Agentic Development of an ARM64 Compiler Backend

An Experience Report on MLKit and ReML

MARTIN ELSMAN, University of Copenhagen, Denmark

Agentic programming offers a way to extend mature software systems with limited human effort. Compiler backend engineering is a demanding test: the result must integrate with an existing runtime and deliver efficient machine code.

Even with an established compiler front end, supporting a new architecture requires labour-intensive implementation, validation, and optimisation across garbage collection, foreign calls, exceptions, and dynamic loading. Successful code generation alone does not deliver a usable native compiler.

We present a native macOS ARM64 backend for MLKit and ReML, developed with Codex and GPT-6 Astra in approximately two days of human work. The result includes self-hosting native compilers, runtime integration, command-line tools, installation support, and garbage-collected REPL execution. In the final evaluation on twenty benchmarks, native ARM64 outperformed the existing X64 backend under Rosetta 2 on 19 workloads with garbage collection and 17 without it. Geometric-mean execution-time ratios were 0.799 and 0.787, with corresponding peak-memory ratios of 0.812 and 0.896. Complementary validation covered application binary interface (ABI) probes, forced collection, regression suites, and native bootstrap fixed points. We explain how milestone specifications, expert direction, and agent initiative produced this result, including regressions and repairs that shaped the final design. The experience demonstrates the delivery of a substantial compiler extension with competitive performance and extensive validation, supported by limited human effort and sustained agent execution.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Software testing and debugging*.

Additional Key Words and Phrases: Standard ML, ARM64, region inference, compiler validation, agentic programming

1 INTRODUCTION

Compiler developers can use agentic programming to delegate substantial implementation work to a language-model-driven system that inspects repositories, edits files, and runs tools. Early Codex research studied code synthesis [5]; contemporary Codex coordinates inference and execution in a continuing agent loop [2]. Extended development reports emphasise goals, feedback, and review [7, 24]. Compiler backend engineering tests whether this approach can deliver an integrated system with useful performance.

A mature compiler supplies analyses and optimisations, but extending it to a new architecture remains labour-intensive. Machine code must satisfy language semantics and runtime contracts for collection, exceptions, foreign calls, and dynamic loading. Correct examples and self-compilation are important milestones; a usable backend additionally requires broad validation and competitive generated code.

The result of our endeavour is a native macOS ARM64 backend for MLKit and ReML, developed with Codex and GPT-6 Astra in approximately two days of human work. It delivers self-hosting compilers, runtime integration, native tools, and installation support while retaining the existing X64 backend. It supports MLKit's collection and profiling variants, ReML's explicit regions, the supported parallel configurations, and dynamic loading for the REPL [15]. GC support in the REPL emerged as an agent-initiated extension during development.

The evaluation establishes quality beyond functional execution. Both ARM64 and X64 reached bootstrap fixed points; regression suites and targeted probes exercised frame layouts, root relocation,

DIKU Technical Report, 2026-09-23.
Author's address: Martin Elsman, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, mael@di.ku.dk.

foreign calls, exceptions, and loaded images. In the final twenty-workload evaluation, ARM64 was faster than X64 under Rosetta 2 on 19 workloads with GC and 17 without it. Geometric-mean time ratios were 0.799 and 0.787; peak-memory ratios were 0.812 and 0.896. Historical focused experiments document the changes incorporated in this final rerun.

The human-effort figure is the developer's retrospective estimate of active work, distinct from agent runtime and elapsed calendar time. The performance baseline includes Rosetta's translation costs, and neither the effort estimate nor the benchmark comparison establishes a universal productivity or architecture advantage. Within those limits, the delivered system provides concrete evidence that agent-assisted development can extend a mature compiler substantially with limited human effort.

This report explains how that result was achieved. The first working backend was slower on every GC workload. We follow the milestone specification through architectural redirection, optimisation, and late REPL repair, examining how expert decisions, agent initiative, and executable evidence improved the implementation. The contribution combines the delivered backend and its evaluation with an account of the interaction that made the result possible.

Section 2 introduces the inherited compiler contracts. Section 3 explains how these contracts informed the milestone specification and ABI design. Section 4 describes the validation process. Sections 5 and 6 trace the optimisation work, benchmark results, and final repairs. Section 7 reflects on human and agent initiative. Section 8 discusses related work, and Section 9 sets out the limitations and reproduction requirements. Section 10 concludes.

