Review paper

Phases of compiler design

The six-phase model, measured against Zero[3] — a compiler that reports its own phase structure as machine-readable data.

Submitted by

Harshit Khemani

Co-authors

Kush Ahuja, Mohit Kumar Mishra, Kushagra Agrawal

Submitted to

Ms. Ankita Sharma

Read online

`zero.khe.money`

Source and dataset at
github.com/HKTITAN/phases-of-zero.

Abstract

Compiler construction is conventionally taught as a pipeline of six phases: lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and target code generation, with symbol-table management and error handling running alongside all six. This decomposition is a teaching abstraction. Production compilers rarely expose it, so students seldom see the phases as separable, measurable objects.

This paper reviews that model against Zero (https://zerolang.ai/)[3] 0.3.4, an experimental graph-first systems language from Vercel Labs whose compiler reports its own phase names, per-phase timings, symbol tables, and diagnostics as structured JSON. We construct a corpus of 8 Zero programs (680 lines, 3,411 tokens, 2,216 graph nodes) and a second corpus of 10 deliberately malformed programs, then measure the compiler across 64 program-target build combinations.

We report four findings. First, the phases do not disappear in a graph-first design — they *relocate*. Lexical, syntactic and semantic analysis move to an ingestion boundary that admits programs into a persistent graph, after which the compile path performs only lowering, code generation and linking. All 7 front-end error cases were refused at that boundary, and in none of them did the malformed program enter the graph store. Second, lowering accounts for effectively all measurable phase time (100%); the entire front end runs below the compiler's 1 ms reporting resolution because it reads stored facts instead of recomputing them. Third, the front end and the back end accept *different languages*: 17 of 64 builds failed with BLD004 on programs that had already passed `zero check`, and the compiler reports `ok: true` alongside `buildable: false` in the same document. Fourth, and against the language's own premise, its structured output is far more expensive to read than its prose — `zero check` prints four bytes where `zero check --json` returns tens of thousands.

8 Programs in corpus	**2,216** Graph nodes analysed
47/64 Builds that succeeded	**100%** Phase time in lowering
10 Error cases	

All figures produced by `tools/capture.mjs` against Zero 0.3.4 (build 5b3a90a) on 13th Gen Intel(R) Core(TM) i5-13450HX, 16 cores, win32/x64. Captured 2026-08-09.

1. Introduction

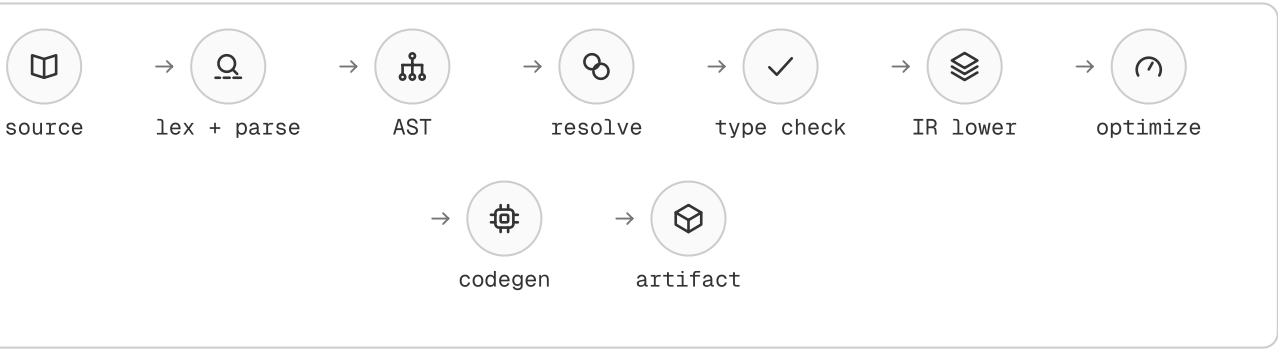
Every undergraduate compilers course opens with the same diagram: source text enters at the top, passes through a vertical stack of labelled boxes, and machine code emerges at the bottom. Two narrow rectangles run down the side of the stack, touching every box — the symbol table and the error handler. The diagram is due in its modern form to Aho, Lam, Sethi and Ullman (https://www.pearson.com/en-us/subject-catalog/p/compilers-principles-techniques-and-tools/P200000003472)[1], and it has organised the field for four decades.

The diagram is also, in an important sense, unfalsifiable by students. Real compilers fuse phases for speed, interleave them for incrementality, and expose none of the boundaries. A student who runs `gcc` or `rustc` sees a binary appear and, on failure, a paragraph of English. The phases are real, but they are not *visible*.

This paper asks a narrow question: **does the classical six-phase model still describe a compiler built on a fundamentally different substrate, and if not, how does it differ?** We answer it empirically — by building a corpus, instrumenting the compiler through its own reporting interfaces, and measuring.

Traditional parse-first compile path

9 stages · per invocation



Stages in order: source files → lexer/parser → AST → name resolution → type checking → IR lowering → optimization → codegen → artifact. All nine run on every invocation, as documented for Rust, Go, Zig and C. Figure 2 aligns them against Zero's sequence. Below 560px the ribbon becomes a vertical stack; the order is unchanged.

2. What Zero is

Zero (https://zerolang.ai/)[3] is an experimental systems programming language released by Vercel Labs in May 2026 under Apache-2.0, with source on GitHub (https://github.com/vercel-labs/zerolang). It compiles to standalone native executables, has no garbage collector, gives explicit control over memory, and sits in roughly the design space of C and Zig. Its compiler core is written in C. Programs use the `.0` extension. We study version 0.3.4, build 5b3a90a.

Its distinguishing premise is stated in its own tagline: *the programming language for agents*. Zero is built on the assumption that AI agents, rather than humans, will be the primary consumers of compiler output — and the design follows that assumption further than any other language we are aware of.

p01_hello / src/main.0 — the smallest complete program

```
pub fn main(world: World) -> Void raises {
    check world.out.write("hello from zero\n")
}
```

Figure 1. `pub fn` exports. `World` carries runtime capabilities. `raises` marks the function fallible, and `check` propagates failure. This compiles to a 1536-byte native executable in 6 ms of lowering.

2.1 How it differs from current languages

Four departures matter for this paper.

The semantic graph is the program, not the text. A Zero package compiles from a binary `zero.graph` store holding declarations, types, calls, scopes, imports, capabilities and source-map facts as rows in sixteen relations. The `.0` files a human reads are a *projection* of that graph. In every other mainstream language text is authoritative and the compiler's internal representation is derived and discarded; Zero inverts that relationship (https://zerolang.ai/concepts/semantic-vs-text)[6].

Diagnostics are data, not prose. Every command accepts `--json` against a versioned schema. Each diagnostic carries a stable code, a span, structured `expected` / `actual` facts, a `fixSafety` rating, and often a typed `repair` identifier.

Effects and capabilities are explicit and target-checked. A function performing I/O receives a `World` handle and is marked `raises`. Each target independently declares which capabilities it provides, so a program using the network is rejected at compile time for a target that declares no `net` capability.

The toolchain reports itself. A single `zero check --json` returns the phase list with timings, graph table row counts, the resolved call graph with contracts, cache keys with hit status, and a target readiness report. There is also `zero tokens`, which counts a program in LLM tokens.

2.2 Why its designers argue it is needed

The case Zero makes is about the cost of a translation layer. In a conventional agent loop, an agent writes text, the compiler renders its objection as English, and the agent parses that English back into an intent it already had. Zero's graph architecture documentation (https://zerolang.ai/concepts/graph-architecture)[4] frames the contrast as two loops.

Three specific failures are claimed. *Line ranges are the wrong handle* — they drift the moment anything reformats, whereas semantic node identifiers do not. *English is a lossy encoding of compiler state* — the compiler knew the exact repair and discarded that structure to print a sentence. *Feedback latency bounds the loop*.

A fourth motive is arguably strongest and less often stated: **capability containment**. For code an agent wrote and no human read line by line, a compiler that refuses to build a program using a capability the target does not declare is a real safety property.

We record the counterarguments in the same breath. Structured diagnostics are not new: rustc (https://doc.rust-lang.org/rustc/json.html) and TypeScript have emitted machine-readable errors for years. And the adoption argument cuts hard against it — models write best in the languages that dominate their training data, so a language with a few thousand GitHub stars is one agents are, today, measurably worse at than Go or Rust. Zero is betting a tight verifiable loop outruns familiarity. That bet is unproven, and this paper does not settle it.

---

3. How traditional compilers work

The canonical decomposition treats compilation as a sequence of meaning-preserving translations. Each phase consumes one representation and produces the next.

The **analysis** half — phases one to three, the front end — determines what the program means. Lexical analysis groups characters into tokens. Syntax analysis arranges tokens into a tree reflecting the grammar. Semantic analysis annotates that tree with types, resolves every name to a declaration, and rejects programs that are well-formed but meaningless.

