead of exception-driven ordinary failure
- lexical `unsafe` as the reserved low-level boundary
- hosted native executables through LLVM IR and Clang
- release gates that separate supported, compatibility, formatter-only, and speculative examples

The parenthesized syntax is therefore a structural source format, not evidence that Glagol is a macro-first Lisp VM or a generic list runtime.

### 3. Current Implementation Surface

At the current technical behavior beta baseline, Glagol supports:

- `check` , `fmt` , `fmt --check` , `fmt --write` , `test` , `build` , and `doc`
- `run` for build-and-execute workflows, `clean` for generated build artifacts, and `new --template binary|library|workspace`
- JSON diagnostics, textual artifact manifests, and lowering inspection
- hosted native executable generation through emitted LLVM IR, host `clang -O2` , and `runtime/runtime.c`
- flat local module projects, explicit import/export lists, local packages, and workspace membership
- installed `share/slovo/std` discovery and ordered `SLOVO_STD_PATH` search
- direct scalar types `i32` , `i64` , `u32` , `u64` , finite `f64` , `bool` , immutable `string` , and internal `unit`
- functions, top-level tests, immutable locals, current mutable whole-value locals, `if` , and `while`
- current direct enum payload families, current known struct field families, concrete option/result families, fixed immutable arrays over direct scalars and `string` , and concrete runtime-owned vector families over `i32` , `i64` , `f64` , `bool` , and `string`
- compiler-known standard-runtime calls through the promoted catalog plus staged source-authored `std/*.slo` gates
- scalar C FFI imports
- benchmark scaffolds for Slovo, C, Rust, Python, Clojure, and Common Lisp/SBCL, with `cold-process` and `hot-loop` timing modes

The current release, `1.0.0-beta.2` , is a beta runtime/resource foundation update on the first release line that may honestly use beta maturity language for this toolchain.

### 4. Diagnostics And Support Discipline

Glagol's quality boundary is not "the parser accepted a form." The required support path is:

1. parse the source
2. lower to AST with spans
3. type-check names and value flow
4. reject unsupported forms before backend panic
5. emit LLVM only from checked representation
6. cover behavior or diagnostics with tests
7. update release docs and fixtures together

This matters because Slovo syntax is intentionally regular. A permissive parser can make unsupported forms look almost supported. Glagol therefore treats backend panics, invalid LLVM from user source, and stale docs that overclaim support as release-blocking defects.

## 5. Runtime And Standard Library Strategy

Glagol currently exposes two related but distinct library surfaces:

- compiler-known standard-runtime calls such as `std.io.print_i32` , `std.string.len` , selected parse/format/conversion calls, host IO, process/environment/file helpers, randomness, time, and stdin
- source-authored beta modules in `lib/std/*.slo` , loaded through explicit imports, installed std discovery, checkout discovery, or `SLOVO_STD_PATH`

This split is deliberate. It lets library design move forward without claiming that the final stable import, compatibility, or package story already exists. Source-authored modules are useful now because they exercise language design, fixtures, and examples. They are beta explicit-import APIs, but not yet a frozen stable `1.0` standard library.

## 6. Benchmark Method

The benchmark suite measures local-machine behavior only. It is a regression and comparison harness, not a public performance claim.

Environment:

Field	Value
Host	Linux 6.17.10-100.fc41.x86_64 x86_64 GNU/Linux
Glagol	glagol 1.0.0-beta benchmark baseline
Python	Python 3.13.9
C compiler	clang version 19.1.7 (Fedora 19.1.7-5.fc41)
Rust	rustc 1.77.2 (25ef9e3d8 2024-04-09)
Clojure	1.11.2
Common Lisp	SBCL 2.5.9-1.fc41

Build and runtime paths compared:

Implementation	Build/runtime path
Slovo	glagol build <benchmark> -> generated LLVM -> host clang -O2 linking runtime/runtime.c
C	clang -O2 -std=c11 on the local scaffold
Rust	rustc -C opt-level=3 -C debuginfo=0 on the local scaffold
Python	python3 running the local scaffold
Clojure	clojure running the local scaffold; timings include JVM and Clojure startup
Common Lisp	sbcl --script running the local scaffold; timings include SBCL startup

Timing semantics:

- The runner builds each implementation once before timing. The reported numbers are execution timings, not compile-time timings.
- `cold-process` launches a fresh process per sample with the base loop count. It measures process startup plus one benchmark run.
- `hot-loop` also launches a fresh process per sample, but with the amplified loop count `10000000` ; the reported normalized median divides the timed total by `10` to compare with the base `1000000` loop count.