2 INHERITED COMPILER CONTRACTS

The native backend extends an existing compiler, so its obligations begin with the memory-management and representation decisions already made upstream. Region inference determines allocation and group reclamation [27, 28]; region-representation inference refines these decisions into bounded stack regions and dynamically growing regions [1, 26]. MLKit combines this discipline with optional copying and tag-free generational GC [18, 21]. Disabling GC leaves region reclamation intact. Enabling it requires discovering live movable pointers and restoring relocated values to every location subsequently used. Region GC-safety results [9, 11] motivate these contracts but do not prove a new emitter correct.

Double-ended bit-stealing compresses datatype representations [12]; deep argument flattening exposes components and unboxed values across compilation units [14]. The backend must distinguish tagged, packed, and unboxed values and handle separate integer and floating-point argument banks. ReML's explicit regions and effects [13] reuse these representations with its existing no-GC policy. Parallel allocation protection follows region effects [19].

Earlier work provides an optimising backend and a region-based abstract machine [16, 17]. The port reused allocation, calling-convention machinery, allocated statements, and the compilation manager, adding ARM instructions, printing, emission, tool invocation, and runtime bridges. Incremental and inter-module compilation [10] made architecture-specific caches essential: successful builds must not reuse incompatible machine code. X64 remained both reference implementation and regression target.

3 SPECIFICATION AND ABI DESIGN

Those inherited contracts had to become concrete acceptance criteria for the new target. The opening request was to review issue #223 because MLKit and ReML needed an ARM backend. The revised specification required an explicit compiler/runtime application binary interface (ABI) before emission and separated compiling the C runtime from generated-code interoperability. Table 1 records its acceptance structure; PR #224 carried implementation through the milestones.

Table 1. The initial milestone contract. Evidence is an acceptance goal, not a claim that every later implementation reran every check.

	Scope	Required evidence
1	Target and runtime	Explicit architecture selection; isolated archives; layout and allocation checks on both targets.
2	ABI and shared interfaces	Register roles, frames, roots, tail calls, and Darwin foreign-call placement; retained X64 behaviour.
3	First native executable	Native entry/exit, integer operations, a C call, Mach-O assembly and linking.
4	Language coverage	Closures, exceptions, regions, floating point, spills, large frames and stack arguments/results.
5	Runtime integration	Root relocation, GC variants, profiling, callbacks, shared libraries and REPL loading.
6	Parallelism	Protected allocation, publication, worker entry/exit, pthreads and optional Argobots.
7	Usable installation	Regression matrix, native tools, packaging, installation, and bootstrap fixed points.

Before emission began, the developer requested that ML calls pass the return address in ARM64's link register (`x30`); the callee would save it in its stack frame. The developer also requested MLKit rather than MLton for local incremental builds and GC-enabled compiler hosts. Host architecture, emitted architecture, and generated-program GC mode were distinguished. Later milestones added compiler-performance investigation, CI and packaging, code-builder cleanup, runtime measurement, and optimisation. Follow-up requests directed fresh Basis builds [20], typed instructions, assembly inspection, and memory measurement; scheduling and broader inlining were deferred.

3.1 Frames, roots, and foreign calls

The requested use of `x30` for return addresses was one part of a broader agreement between generated code and the runtime. To make calls, collection, and foreign interoperability work together, the ABI had to specify where arguments, saved state, and roots reside. ML integer arguments use `x0–x7` in closure, value, and region order; unboxed floating-point arguments use `d0–d7`. Registers `x16/x17` are scratch, `x18` is platform-reserved, `x28` holds runtime context, `x29` is the frame pointer, and `x30` carries returns. C-facing bridges preserve Darwin's callee-saved state independently of internal ML conventions.

`FrameLayout` distinguishes how a return address reaches the callee from where it is saved during execution. X64 calls place the return address on the stack. ARM64 ML calls instead place it in `x30`, and the callee saves that register in its frame header. Reserving space for this saved address does not mean the caller writes it there. For an even number F of local words, A spilled arguments, and R spilled results, the conservative frame has $F + E(A) + 2 + E(R)$ words, where $E(n) = 2\lceil n/2 \rceil$. Independently padded arguments and results preserve alignment after teardown. The header stores the previous frame pointer and return PC; collection cannot observe an activation before its saved state is established. Tail transfers preserve the original continuation and result area while moving overlapping arguments; exceptions also restore regions and profiling state.

GC initially used a dynamic return-PC index. At the developer's request, descriptors moved immediately before continuations, following X64. GC calls materialise a continuation in `x30` and use `b/br`, skipping the descriptor; Section 5.3 explains the resulting return-opcode correction. Image registration remains necessary for static data and global roots.