The **synthesis** half — phases four to six, the back end — determines how the program runs: an intermediate representation, optimization over it, then instruction selection and object emission.

Two activities refuse to sit in any single box. The **symbol table** is written by the front end and read by every later phase. **Error detection** occurs in all six, and the phase that detects an error largely determines how good the message can be. That last point is the one we test in §6.4.

Phases 1–3 · at the ingestion gate

`zero import · once per edit`

phase 1

Lexical analysis

`Character stream → Token stream`

`parse` `locus: ingestion`

phase 2

Syntax analysis

`Token stream → Parse tree / AST`

`parse` `locus: ingestion`

phase 3

Semantic analysis

`AST + symbol table → Annotated AST, type facts`

`resolve` `interface` `check` `locus: both`

`Runs at both: admitted at ingestion, re-consulted from stored facts on the compile path.`

These three run when text enters the graph and are not re-run to produce a build. Phase 3 is the boundary case: it is admitted here and then re-consulted on the compile path from stored symbol, type and scope tables rather than rebuilt.

The admission boundary — `above: once per edit · below: once per build`

Phases 4–6 · on the compile path

`zero build · once per build`

phase 4

Intermediate code generation

`Annotated AST → Intermediate representation`

`lower` `locus: compile-path`

phase 5

Code optimization

`Intermediate representation → Improved IR`

`lower` `codegen` `locus: compile-path`

phase 6

Target code generation

`Optimized IR → Machine code / object`

`codegen` `object` `link` `locus: compile-path`

These three are what a build actually costs. They read a graph whose names are already bound and whose types are already recorded, which is why the compile path can begin at lowering.

`cross-cutting · spans phases 1-6`

Symbol table management

Classical

A data structure rebuilt by the front end on every compilation.

As Zero realises it

A persisted set of graph tables (schema, package, module, declaration, scope, import, symbol, type, effect, capability, ownership, resource, node, edge, projection, sourceMap) carried between runs in zero.graph and addressable by stable node handles.

`probe: zero query --json --full`

`cross-cutting · spans phases 1-6`

Error detection and reporting

Classical

Diagnostics rendered as prose for a human reader.

As Zero realises it

Structured records with a stable code, a span, expected/actual facts, a fixSafety rating and an optional typed repair id — designed to be consumed by a program, not parsed from English.

`probe: zero check --json | zero explain --json | zero fix --plan --json`

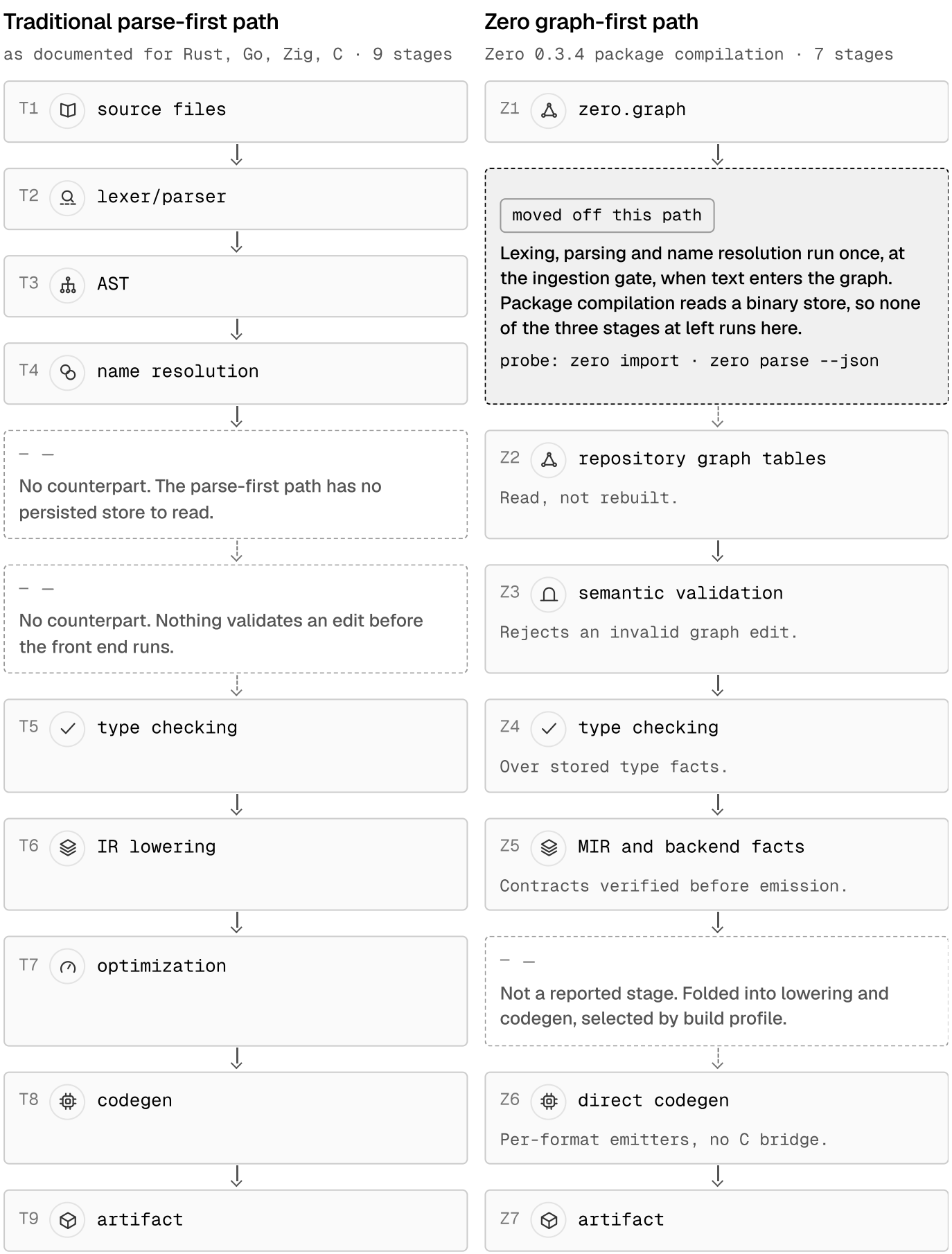
The classical six-phase stack, drawn once. The split is not a redrawing of the phases but a statement about cadence: phases 1–3 are paid when an edit is admitted, phases 4–6 when an artifact is requested. The two panels are the cross-cutting concerns the textbook draws as vertical bars beside the stack; in Zero both are persisted and directly inspectable, which is why the split above is observable rather than asserted. Phase numbering, input/output pairs, Zero phase names and locus values are read from `lib/phases.ts`. On a narrow screen the two bars move underneath the stack rather than beside it.

4. How Zero's compiler differs

Zero's own compile-path documentation (https://zerolang.ai/concepts/compile-path)[5] states the contrast directly. A conventional compiler runs source files through a lexer and parser to an AST, then name resolution, type checking, IR lowering, optimization and codegen. Zero starts from the graph store.

Zero graph-first compile path `7 stages · per zero build`

Stages in order: zero.graph → repository graph tables → semantic validation → type checking → MIR and backend facts → direct codegen → artifact. There is no lexer, parser or name-resolution stage here — those ran once, at the ingestion gate, when text entered the graph. Figure 2 is the stage-by-stage comparison. Below 560px the ribbon becomes a vertical stack; the order is unchanged.



Both tracks read top to bottom. Slots are aligned, so a dashed box marks a stage one path has and the other does not: three traditional stages leave the Zero build path entirely, one more (optimization) is folded into two others, and two Zero stages have no traditional counterpart. Stage labels are the documented names, unabbreviated.

Text alternative to the diagram above: the two compile paths as ordered stage sequences, 9 stages against 7. Rows are positional only — row n holds stage n of each sequence and does not assert a correspondence between them. — means the sequence has ended. No units; this table is definitional.

#	Traditional parse-first stage	Zero graph-first stage
1	source files	zero.graph
2	lexer/parser	repository graph tables
3	AST	semantic validation
4	name resolution	type checking
5	type checking	MIR and backend facts
6	IR lowering	direct codegen
7	optimization	artifact
8	codegen	–
9	artifact	–

The consequential difference is what is *missing* from the second sequence. There is no lexer, no parser and no separate name-resolution stage on the compile path, because by the time a program is in the graph store its names are already bound and its types already recorded. Those stages still exist — they run at the boundary where text is admitted into the graph.

This is why the compiler reports `resolve` *before* `parse`, an ordering that reads as a typo until you understand the substrate. The eight phases it reports, in its own order:

- `resolve` — Binds names against stored symbol facts.
- `parse` — Graph-native; no character scan on the package path.
- `interface` — Public symbol surface and import graph fingerprinting.
- `check` — Types, effects, ownership, capability requirements.
- `lower` — Graph HIR to MIR, with contract verification before emission.
- `codegen` — MIR to target machine code via direct emitters.
- `object` — Object-file construction in the target format.
- `link` — The only phase the compiler marks non-cacheable.