## Benchmark kernels:

- `math-loop` : scalar arithmetic accumulation
- `branch-loop` : scalar branching and accumulation
- `parse-loop` : repeated decimal parsing with checksum validation
- `array-index-loop` : checked fixed-array indexing and scalar accumulation
- `string-eq-loop` : exact string content equality reduced to an `i32` checksum
- `array-struct-field-loop` : immutable struct-field access over a fixed `i32` array plus scalar accumulation
- `enum-struct-payload-loop` : repeated enum `match` payload extraction over an immutable struct payload carrying a fixed `i32` array
- `vec-i32-index-loop` : runtime-owned `i32` vector indexing and scalar accumulation
- `vec-string-eq-loop` : runtime-owned string vector indexing plus exact string equality reduced to an `i32` checksum

## Comparison boundaries:

- `math-loop` and `branch-loop` compare structurally similar loop bodies across all implementations.
- `parse-loop` keeps the same input text and checksum, but not the same parser implementation. Slovo uses `std.string.parse_i32_result`, C uses `strtol`, Rust uses `text.parse::<i32>()`, Python uses `int`, Clojure uses `Integer/parseInt`, and Common Lisp uses `parse-integer`.
- `array-index-loop` keeps the same eight-element integer corpus and `% 8` dynamic index-selection pattern across all implementations. It stays on immutable fixed-array indexing and scalar accumulation only.
- `string-eq-loop` keeps the same five-word ASCII corpus and runtime-supplied target string across all implementations. It measures exact content equality only. It does not compare regex engines, normalization, locale handling, or pointer identity.
- `array-struct-field-loop` keeps the same eight-element integer corpus and `% 8` dynamic index-selection pattern, but moves the array through one immutable struct field. It is a narrow benchmark for the promoted `exp-120` direct struct-field lane, not a broad claim about every struct layout.
- `enum-struct-payload-loop` keeps the same eight-element integer corpus inside an immutable struct payload, matches one enum value on every iteration, and indexes the bound struct field. It is a narrow benchmark for the promoted `exp-121` struct-payload enum lane, not a broad tagged-union or ADT claim.
- `vec-i32-index-loop` keeps the same eight-element integer corpus and `% 8` dynamic index-selection pattern as `array-index-loop`, but routes that access through the promoted runtime-owned (`vec i32`) lane instead of fixed arrays.
- `vec-string-eq-loop` keeps the same five-word ASCII corpus and runtime-supplied target as `string-eq-loop`, but routes selection through the promoted runtime-owned (`vec string`) lane instead of fixed arrays.
- Because Rust is timed at `opt-level=3` while Slovo and C are timed through `clang -O2`, the suite is a useful local regression/comparison harness, not a strict same-flags compiler shootout.

## Hot-loop commands:

```
python3 benchmarks/math-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/branch-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/parse-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/array-index-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/string-eq-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/array-struct-field-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/enum-struct-payload-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/vec-i32-index-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
```

```
compiler/target/debug/glagol
python3 benchmarks/vec-string-eq-loop/run.py --mode hot-loop --repeats 5 --warmups 1 --glagol
compiler/target/debug/glagol
```

Cold-process commands:

```
python3 benchmarks/math-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/branch-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/parse-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/array-index-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/string-eq-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/array-struct-field-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/enum-struct-payload-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/vec-i32-index-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
python3 benchmarks/vec-string-eq-loop/run.py --mode cold-process --repeats 3 --warmups 1 --glagol
compiler/target/debug/glagol
```

## 7. Benchmark Results

The benchmark rows below remain the full-suite 1.0.0-beta publication baseline. 1.0.0-beta.1 changes tooling and install workflow, and 1.0.0-beta.2 adds runtime/resource APIs; neither release claims changed benchmark performance.

The exp-123 publication baseline widened the paired same-machine result set from seven rows to nine by adding two owned-vector kernels:

- vec-i32-index-loop
- vec-string-eq-loop

Hot-loop normalized median time, in milliseconds per one million iterations:

Benchmark	Slovo	C	Rust	Python	Clojure	Common Lisp/SBCL
math-loop	1.121	1.121	1.138	111.092	241.220	1.753
branch-loop	2.014	2.012	2.032	114.016	241.469	4.624
parse-loop	6.456	16.233	7.169	134.465	264.669	20.108
array-index-loop	1.103	1.109	1.128	96.649	298.388	3.379
string-eq-loop	4.332	4.092	2.279	120.453	288.617	11.128
array-struct-field-loop	1.139	1.116	1.129	110.854	277.466	3.663
enum-struct-payload-loop	4.304	1.512	1.880	302.252	310.066	5.297
vec-i32-index-loop	1.328	1.103	1.131	111.153	272.914	2.231
vec-string-eq-loop	5.210	4.122	3.471	122.826	302.817	10.431

Cold-process median time, in milliseconds per benchmark run:

Benchmark	Slovo	C	Rust	Python	Clojure	Common Lisp/SBCL
math-loop	1.625	1.675	1.765	121.014	2808.812	9.435
branch-loop	2.563	2.517	2.682	130.790	2674.146	15.027
parse-loop	6.942	16.749	7.857	149.594	2835.421	27.750
array-index-loop	1.599	1.606	1.807	107.150	2812.157	17.589
string-eq-loop	4.826	4.756	2.938	135.748	2892.359	21.504
array-struct-field-loop	1.670	1.612	1.837	115.371	2823.026	13.411
enum-struct-payload-loop	4.934	2.000	1.783	291.850	2516.815	27.555
vec-i32-index-loop	3.047	2.851	1.776	112.950	2911.603	10.017
vec-string-eq-loop	5.427	4.575	4.081	134.914	2567.482	18.950

Compiler interpretation:

- The current hosted build path keeps Slovo essentially on the local native baseline for the scalar and fixed-array kernels. In hot-loop mode, `math-loop`, `branch-loop`, and `array-index-loop` all land very close to the C scaffold and within a narrow distance of the Rust scaffold.
- `parse-loop` is now more than a backend-loop benchmark. It compares end-to-end parser and runtime choices. On this machine, the current Slovo decimal parse path outperforms the C scaffold built around `strtol` and stays close to the Rust scaffold.
- `string-eq-loop` exposes a different boundary: exact content equality is clearly efficient enough for native-code use, but the current Slovo runtime path is still behind the Rust scaffold and slightly behind the C scaffold on this machine.
- `vec-i32-index-loop` shows the cost of routing the same integer corpus through the promoted owned-vector lane instead of fixed arrays. On this machine the Slovo lane remains practical native code, but it is visibly more expensive than the fixed-array kernel.
- `vec-string-eq-loop` shows the same tradeoff for owned string vectors. It stays in the same broad range as the fixed-array string kernel, but it is a more allocation- and indirection-heavy path than direct fixed-array access.
- `array-struct-field-loop` stays close to the direct fixed-array kernel. On this machine, routing the same `% 8` indexing pattern through one immutable struct field keeps Slovo, C, and Rust tightly grouped in hot-loop mode.
- `enum-struct-payload-loop` exposes a current composite-data boundary. The Slovo lane remains practical native code, but repeated struct-payload enum matching is still materially slower than the C and Rust scaffolds on this machine.
- Cold-process timings show native executable startup plus one benchmark run. They are not compile-time numbers and are more sensitive to launcher/runtime initialization effects than hot-loop mode.
- Clojure is dramatically slower in this process-per-run harness because each sample includes JVM and hosted runtime startup, and the benchmark bodies stay on high-level runtime paths. The effect is still strongest in the more allocation- and dispatch-heavy composite kernels.
- Common Lisp/SBCL remains much closer to native baselines than Clojure in the same harness. That is why both Lisp-family comparison points are useful.

## 8. Current Technical Risks

The main risks in beta are not syntax parsing. They are engineering coverage and compatibility:

- source forms reaching backend paths without clear diagnostics
- standard-library source helpers drifting from compiler-known runtime calls
- feature claims appearing in docs before fixtures and tests exist
- collection and ADT breadth growing faster than the compatibility story
- benchmark breadth growing faster than the language contract can stabilize
- benchmark numbers being misread as public thresholds or cross-machine claims
- package behavior becoming stable before dependency, manifest, and versioning rules are precise

## 9. Path Beyond 1.0.0-beta.2

Glagol now implements the first real beta Slovo contract, the first post-beta tooling/install hardening release, and the first runtime/resource foundation release. The remaining path is from beta to stable.

Recommended compiler sequence:

1. Complete the next blocked post-beta language-breadth slices from the Slovo roadmap without regressing the beta baseline.
2. Broaden runtime-owned strings, collections, and composite value flow without exposing unstable ABI details as stable contracts.
3. Refine `f32` policy, additional integer families, explicit conversion behavior, and remaining library/runtime gaps.
4. Harden package, workspace, and standard-library import/search behavior into a compatibility-governed stable toolchain story.
5. Strengthen diagnostics, generated docs, conformance fixtures, and release gates as first-class compiler interfaces.
6. Keep benchmark publication local and repeatable while deferring public performance claims until methodology is stronger.
7. Freeze formatter output, diagnostics schema, package behavior, stdlib compatibility, migration policy, and toolchain contracts for 1.0.0.

## 10. Conclusion

Glagol has moved Slovo from a manifesto into a working beta native compiler track. The important result is not only that programs compile. It is that the support boundary is visible enough to review: source contracts, diagnostics, tests, lowering, benchmarks, and publication artifacts can be kept in sync.

The compiler is now useful enough for ordinary local tools and libraries within the documented beta contract. The path forward remains disciplined breadth and compatibility hardening, not unsupported feature claims.