Foreign calls follow Apple's ARM64 ABI, including alignment, independent register banks, narrow stack arguments, and variadic placement. Emitter probes cover more cases than the source automatic-FFI interface exposes. Since the foreign-call intermediate representation has no root map, collection is deferred across foreign calls and callbacks, restoring the previous policy on return. Exceptions cannot unwind through arbitrary C frames. Deferred collection must preserve pending requests, a requirement later exposed by memory measurements.

Archives, objects, caches, and installed binaries were isolated by architecture and mode. Parallel support retained existing untagged, no-GC, non-profiling configurations; tests covered worker contexts, release/acquire publication, protected allocation, and freelists. Parallel GC and parallel image registration remained unsupported. Optional Argobots execution was reported as skipped when unavailable.

Milestones distinguished archive builds, runtime interoperability, and self-hosting. Explicit requests to continue guided implementation; later milestone numbering changed as experiments developed. Source observations were kept separate from runtime evidence.

4 VALIDATION THROUGHOUT DEVELOPMENT

The ABI specifies how generated code and the runtime should cooperate; validation had to establish that they actually did so, from individual transfers to whole programs. We first describe probes that deliberately exercise difficult interfaces, then explain what regression suites, bootstrap, and CI add to that evidence. Their different failure sensitivities made them complementary throughout development.

4.1 Complementary probes

Ordinary programs may leave unusual frame layouts and relocation paths untested. Targeted ABI tests therefore enumerated zero through seven spilled arguments and results. A production-emitter harness exercised four through seven physical returns and tail transfers with changing argument areas: source tuples can remain boxed and therefore cannot establish stack-result coverage. A million tail calls checked bounded stack use. Forty live arguments across a C call tested pressure, and a 4,200-word record forced a frame exceeding 32 KiB. Native tests also covered closures, indirect calls, mixed arguments, NaNs, overflow, references, exceptions, and explicit regions. They checked output, architecture, cache isolation, and rejection of unsupported modes.

Forced entry collection exercised forty live list roots, multiword bitmaps, spills, handler roots, updated references, static data, and per-image globals. A C walker probe checked relocation of register, stack, and global slots and rejected reserved-register masks. Forced profiling and an AddressSanitizer run supplemented a shared profiler use-after-free regression. REPL tests retained values across loaded images and collections and continued after an uncaught exception.

Optimisations added adversarial helpers that clobbered all C-volatile ML registers, checked alignment, and wrote through region descriptors. Branch tests covered inline data, alignment, section changes, and unknown directives; peepholes included rejection cases. Nested C-to-ML-to-C-to-ML callbacks checked that pending collection survived disabled intervals and ran afterwards. A disabled-collector call with a null root image checked that invalid roots were not inspected. Such probes test contracts that ordinary helpers and output comparisons can accidentally leave unexercised.

Foreign-call probes checked stack arguments, converted integers and booleans, exported captured closures, and Darwin callee-saved registers. The distinction between source and emitter coverage mattered: testing a mixed-argument assembly interface did not establish that the automatic FFI exposed that signature to source programs. Likewise, testing relocation required traversing retained structures after collection, not merely reaching the collector and returning. Exact output checks

Table 2. Complementary quality evidence. Counts are milestone 7 results; the last row is validation of the final repair. They are not a single test run of the final implementation.

Boundary	Recorded result
Language regressions	130/130 developer tests on each target; 180/180 native tests in each of six modes, then 181/181 GC tests after adding a regression.
Explicit regions and threads	79/79 ReML cases; 13/13 each in dedicated pthread and configured Argobots suites.
Self-hosting	Native ARM64 and X64 stripped stage-two/stage-three fixed points.
Runtime contracts	Forced-GC relocation, foreign-register clobbering, large frames, stack results, exceptions, profiling, and multi-image roots.
Final REPL repair	Eight full-Basis tests in each of no-GC and GC mode with each of two compiler hosts; native integration also passed.

were useful but not exhaustive: some benchmark programs, notably `msort`, print progress rather than a checksum of the complete result.

4.2 Regression, bootstrap, and diagnostics

The probes isolated difficult contracts; regression suites and bootstrap tested their composition in larger programs. Table 2 separates milestone 7 results from final-repair checks. The six native modes were no-GC, GC, generational GC, no-GC profiling, GC profiling, and pthreads. Three-stage bootstraps with fresh caches checked architecture and execution and reached byte-identical stripped stage-two/stage-three binaries on both targets. X64 used the classic Darwin linker to avoid nondeterministic GOT ordering; ARM comparison copies retained matching basenames because stripping affects signatures. These fixed points support build consistency, not universal compiler correctness.