And the two cross-cutting concerns become first-class inspectable objects:

Symbol table management. A persisted set of graph tables (schema, package, module, declaration, scope, import, symbol, type, effect, capability, ownership, resource, node, edge, projection, sourceMap) carried between runs in zero.graph and addressable by stable node handles. Probe: `zero query --json --full`

Error detection and reporting. Structured records with a stable code, a span, expected/actual facts, a fixSafety rating and an optional typed repair id — designed to be consumed by a program, not parsed from English. Probe: `zero check --json | zero explain --json | zero fix --plan --json`

5. Methodology

5.1 Corpus construction

We wrote 8 Zero packages of increasing size and feature coverage, from a six-line hello-world to the 304-line arithmetic tokenizer in 🕮 `p08_lexer/src/lib.0`. Each was developed until `zero check`, `zero test` and `zero run` all succeeded, or until the obstruction was documented as a finding. The corpus totals 680 non-empty lines, 3,411 tokens and 23 passing test blocks. All eight are reproduced in full in §7.

5.2 Error corpus

Measuring where errors are detected requires programs that fail on purpose. We wrote 10 minimal packages, each violating exactly one rule and targeting a specific phase. Each carries a `case.json` declaring the phase it targets *before* measurement, so the mapping in §6.4 is a prediction rather than a post-hoc fit.

5.3 Instrumentation

One harness, `tools/capture.mjs`, drives every measurement and writes a single JSON document. Per program it captures the token stream, parse summary, full semantic report, graph, source mappings, phase timings and artifact measurements, then builds every program for every advertised target. We report phase timings from `zero time` rather than `zero check`, because the former drives the pipeline through emission; and wall-clock separately, as median of five runs, because ~40 ms of process startup dominates at this scale.

5.4 Reproducibility

Every number derives from one machine-generated dataset. No figure is transcribed by hand; the prose reads the same JSON the charts do. Appendix A gives the commands. The environment was an 13th Gen Intel(R) Core(TM) i5-13450HX with 16 logical cores and 31.7 GB of memory, running win32/x64.

---

6. Results

6.1 Mapping the classical phases onto Zero

Zero reports eight phases; the classical model names six. They do not correspond one-to-one, and the mismatches are informative.

The six canonical front-to-back compiler phases (Aho, Lam, Sethi and Ullman) against the phase names Zero 0.3.4 reports in its own `--json` output. One mapping row per phase, followed by a row stating where the realisation departs from the textbook. Counts and units: none — this table is definitional.

#	Classical phase	Input → output	Zero phases	Probe	Locus
1	Lexical analysis	Character stream → Token stream	`parse`	`zero tokens --json`	ingestion
	Divergence — Runs only when text enters the system. Package compilation reads a binary graph store and never re-scans characters, so the scanner is absent from the steady-state path.				
2	Syntax analysis	Token stream → Parse tree / AST	`parse`	`zero parse --json`	ingestion
	Divergence — The tree is not the compiler's working representation. Parsing exists to admit text into the graph; once admitted, structure is stored, not re-derived.				
3	Semantic analysis	AST + symbol table → Annotated AST, type facts	`resolve` `interface` `check`	`zero check --json`	both
	Divergence — Split across three reported phases and persisted as typed graph facts. The symbol table is not rebuilt per compile — it is the stored `symbol`, `type` and `scope` tables.				
4	Intermediate code generation	Annotated AST → Intermediate representation	`lower`	`zero size --json (loweredIrBytes)`	compile-path
	Divergence — Lowering runs graph HIR to MIR directly, and MIR contracts are verified before emission. The IR is never printed: --emit llvm-ir fails with BLD004 on every target, because the direct backend has no LLVM path. The only observable is its size in bytes.				

#	Classical phase	Input → output	Zero phases	Probe	Locus
5	Code optimization	Intermediate representation → Improved IR	`lower` `codegen`	`zero build --profile release-small | tiny`	compile-path

Divergence — Not a separately reported phase. Optimization is selected by build profile rather than exposed as a pass pipeline, so its cost is folded into lower and codegen.

#	Classical phase	Input → output	Zero phases	Probe	Locus
6	Target code generation	Optimized IR → Machine code / object	`codegen` `object` `link`	`zero build --emit exe && zero size --json`	compile-path

Divergence — Three reported phases, not one. Direct per-format emitters replace a C bridge, and only `link` is marked non-cacheable.

Semantic analysis fragments into three reported phases because interface fingerprinting is separated out to drive incremental invalidation. Optimization has *no* reported phase: it is selected by build profile. And the IR is never printed — `--emit llvm-ir` fails with BLD004 on every target, leaving the lowered module's *size* as the only observable.

6.2 Where compile time actually goes

Across all 8 programs, 100% of reported phase milliseconds are spent in `lower`. Every other phase reports 0 ms.

Lowering cost by program — `zero time --json, cold`

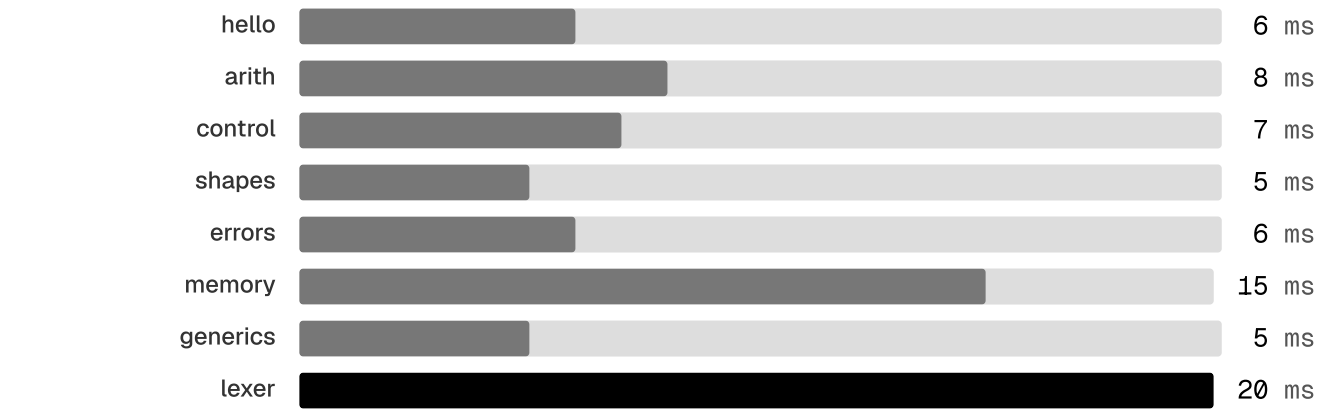


Figure 5. Milliseconds in `lower`. Every other phase reports 0 ms.

Graph size against source size — 8 programs

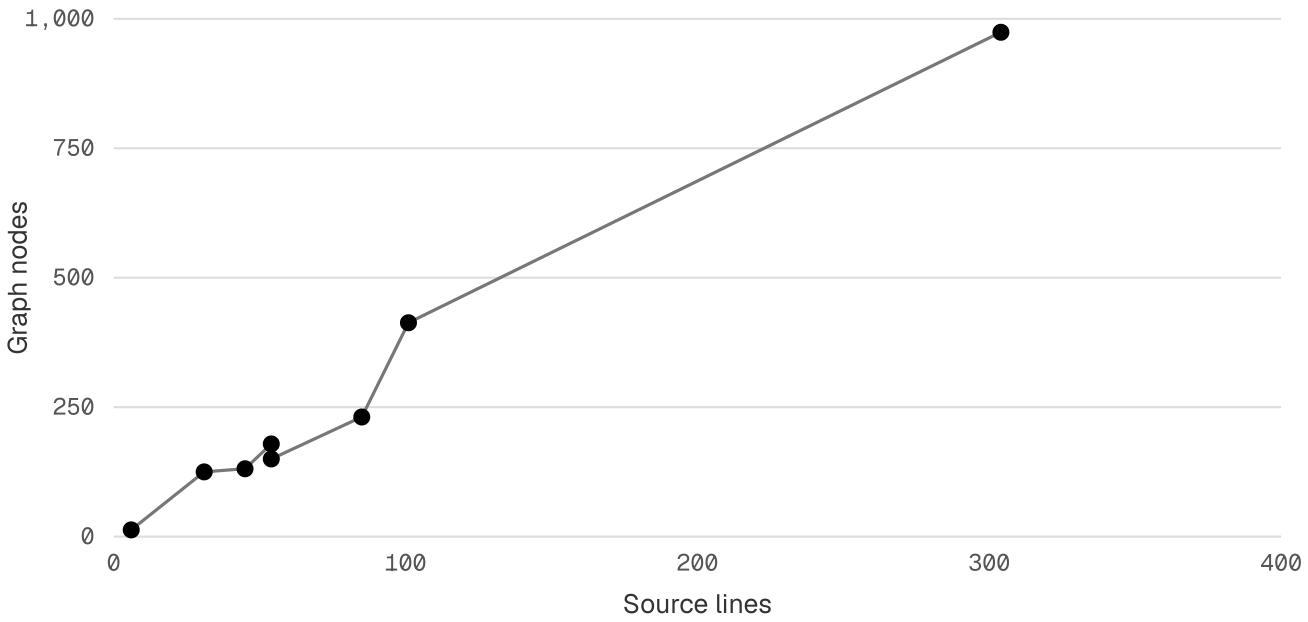


Figure 6. ≈3.3 graph nodes per source line — a constant-factor expansion, not a blow-up.

This is the expected consequence of the architecture. In a text-first compiler the front end reconstructs meaning from characters on every invocation. In Zero it reads a store where names are already bound, so it does almost no work. What remains expensive is the one phase that cannot be cached away.

The honest caveat is resolution: the compiler reports integer milliseconds, so "0 ms" means "under one millisecond", not "free".

Phase composition, largest program — p08_lexer · 304 lines

p08_lexer, cold cache · total 20.0 ms

Figure 7. All eight reported phases. The bar is effectively one segment: `lower` at 20 ms.

6.3 The symbol table, persisted

In the classical model the symbol table is a structure the front end builds and discards. In Zero it is sixteen persisted relations, carried between invocations and reported as row counts on every compile.

All 8 corpus programs in corpus order, measured at each stage of the pipeline. Lines and bytes are source text; tokens are the total emitted by `zero tokens` including newlines; nodes, edges, declarations, symbols and types are row counts in the persisted graph; typed nodes and calls come from `zero check`; artifact is the host-target executable in bytes, or — where the backend refused to build it.

Program	Lines	Bytes	Tokens	Nodes	Edges	Decls	Symbols	Types	Typed nodes	Calls	Artifact bytes
p01_hello	8	140	41	13	12	2	3	5	5	1	1,536
p02_arith	40	781	219	125	123	15	16	29	29	19	2,048
p03_control	62	1,624	297	179	177	16	17	31	31	21	2,048

Program	Lines	Bytes	Tokens	Nodes	Edges	Decls	Symbols	Types	Typed nodes	Calls	Artifact bytes
p04_shapes	102	2,478	471	231	229	33	34	55	55	24	2,048
p05_errors	66	1,984	292	150	148	20	21	35	35	19	—
p06_memory	123	3,518	786	413	411	49	47	96	96	45	3,584
p07_generics	56	1,298	277	131	129	20	21	37	37	20	1,536
p08_lexer	335	10,390	1,804	974	972	108	109	176	176	106	5,632

Two structural invariants hold across the corpus. The `sourceMap` row count equals the `node` count exactly for every program — every graph node retains a source position. And the edge count is exactly two fewer than the node count, consistent with a spanning structure over the module and package roots.

6.4 Error handling: three admission gates, not one

This is the paper's central empirical result. We expected the error corpus to partition by phase within a single compile. Instead it partitions by *gate*, and there are three.

10 deliberately malformed programs enter at the left. Each gate admits what it cannot fault and turns the rest away downward.

gate 1 of 3

zero import

Front end

`lexical` `syntax` `name resolution` `type` `mutability` `effect` `memory`

entered	10
rejected here	7
continue to zero check --target	3

↓

Rejected: 7 of 10

PAR100	Parsing	×2
ERR003	Effect checking	×1
MEM003	Memory / frame budget	×1
NAM003	Name resolution	×1
TYP002	Type checking	×1
TYP009	Type checking	×1

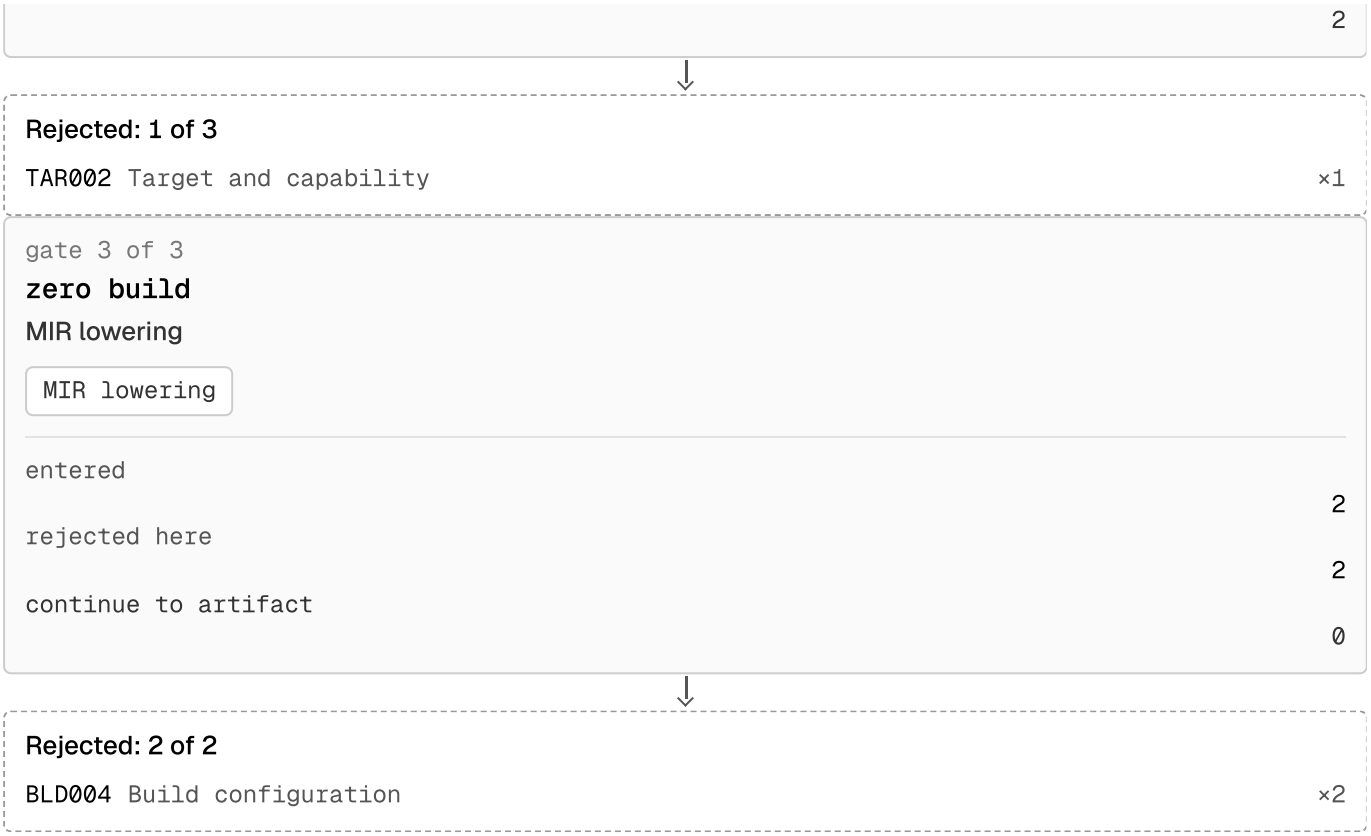
gate 2 of 3

zero check --target

Target capability

`target capability`

entered	3
rejected here	1
continue to zero build	



Counts and codes are derived at render time from `capture.errorCases`: a case is attributed to the gate its `rejectedAt` field names, and the codes are the diagnostics that gate itself emitted. 0 of 10 malformed programs reach an artifact. On a narrow screen the gates stack, so the left-to-right flow becomes top to bottom; the rejected branch stays directly beneath its gate either way.

Text alternative to the diagram above: the three admission gates in order, with the population entering each, the number it rejected, the number it passed on, and the diagnostic codes it rejected them with. Population: the 10 error-corpus programs. Counts are programs, not diagnostic lines.

#	Gate	Checks	Entered	Rejected	Continued	Codes rejected with
1	`zero import`	lexical, syntax, name resolution, type, mutability, effect, memory	10	7	3	`PAR100 ×2, ERR003 ×1, MEM003 ×1, NAM003 ×1, TYP002 ×1, TYP009 ×1`
2	`zero check --target`	target capability	3	1	2	`TAR002 ×1`
3	`zero build`	MIR lowering	2	2	0	`BLD004 ×2`
Reached an artifact				—	0	—

The 10 deliberately malformed programs in the error corpus, one row each. The three gate columns record the outcome of `zero import`, `zero check` and `zero build`: “ok” passed, “rejected” was refused, — was never reached. No units.