Later changes reran focused suites rather than the complete milestone matrix. The final repair passed full-Basis GC/no-GC REPL checks with both an X64-hosted emitter and an ARM-native compiler, plus native integration. Historical suite results therefore remain distinct from final-implementation evidence.

CI added architecture checks, separate artifacts, tools, regressions, bootstrap, and installation. Diagnostic integrity itself required repair: a self-referential filesystem-test symlink broke artifact collection and lost hosted REPL logs. Collecting regular files without following links fixed staging; the REPL harness retained raw output and checked compiler status before filtering. Local reproduction established a defect in the failing path, but missing hosted logs prevent confirmation of the exact hosted crash.

5 FROM WORKING CODE TO EFFICIENT CODE

With functional coverage and self-hosting established, the next question was efficiency. Performance work concerned two products: the compiler itself and the programs it generated. We follow the shift from expensive instruction-list construction to runtime bottlenecks, showing how measurements and human redirection selected successive experiments. Several plausible structural improvements initially failed to improve execution time.

5.1 Compiler construction and instruction representation

Compiler timings first suggested that constructing ARM instructions was itself expensive. Early nineteen-program reports showed compiler user CPU of 30.10 s for native ARM versus 14.87 s

for X64/Rosetta, with code generation accounting for 11.08 s versus 1.16 s. These single-run observations identified a suspicious phase. The developer noticed repeated list concatenation and requested suffix-passing builders, initially measuring only `nucleic`. Ordered static-data chunks and builders that prepend to a supplied suffix reduced its code-generation CPU from 1.733 s to 0.556 s. Completing suffix passing changed five-run medians only from 0.546 s to 0.536 s; the developer retained the uniform design. A local composition operator, `fun (f ++ g) x = f(g x)`, preserved readable execution order.

Inlining ordinary allocation and simple resets, with shared preserving slow paths, reduced emitted instructions from 243,120 to 102,765 and seven-run median user CPU from 29.097 ms to 14.671 ms on the small `nucleic` test. This microbenchmark differs from the paper workload of the same name. Profiling and protected allocation retained their bookkeeping.

The developer then challenged string-based instructions and optimiser matching, requesting a datatype like X64's and a separate printer. Fixed-arity constructors such as `b_ne` made operands and addressing variants explicit. Codex migrated representation, emission, printing, and tests; normalised assembly was unchanged. Code-generation time fell from 0.098 s to 0.058 s, but whole compilation only from 3.839 s to 3.711 s because formatting partly moved outside the phase timer. Interaction improved maintainability while measurement bounded the performance claim.

Builder validation compared instruction structure when label identities changed. The later composition cleanup produced byte-identical assembly for `nucleic` and all 26 emitter files. Typed instructions preserved normalised assembly, but legality, immediate ranges, aliasing, and optimisation conditions still required checks.

5.2 An unfavourable baseline and successive hypotheses

Cheaper instruction construction did not establish efficient generated programs. The initial runtime evaluation therefore used twenty workloads from `debs-icfp24` [12] and `mlkit-bench` on an M2 Max, with one warmup and five measured runs per target/mode. All 480 executions passed output checks, yet ARM/X64 geometric-mean time ratios were 2.01 with GC and 1.52 without it; every GC workload was slower. Functional completion redirected attention towards explaining costs.

Helper-preservation reductions, register-home removal, and peepholes made `professor` 11–13% faster but `mlyacc` 5–8% slower. The developer retained inline descriptors despite initial slowdowns of 2.2% for `nucleic`, 18.4% for `mlyacc`, and 11.8% for `professor`. Jump tables barely changed measured times. Structural improvements remained performance hypotheses.

Comparison with X64 exposed unnecessary scratch-register and stack staging. Direct operand lowering removed traffic at calls, arithmetic, records, and switches, retaining conservative aliasing fallbacks. Bounded-distance branches stayed short; unknown directives remained barriers. GC entry used shared snapshot stubs. Compile-time physical-register liveness reduced region-helper preservation to C-clobbered live registers, with conservative handler treatment. Preservation sets remained distinct from pointer-root maps. A focused GC experiment improved `professor` by 16.2%, but `mlyacc` by only 2.0%. Identity wrappers and eligible self-tail frames were subsequently removed.

Static-pointer classification was another GC cost. Sealing the initial executable's data ranges into an enclosing interval enabled bounds checks, while separately loaded images retained exact ranges. Combined with allocation-aware polling and simpler constructor tests, this reduced `nucleic` elapsed time from 1.488 s to 0.814 s and collector CPU from 0.8273 s to 0.1570 s, with 3,106 collections unchanged. The report attributed most of the gain to classification without isolating each change's contribution.

Table 3. GC return-convention experiment: median elapsed seconds, five measured runs. X64 executes under Rosetta 2.

Program	X64	ARM before	ARM after	Reduction
`nucleic`	0.9663	0.7240	0.6441	11.0%
`mlyacc`	0.1903	0.2403	0.1468	38.9%
`professor`	0.2333	0.2485	0.1911	23.1%

5.3 Matching call and return conventions

The modest benefit for `mlyacc` left its call overhead unexplained. Further investigation exposed a mismatch: GC calls used explicit continuations and `b/br`, but epilogues still used `ret`, without a matching hardware call to update return prediction state. GC ML returns changed to `br x30`, including callback entry conventions; no-GC retained `bl/blr` with `ret`. Native helpers and C bridges kept ordinary returns.

The experiment rebuilt all Basis and benchmark units with fresh caches, keeping the ARM runtime fixed. One warmup and five shuffled measured runs produced 54 correct outputs and unchanged collection counts (Table 3). Assembly inspection confirmed replacement of 8,162 ML return sites. The controlled opcode change supports the transfer-mismatch diagnosis, without directly measuring proprietary predictor internals or changing allocation, scheduling, or inlining.

6 BROAD EVALUATION AND FINAL OPTIMISATIONS

Focused experiments explained individual costs, but a usable backend also needed evidence across diverse workloads. We first present one consistent final timing and memory evaluation, then explain the earlier outlier investigations and repairs that contributed to it. The historical experiments establish how the implementation evolved.

6.1 Twenty-program measurements

To assess the completed implementation, Tables 4 and 5 report all twenty workloads rerun for both architectures and GC modes from one clean snapshot, including the final optimisations and REPL repairs. The machine was an Apple M2 Max with 32 GiB RAM and macOS 26.5.1. Both target compilers were rebuilt with MLKit and GC enabled; compiler execution was outside timing. Both runtime variants and all Basis libraries were rebuilt with separate, initially empty target/mode caches. All eighty executables were built and architecture-checked before measurement. Each received one warmup and seven measured executions, with deterministically shuffled configuration order. All 640 executions exited successfully and matched expected stdout byte for byte.

ARM64 was faster on 19/20 GC workloads and 17/20 no-GC workloads, with geometric-mean time ratios 0.799 and 0.787. Peak-memory ratios were 0.812 and 0.896, with lower ARM memory on 19 and 15 workloads. Memory is the median of seven child-process peak RSS values from `wait4`, including code, stack, heap, and process-associated Rosetta memory. Timing includes startup and I/O; GC runs use `-report_gc`. These are complete execution configurations, not bare instruction-set comparisons. The no-GC configuration uses a non-tagged representation of values, supported by MLKit's compilation of polymorphic equality [8]. Thus GC/no-GC differences include representation and settings, not just collector cost.

The remaining timing exceptions were `mpuz` with GC (ratio 1.03), and `kbc`, `mpuz`, and `zebra` without GC (1.01, 1.10, and 1.04). No-GC `DLX` RSS varied substantially on X64: 62.3–199.0 MiB

Table 4. Final twenty-program execution times (seconds), medians of seven measured runs after one warmup. X64 runs under Rosetta 2; ARM64 runs natively. Ratios are ARM64/X64. All configurations use one final source snapshot.