Case	Targets phase	Import	Check	Build	Rejected at	Codes
`e01_lexical`	lexical	rejected	ok	—	import	`PAR100`
`e02_syntax`	syntax	rejected	ok	—	import	`PAR100`
`e03_name`	resolve	rejected	ok	—	import	`NAM003`
`e04_type`	check	rejected	ok	—	import	`TYP002`
`e05_mutability`	check	rejected	ok	—	import	`TYP009`
`e06_effect`	check	rejected	ok	—	import	`ERR003`
`e07_memory`	check	rejected	ok	—	import	`MEM003`
`e08_target`	target	ok	rejected	rejected	check	`TAR002`

Case	Targets phase	Import	Check	Build	Rejected at	Codes
`e09_lowering`	lower	ok	ok	rejected	build	BLD004
`e10_match`	—	ok	ok	rejected	build	BLD004

The 7 front-end failures were all refused at `zero import`. Critically, `zero check` subsequently reported `ok` for all of them, because the malformed program never entered the store — after each rejection `zero view --fn main` still projected the *previous* program. The capability violation passed ingestion and was refused by `zero check --target`. And 2 cases passed both and were refused only at `zero build`.

6.5 The front end and the back end accept different languages

We built every corpus program for every advertised target: 64 combinations. 47 succeeded (73%). Every one of the 17 failures was BLD004 — the back end declining to lower a construct semantic analysis had accepted.

Build outcome for every corpus program on every target the installed toolchain advertises (8 programs × 8 targets = 64 builds, 17 of them refused). A cell holding a number is the emitted artifact size in bytes; a cell holding a diagnostic code is a refusal, and the code is the marker.

Program	darwin-arm64	darwin-x64	linux-musl-x64	linux-musl-arm64	linux-x64	linux-arm64	win32-x64.exe	win32-arm64.exe
`p01_hello`	16,632	16,636	352	312	352	312	1,536	1,536
`p02_arith`	16,632	16,636	633	649	633	649	2,048	2,048
`p03_control`	16,632	16,636	795	867	795	867	2,048	2,048
`p04_shapes`	16,632	16,636	826	BLD004	826	BLD004	2,048	BLD004
`p05_errors`	16,632	BLD004	826	BLD004	826	BLD004	BLD004	BLD004
`p06_memory`	16,632	16,636	2,439	BLD004	2,439	BLD004	3,584	BLD004
`p07_generics`	16,632	16,636	588	BLD004	588	BLD004	1,536	BLD004
`p08_lexer`	16,632	16,636	4,237	BLD004	4,237	BLD004	5,632	BLD004
Built	**8/8**	**7/8**	**8/8**	**3/8**	**8/8**	**3/8**	**7/8**	**3/8**

BLD004 · `record` — 9 builds

BLD004 · `IR_VALUE_CHECK` — 4 builds

BLD004 · `unsupported instruction` — 3 builds

BLD004 · `IR_VALUE_RESCUE` — 1 build

The failures name specific constructs the MIR subset cannot represent: aggregate values crossing a function boundary, `check` and `rescue` on user-defined fallible functions, and instructions absent from an architecture's emitter.

First, **backend completeness is target-specific.** The `p05_errors` program fails to build on the Windows host but cross-compiles to an 826-byte ELF for `linux-musl-x64`. Same graph, same front end, two answers.

Second, **a passing `zero check` does not imply a buildable program** — though the compiler is not ignorant. It records `targetReadiness.buildable: false`, `stage: "lower"` and a BLD004 naming the construct. The difficulty is that the same document carries `ok: true` with an empty top-level `diagnostics` array. A consumer testing the field the schema presents as the verdict gets the wrong answer.

Table 6. Fields reported by `zero check --json` for the two lowering cases. Both rows describe the same compilation.

Case	Construct	`ok`	`diagnostics`	`buildable`	`stage`	Actual build
`e09_lowering`	`Outcome`	`true`	`0`	`false`	`lower`	BLD004

Case	Construct	ok	diagnostics	buildable	stage	Actual build
e10_match	Match	true	0	false	lower	BLD004

Sections 7 to 9 are interactive

Three parts of this study cannot be printed, because they are things you operate rather than things you read:

§7 The corpus in full — all 8 programs, 680 lines of Zero, with their measurements and real output, plus a stepped replay of one program being compiled.
§8 Phase explorer — one program walked through all six classical phases, showing the artifact the compiler actually emitted at each.
§9 Playground — write Zero and have it lexed and parsed in the browser by a reimplementation that agrees exactly with `zero tokens --json` on all 8 corpus programs.

Scan the code, or visit **zero.khe.money**. The complete dataset and the harness that produced it are at github.com/HKTITAN/phases-of-zero.

10. When the reader is a program

Zero is built on the premise that its output will be consumed by a program rather than read by a person. Every reporting command has a `--json` form: tokens, parse trees, the semantic graph, phase timings, artifact sizes and diagnostics all answer in a schema on request. Diagnostics carry a stable code, a span, expected and actual facts, a fix-safety rating and a typed repair identifier instead of a sentence. The graph can be queried without re-reading a character of source. The implicit promise is efficiency: a machine reader should not have to pay for prose that was shaped for a human.

We measured that promise, and it does not hold. The result is worth stating before the method rather than after it, because it runs the opposite way to the intuition the design invites.

FINDING

Structured output is **far more expensive** than prose, not less. Asked whether a program is correct, `zero check` answers in 4 bytes. `zero check --json` answers the same question for the same 6-line program in 17,898 bytes, and for the largest program in the corpus in 189,743 bytes — a factor of 17,625× over the corpus as a whole.

10.1 Method: one question, two answer forms

For each of the 8 corpus programs the harness asks the compiler three questions twice — once in the form a person would read, once in the form a program would parse — and records the size of each answer. *Show me one function* is `zero view --fn main` against `zero query --json --fn main`. *What does this program call, and is each call checked* is the whole source against `zero query --json --calls std`. *Is this program correct* is `zero check` against `zero check --json`. Byte counts are exact; token figures are estimated at four characters per token and are only ever estimates.

Table 10.1 — Cost of answering the same question in text and in structured form, for each of the 8 corpus programs. All figures are bytes of command output, measured on the capture host. “Source” is every byte of every file in the package, which is what a reader with no query interface must ingest.

Program	Lines	Source	`view --fn`	`query --json --fn`	`check`	`check --json`
`p01_hello`	6	141	95	3,286	4	17,898
`p02_arith`	31	782	128	3,924	4	47,207
`p03_control`	54	1,625	196	4,341	4	46,633
`p04_shapes`	85	2,479	355	4,972	4	59,943
`p05_errors`	54	1,985	262	4,475	4	51,380
`p06_memory`	101	3,519	752	7,001	4	104,221
`p07_generics`	45	1,299	147	3,951	4	46,971
`p08_lexer`	304	10,391	723	6,106	4	189,743
Corpus total	**680**	**22,221**	**2,658**	**38,056**	**32**	**563,996**

Two effects are tangled together in that table, and separating them is the whole argument. **Targeting works.** Reading one function through `zero view --fn` costs 2,658 bytes across the corpus against 22,221 bytes of source — 8.4× cheaper — because the compiler already knows where the function is and a text-first reader does not.

Structuring does not. The same targeted answer requested as data costs 38,056 bytes: 14.3× the text view, and 1.7× the cost of simply reading every line of every program in the corpus. The saving that targeting earns is spent on the encoding, and then some. At the extreme, the verdict on a 6-line program grows from 4 bytes to 17,898.

So the trade a machine-first compiler offers is not tokens for tokens. It is tokens for actionability, and the next table is what the extra tokens buy.

10.2 What the extra bytes carry

For diagnostics the premium is much smaller and the return is much clearer. Across the 7 error cases the ingestion gate refuses, prose costs 2,061 bytes and the structured form costs 4,796 — 2.3× in aggregate, 1.9× to 4.0× case by case. For that the structured form exposes 9 to 10 separately addressable fields and a typed repair identifier on every one.

Table 10.2 — `zero import` against `zero import --json` for all 10 error cases. Bytes are the complete command output including stderr. “Machine fields” counts the diagnostic fields a consumer can read by name without parsing English. The 3 cases the gate admits carry no diagnostic at all, which is why their field count is zero.

Case	Import	Code	Prose	JSON	Ratio	Machine fields	Typed repair	Fix safety
`e01_lexical`	rejected	`PAR100`	175	586	3.3×	10	yes	requires-human-review
`e02_syntax`	rejected	`PAR100`	141	561	4.0×	9	yes	requires-human-review
`e03_name`	rejected	`NAM003`	330	692	2.1×	10	yes	requires-human-review
`e04_type`	rejected	`TYP002`	276	584	2.1×	10	yes	behavior-preserving
`e05_mutability`	rejected	`TYP009`	322	734	2.3×	10	yes	behavior-preserving
`e06_effect`	rejected	`ERR003`	350	770	2.2×	10	yes	api-changing
`e07_memory`	rejected	`MEM003`	467	869	1.9×	10	yes	requires-human-review
`e08_target`	accepted	–	56	2,240	40.0×	0	no	—
`e09_lowering`	accepted	–	58	2,251	38.8×	0	no	—
`e10_match`	accepted	–	55	2,236	40.7×	0	no	—

The distinction the ratio column hides is not information but *addressability*. Zero's prose diagnostic is not terse — it carries a code, a path, a line and column and a help line in 294 bytes on average. What it does not carry is a schema. A consumer

wanting the expected type has to find it inside a sentence; a consumer wanting to know whether an automated fix would change behaviour has to infer it. The structured form names both, and adds one thing the prose has no equivalent of: a repair identifier drawn from a closed vocabulary. Our 10 error cases between them produce 8 distinct identifiers — `check-or-rescue-fallible-call`, `choose-supported-backend`, `choose-target-with-required-capability`, `declare-missing-symbol`, `make-binding-mutable`, `manual-review`, `move-large-locals-off-stack`, `repair-syntax` — each paired with one of 3 fix-safety ratings: `requires-human-review`, `behavior-preserving` and `api-changing`. A program can branch on those without a language model in the loop. That is what the premium buys, and it is a real thing to buy.

The premium is worst where there is nothing to say. The 3 cases the gate admits produce 169 bytes of prose between them and 6,727 bytes of JSON — 39.8× to report success. A structured schema pays its fixed cost whether or not the run had anything to report, and most runs do not.

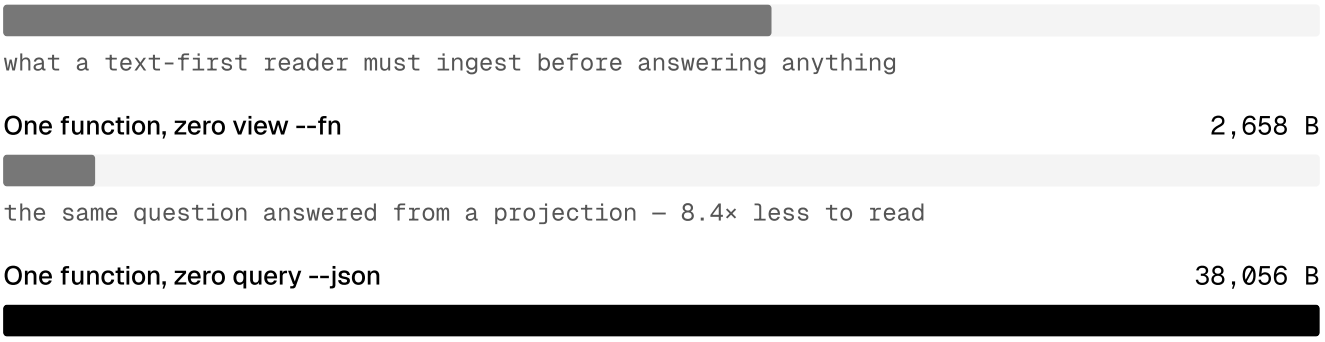
10.3 The token budget, and what is on the wrong side of it

Zero ships `zero tokens` as a first-class command. A token count is something the compiler will tell you about a program on request, in the same way it will report its phase timings or its artifact size — and the token is the unit in which a language model is metered. A compiler that publishes one is a compiler that expects to be read by something that counts.

Set that against Table 10.1. The compiler reports 35 tokens for `p01_hello`, and returns 17,898 bytes — roughly 4,475 estimated model tokens — when asked in JSON whether those 35 tokens are correct. Across the corpus the structured verdict runs between 32× the size of the program it describes (`p08_lexer`) and 128× (`p01_hello`). The compiler's account of a program is consistently, and by a wide margin, the largest artifact in the exchange.

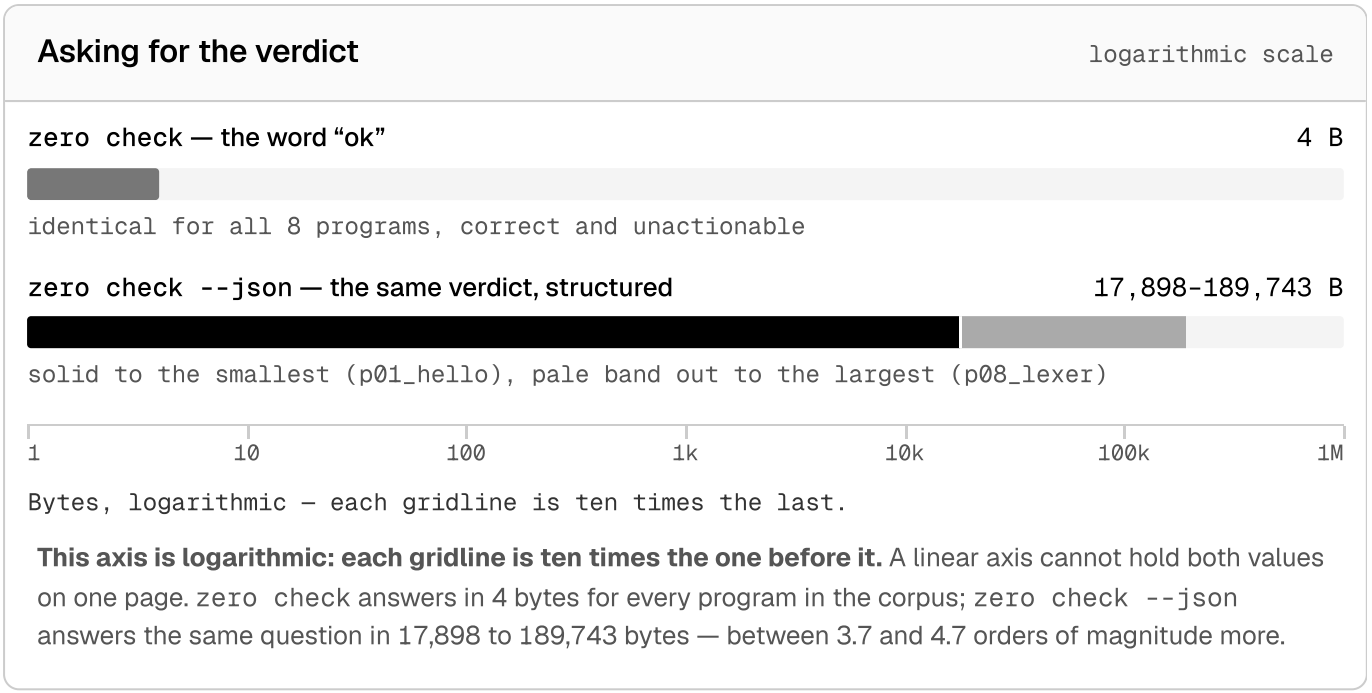
We do not read this as an argument against structured output; §10.2 is an argument for it. We read it as a measurement that the structured interface has not yet been costed for the reader it was designed for. Nothing in the schema is negotiable per call: we found no field selection, no severity filter and no way to ask `zero check --json` for the verdict without the report that surrounds it. A compiler whose stated audience is billed by the token has, on this build, no way to ask it for less.

Reading one function instead of the whole program — 8 programs, linear scale



ⓘ The third bar cuts against the premise. Structured retrieval only pays off once a program is large: 7 of 8 programs cost more to query one function from than to read end to end.

Totals across all 8 corpus programs. `zero view --fn` is the projection a reader actually needs, and it costs 8.4× less than reading every source file — but only for large programs: the ratio runs from 1.5× on `p01_hello` to 14.4× on `p08_lexer`.



Text alternative to both figures above. Measured bytes per program for each way of asking the same two questions: “show me one function” and “is this program correct?”. Token figures in the paper are estimates at roughly four characters per token; these are bytes, measured.

Program	Whole source	view --fn	query --json	check	check --json
p01_hello	141	95	3,286	4	17,898
p02_arith	782	128	3,924	4	47,207
p03_control	1,625	196	4,341	4	46,633
p04_shapes	2,479	355	4,972	4	59,943
p05_errors	1,985	262	4,475	4	51,380
p06_memory	3,519	752	7,001	4	104,221
p07_generics	1,299	147	3,951	4	46,971
p08_lexer	10,391	723	6,106	4	189,743
Total	22,221	2,658	38,056	32	563,996

11. Discussion

11.1 Phases relocate; they do not disappear

It would be easy to read Zero's architecture as abolishing the front end. It does not. Every classical phase is present and every one still runs — but the first three moved out of the compile path into an ingestion gate that runs once per edit rather than once per build.

This relocation explains all our findings at once: the timing distribution in §6.2, the containment result in §6.4, and the acceptance gap in §6.5. One architectural choice, three consequences.

11.2 Two levels of truth in one document

Our first reading of the acceptance gap was wrong: we recorded it as the compiler failing to detect a problem. It detects it. `targetReadiness` carries the correct verdict and a diagnostic naming the offending construct.

What the compiler does is subtler and, for its stated audience, arguably worse than not checking: it publishes two summaries of the same compilation that disagree. A human reading the whole document notices. A program reading the field the schema presents as the verdict does not. A language whose central claim is that output should be consumed by machines has taken

on an obligation a human-facing compiler has not: its top-level fields are an API. Machine-readability is necessary but not sufficient for machine-*reliability*.

11.3 Implications for compiler pedagogy

Our practical conclusion is narrower than Zero's marketing and, we think, more durable. A student using Zero can print the phase list, time each phase, read the symbol table as sixteen relations, watch a program be refused at three distinct gates, and diff the object formats produced by five emitters from one source graph — from the command line, without patching a compiler. We are not aware of another production-intent compiler where the phase structure is this directly inspectable.

12. What this study does not cover

A phase-by-phase account of a compiler invites the reader to assume that everything in the textbook was examined. It was not. Five classical topics are absent from our results, and in each case the reason is different: one is absent because Zero does not implement it, two because Zero does not expose them, one because Zero replaces it with something that is not a phase, and one because our method could not reach it. Naming which is which is more useful than an apology.

> SCOPE
>
> Not covered: error recovery, incremental invalidation cost, register allocation, instruction selection, and bootstrapping the compiler in its own source language. The optimization phase is covered only in the form Zero provides it — a fixed catalogue of build profiles rather than a pass pipeline.

12.1 Error recovery

A classical front end is expected to recover. On a syntax error it discards tokens to a synchronising symbol, resumes, and reports as many independent errors per run as it can without inventing them. The quality of that resynchronisation is a research topic in its own right, and it is the difference between a compiler that costs one edit-compile cycle per error and one that costs a cycle per *run*.

Zero does not recover. Its ingestion gate either admits an edit into the graph or refuses it whole. We did not set out to measure this and cannot claim to have tested it properly — every case in our error corpus seeds exactly one defect — but the consequence is visible in the output all the same.

10 error cases in the corpus	**1** diagnostics reported per run
3 carrying a related location	**0** cases that reached the graph store

Every one of the 10 cases produced exactly one diagnostic, at whichever gate refused it. 3 of them attach a related source location to that single diagnostic — a cross-reference inside one record, not a second finding. The store was never contaminated: a refused edit leaves no partial state behind, which is why a second run of the same command reports the same one diagnostic rather than a different one.

What that buys is the absence of the cascade. A missing closing brace in a conventional compiler produces a first honest error and then a column of phantom ones caused by the parser's own recovery guess, and a reader — human or otherwise — has to decide which ones are real. Zero never presents that decision. What it costs is round trips: with one diagnostic per run, an edit containing five defects takes five refusals to clear, and for a caller paying per round trip that is five times the fixed cost measured in §10.

We are explicit that this is an observation, not a finding. Establishing it would need a corpus of multi-defect programs and a comparison against a recovering front end on the same inputs. Neither exists here.

12.2 Incremental compilation and the cost of invalidation

Zero reports its caches, and this is the part of the compiler where its self-description is most complete: 6 named caches, each with a key, a hit flag and a plain-language statement of what invalidates it. We reproduce that report rather than summarise it, because the `invalidates0n` column is the design.

The table contains a result we did not expect and should not bury. After `zero clean --all`, the first run still reports 5 of 6 caches as hits. Only `emitted0bject` misses, and it misses on the warm run too. The reason is in the `invalidates0n` column: every cache above it is keyed on the *ProgramGraph input*, and cleaning a build directory does not change the graph. Our “cold” measurement is therefore a cold artifact directory, not a cold cache.

That has a direct consequence for scope. We never observed a miss on `parseTree`, `interface`, `checkedBody`, `specialization` or `mappedFinalMir`, so we cannot report the cost of an invalidation — which is the only number that matters for incremental compilation. Producing it would require mutating source between runs and re-measuring, which our harness does not do.

The same gap covers interface fingerprinting. The `interface` cache invalidates on *graph public symbols/import graph*, and the compiler describes its strategy as *fingerprint changed modules and dependent bodies*. The point of that design is that editing a function body without changing its signature should not force dependents to be re-checked. Our packages contain at most 2 modules and none depends on another package, so the longest dependency chain the fingerprint could protect is 1 edge long. There is no dependent far enough away for the optimisation to show. We report the mechanism; we do not report evidence that it works.

12.3 Register allocation and instruction selection

These are the two topics a back-end course spends the most time on, and this study says nothing about either. That is not a choice we made. Zero's reported phase list ends `lower → codegen → object → link`, and none of those four decomposes further in any `--json` payload we could find: there is no allocator report, no instruction-selection trace, no spill count, no register pressure figure.

Nor can the question be approached from the artifact side. Asking for the intermediate form directly fails on every corpus program with `BLD004: direct backend does not support --emit llvm-ir`. The direct emitters go from MIR to 3 object formats (macho, elf, coff) without an inspectable middle, so the only back-end observables the compiler offers are the size of the lowered IR in bytes, the size of the artifact, and whether the build succeeded. Everything a classical back-end chapter is about happens inside a step that reports one number.

12.4 Optimization is a profile, not a pass pipeline

Phase five of the classical model is code optimization, and Zero has no phase by that name. What it has instead is a fixed catalogue of 6 build profiles, each a named bundle of a codegen setting, a link setting, a metadata retention policy and a size budget. Selecting `--profile tiny` is the closest a user gets to requesting an optimization, and the request is categorical rather than compositional: there is no `-O2`, and no way to enable one transformation without the rest of its bundle.

Read down the goal column and the catalogue turns out not to be a speed dial at all. Exactly 1 of the 6 profiles names throughput as its goal; 2 name binary size at different intensities, and the rest name observability, edit latency and release auditability. That is a defensible set of axes for a compiler aimed at machine-generated code, where binary size and reproducible metadata matter more than the last few percent of a benchmark. But it means the classical question — which transformations ran, in what order, and what did each one buy — has no answer here, and we do not pretend to have measured one. Our §6 figures fold optimization cost into `lower` and `codegen` because the compiler does.

12.5 Bootstrapping and self-hosting

Whether a compiler can compile itself is the traditional closing chapter, and it is the one topic here where Zero answers the question directly and the answer is short: not yet, by design, and it has removed the machinery it would have used to get there.

The mode is `native-bootstrap` and every reported phase — `parse`, `check`, `lower`, `emit` — routes to `zero-c`. None routes to a compiler written in Zero. The three components a bootstrap normally needs are all recorded as removed: `seedCompiler`, `browserCompiler`, `portableEmitter`. The C bridge is gone with them, replaced by direct per-format emitters.

This is a coherent position rather than an omission — a compiler that emits 3 object formats directly has no need of a portable C fallback, and removing the seed compiler removes a whole class of trust problem. But it means the classical bootstrapping exercise cannot be run on this build, and a compiler that is not written in its own language has not yet made the argument that the language is adequate for compilers. We note the same restriction bites elsewhere in this paper: the browser playground in §9 reimplements the lexer in TypeScript precisely because `browserCompiler` is one of the removed components.

One last piece of scope worth naming, since it is easy to miss. The 16 graph tables in §6 are what the compiler chooses to publish. Nothing in this study inspects the graph store's own encoding, its index structures, or its behaviour under concurrent writers. We measured a reporting interface, and a reporting interface is not an implementation.

---

13. Threats to validity

Single version, single platform. All measurements are from Zero 0.3.4 build 5b3a90a on one Windows x64 machine. Zero is explicitly experimental and shipped four minor versions in roughly a month; the backend gaps we document are the ones most likely to close.

Corpus scale. 680 lines across 8 programs is small. It is adequate for demonstrating phase structure and the acceptance gap, both qualitative properties, but too small to resolve front-end phase timings against the compiler's 1 ms granularity or to claim asymptotic scaling.

Corpus authorship. The corpus was written to exercise particular compiler tables, so coverage is deliberate rather than representative. Several programs were shaped by what the backend would accept, which biases them toward the lowerable subset — if anything this *understates* the acceptance gap.

Timing methodology. Wall-clock includes ~40 ms of process startup we could not separate without instrumenting the binary, so all phase-level claims rest on the compiler's self-reported times.

Browser reimplementation. The playground's lexer and parser are ours, not Zero's. They agree exactly with `zero tokens --json` across all 8 corpus programs, but that corpus is 680 lines written by us: agreement on it is evidence of correctness on the constructs we used, not a proof of equivalence. Input using syntax the corpus never exercises may diverge, and only the native compiler is authoritative.

Documentation discrepancies. Several examples in the shipped language guide do not compile on this build. We report these as observations about version 0.3.4, not claims about the language design.

---

14. Related work

The phase decomposition we test against is due to Aho, Lam, Sethi and Ullman (https://www.pearson.com/en-us/subject-catalog/p/compilers-principles-techniques-and-tools/P200000003472)[1], with incremental and query-driven alternatives surveyed by Cooper and Torczon (https://www.elsevier.com/books/engineering-a-compiler/cooper/978-0-12-815412-0)[2]. Zero's persistent-graph approach has clear antecedents: the Roslyn (https://github.com/dotnet/roslyn)[8] compiler platform exposed compilation as a queryable object model, and query-based architectures — rustc's demand-driven query system (https://rustc-dev-guide.rust-lang.org/query.html)[9] and the salsa (https://github.com/salsa-rs/salsa)[11] framework behind Rust Analyzer — already treat compilation as incremental recomputation over a memoised fact database. Zero's distinguishing move is making the graph the *authoritative stored artifact*, with text demoted to a projection.

On structured diagnostics, both rustc (https://doc.rust-lang.org/rustc/json.html) and TypeScript ship machine-readable error output; Zero's contribution is typed repair identifiers and safety ratings rather than machine-readability itself.

We are not aware of prior empirical work measuring a compiler's phase structure through its own reporting interface, largely because few compilers expose one.

---

15. Outlook

It is worth separating what this study licenses us to say about where compilers are going from what we would merely like to be true. The table states which is which for each claim we make below; the prose then argues them in order.

Table 15.1 — The four claims in this section, the evidence each rests on, and its epistemic status. “Measured” means the figure appears in this paper's dataset; “argued” means it is an inference from those figures; “speculation” means we have no evidence and say so.

Claim	What it rests on	Status
Compiler interfaces are becoming APIs.	Every structural claim in this paper was read out of a documented --json schema. None required patching the compiler, scraping a log, or parsing an English sentence.	Measured
A top-level field carries an obligation prose never did.	2 of 10 error cases report ok: true at the top level of zero check --json while the nested targetReadiness reports buildable: false.	Measured
Observability is worth having whether or not agent-oriented languages win.	16 graph tables, per-phase timings, 3 distinct refusal gates and 8 target emitters, all reachable from a shell without a debugger.	Argued from the measurements
Adoption will be decided by pretraining distribution, not interface quality.	Nothing in this study bears on it. We have measured one compiler, not a market.	Speculation

15.1 The interface becomes the contract

The most durable observation in this paper is also the least dramatic: we were able to write it. Every claim we make about the compiler's structure came out of a documented `--json` schema — the exceptions are wall-clock times, which no compiler can report about itself, and the sizes of the compiler's own prose output in §10, which are the point of the comparison. A decade ago the equivalent study would have required instrumenting a compiler; here the instrumentation was the product. That direction of travel is not unique to Zero — `rustc --error-format=json` and the TypeScript compiler API arrived at the same place from a different premise — and it is the part of Zero's design we would expect to generalise regardless of what happens to the language.

The consequence is an obligation that prose never carried. An English diagnostic that overstates its confidence is read by a person who can weigh it against the rest of the output. A JSON field named `ok` is not weighed; it is branched on. We found 2 of 10 error cases where `zero check --json` reports `ok: true` at the top level while its own nested `targetReadiness` reports `buildable: false`, with a diagnostic naming the construct the backend will refuse. The compiler knew in every one of those cases. A human reading the whole document notices the disagreement. A program reading the field the schema presents as the verdict does not.

We think this is the general shape of the problem rather than a bug in one build. Publishing a schema converts every top-level field into a promise with a much wider blast radius than a sentence, and a compiler that adds a field faster than it can define what the field means will produce exactly this class of contradiction. Machine-readable is a property of a format. Machine-*reliable* is a property of a contract, and it is a harder thing to ship.

15.2 Observability outlives the premise that motivated it

Zero exposes its internals because it expects an agent to consume them. That motivation may or may not turn out to be right, and the observability is valuable either way. A student can print the phase list, time each phase, read the symbol table as 16 relations, watch the same program be refused at 3 distinct gates, and diff the object formats produced by 8 emitters from one source graph — from a shell, without patching a compiler or attaching a debugger.

None of that depends on the agent thesis being correct. It is a teaching property that fell out of an engineering decision, and it is the property we would most like to see other toolchains copy, because it is the one that costs the least to adopt. A compiler does not have to be graph-first to report its phases honestly, and §10 suggests it should think carefully about the size of the report while it does — the 2.3× premium structured diagnostics carry is a fair price; the premium on the full verdict payload is not yet costed for anyone.

15.3 The open question is distribution, and we cannot answer it

What follows is speculation, and we mark it as such because nothing in our dataset bears on it. A language designed for machine authorship faces a bootstrapping problem that has nothing to do with compilers: a model writes the languages it has seen. A language with no corpus is a language a model must be taught in-context, on every call, at a token cost that competes directly with the savings a machine-first interface is supposed to deliver. Interface quality does not obviously move that constraint, and neither does a good diagnostic schema, if the language guide has to travel in the context window alongside the program the diagnostic is about.

We can say what would change our mind, which is the most an honest outlook can offer. The measurement that matters is end-to-end: tokens spent per accepted edit, for the same task, in a language with a large pretraining corpus and a prose-oriented compiler versus a language with no corpus and a structured one. §10 supplies one half of that — the cost of the compiler's side of the loop — and says nothing at all about the other. If the structured loop wins on that measurement, the

design is vindicated on its own terms. If it does not, the observability in §15.2 is still worth keeping, and that is the conclusion this paper is actually in a position to defend.

---

16. Conclusion

The classical six-phase model survives contact with a graph-first compiler, but not in the shape the diagram suggests. Across 8 programs and 64 build combinations we find that Zero implements every classical phase while relocating the first three out of the compile path into an ingestion gate. That single change explains our measurements: lowering accounts for 100% of reported phase time; all 7 front-end error cases were contained at the gate; and 17 of 64 builds failed on programs the front end had accepted.

For a compilers course, the practical finding is that Zero makes the phases *visible* in a way mainstream toolchains do not. Whether the language succeeds on its own agent-oriented terms is a separate question, whose answer probably has more to do with training-data distribution than with compiler design. The observability is worth studying either way.

References

A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* 2nd ed., Pearson, 2007. pearson.com

K. D. Cooper, L. Torczon. *Engineering a Compiler.* 3rd ed., Morgan Kaufmann, 2022. elsevier.com

Vercel Labs. *Zero — the programming language for agents.* zerolang.ai · github.com/vercel-labs/zerolang (Apache-2.0). Version 0.3.4, build 5b3a90a.

Vercel Labs. *Graph architecture.* zerolang.ai/concepts/graph-architecture

Vercel Labs. *Compile path.* zerolang.ai/concepts/compile-path

Vercel Labs. *Semantic graph vs text.* zerolang.ai/concepts/semantic-vs-text

Vercel Labs. *Getting started.* zerolang.ai/getting-started. Version-matched language documentation retrieved via `zero skills get language --full`.

Microsoft. *.NET Compiler Platform (Roslyn).* github.com/dotnet/roslyn

N. Matsakis et al. *The rustc query system.* rustc-dev-guide.rust-lang.org/query.html

The Rust Project. *JSON output.* doc.rust-lang.org/rustc/json.html

salsa-rs. *salsa — on-demand, incrementalized computation.* github.com/salsa-rs/salsa

Vercel Labs. *wterm — a DOM-based terminal emulator with a WebAssembly core.* github.com/vercel-labs/wterm. Reviewed as a delivery option; see A.1.

Vercel. *Geist* and *Web Interface Guidelines*. vercel.com/geist · vercel.com/design/guidelines, with WCAG 2.2 AA for contrast and structure.

Appendix A. Reproduction

Everything is regenerated from one command. Source and dataset: github.com/HKTITAN/phases-of-zero (https://github.com/HKTITAN/phases-of-zero).

```
regenerate everything
```

```
curl -fsSL https://zerolang.ai/install.sh | bash
export PATH="$HOME/.zero/bin:$PATH"
zero --version          # 0.3.4 (build 5b3a90a)

npm install
npm run paper           # capture -> qr -> build -> pdf -> epub -> previews
```

A.1 Why the explorer ships precomputed data

We considered compiling Zero in the browser, following wterm (https://github.com/vercel-labs/wterm)[12], which runs a Zig-authored terminal core as WebAssembly. Zero 0.1.3 advertised `wasm32-wasi` and `wasm32-web` targets. Version 0.3.4 advertises neither: the target list contains 8 native targets and no WebAssembly target, and the compiler's own `selfHostRouting` report marks `browserCompiler` as removed. We therefore reimplemented phases one and two in TypeScript for the playground and replay the rest from the capture.

A.2 Corpus contents

p01_hello, p02_arith, p03_control, p04_shapes, p05_errors, p06_memory, p07_generics, p08_lexer under `corpus/`, and e01_lexical, e02_syntax, e03_name, e04_type, e05_mutability, e06_effect, e07_memory, e08_target, e09_lowering, e10_match under `errors/`.

Harshit Khemani, Kush Ahuja, Mohit Kumar Mishra, Kushagra Agrawal · zero.khe.money

Zero 0.3.4 · 2026-08-09