	GC evaluation			No-GC evaluation		
Program	X64	ARM64	Ratio	X64	ARM64	Ratio
barnes-hut	0.8590	0.8389	0.98	1.4406	1.0365	0.72
calc	1.9381	1.1920	0.62	1.0704	0.8073	0.75
DLX	0.3852	0.3523	0.91	0.1959	0.1343	0.69
fft	0.3626	0.2881	0.79	0.2593	0.2319	0.89
kbc	0.7291	0.6487	0.89	0.5976	0.6053	1.01
lexgen	0.4153	0.3247	0.78	0.3570	0.2718	0.76
life	0.7321	0.6214	0.85	0.4677	0.4599	0.98
logic	1.0589	0.7832	0.74	0.6572	0.4550	0.69
mandelbrot	0.2972	0.1924	0.65	0.2955	0.1924	0.65
mlyacc	0.1879	0.1428	0.76	0.1870	0.1358	0.73
mpuz	0.3834	0.3966	1.03	0.2641	0.2906	1.10
msort	0.5254	0.4300	0.82	0.3482	0.2640	0.76
nucleic	0.9644	0.6423	0.67	1.0005	0.5284	0.53
patricia	3.7086	3.2083	0.87	2.0813	1.5400	0.74
professor	0.2383	0.1974	0.83	0.2049	0.1765	0.86
ray	0.3597	0.2823	0.78	0.3300	0.2388	0.72
simple	0.5664	0.4283	0.76	0.4381	0.4198	0.96
uf	0.1275	0.1002	0.79	0.1008	0.0798	0.79
vliw	0.4732	0.3189	0.67	0.4350	0.2753	0.63
zebra	0.7412	0.7038	0.95	0.5397	0.5589	1.04
Geometric mean ratio			0.799			0.787

across seven runs, compared with 66.7 MiB throughout on ARM64. The tables report medians, not precise bounds on future runs.

Earlier outlier investigations explain changes included in these final results. Their before/after experiments retain their original baselines; no historical samples enter the final tables.

6.2 Arithmetic, deferred collection, and spill traffic

The outliers narrowed the next investigations to arithmetic overhead, collection policy, and frame traffic. In `uf`, audited word-modulo helpers neither allocated nor called back into ML, but paid general foreign-call protection and staging costs. Direct calls and removal of redundant tagged-word normalisation improved focused `uf` time from 0.1788 s to 0.1066 s, preserving division-by-zero and wrapping semantics. Arbitrary foreign calls retained protection.

`DLX` exposed a shared runtime defect: thresholds reached while GC was disabled lost pending requests. Recording requests regardless of disabling, and checking disable state before inspecting roots, restored collection on both targets. Focused RSS fell from 116.0 to 26.8 MiB while time rose from 0.2541 to 0.3632 s; collections rose from eight to 683, matching updated X64. The slower result was the desired repair. For `msort`, higher RSS despite faster execution suggested frame-size work; branch-exclusive spill-slot sharing remained unimplemented.

Final work extended audited direct calls to untagged modulo, inhibited load pairing when a second destination overwrote the base, and reused eligible no-GC recursive frames despite spills and nested calls. Private spills were delayed while register copies or field projections sufficed;

Table 5. Final peak resident memory (MiB), medians of seven per-process peaks from the same campaign as Table 4. Ratios are ARM64/X64. Code, stack, heap, and process-associated Rosetta memory are included; these are not ML-heap-only measurements.

	GC evaluation			No-GC evaluation		
Program	X64	ARM64	Ratio	X64	ARM64	Ratio
barnes-hut	5.5	4.4	0.80	635.2	636.3	1.00
calc	86.6	66.8	0.77	321.4	324.6	1.01
DLX	25.1	23.9	0.95	106.7	66.7	0.63
fft	82.2	79.2	0.96	55.8	52.4	0.94
kbc	10.9	9.6	0.88	8.3	7.1	0.85
lexgen	14.6	12.9	0.89	66.6	66.0	0.99
life	4.4	3.0	0.67	30.7	29.5	0.96
logic	5.2	3.5	0.67	815.2	816.6	1.00
mandelbrot	4.1	2.9	0.71	3.9	2.8	0.70
mlyacc	18.0	15.2	0.85	111.0	108.6	0.98
mpuz	4.1	2.9	0.71	4.0	2.8	0.69
msort	132.4	146.7	1.11	365.9	381.0	1.04
nucleic	6.6	4.8	0.72	1104.2	1106.6	1.00
patricia	6.8	5.4	0.79	8.9	7.6	0.86
professor	5.0	3.8	0.75	12.0	10.6	0.89
ray	14.5	13.0	0.90	17.8	17.5	0.98
simple	7.1	5.7	0.80	6.1	4.6	0.76
uf	8.8	7.6	0.86	7.1	5.9	0.83
vliw	18.5	16.8	0.91	171.8	171.0	0.99
zebra	4.4	3.0	0.67	119.6	118.9	0.99
Geometric mean ratio			0.812			0.896

Table 6. Historical focused optimisation experiment: median seconds over seven measured runs after one warmup. All Basis caches were rebuilt.

Mode	Program	ARM before	ARM after	X64/Rosetta	Reduction
No GC	simple	0.5579	0.4610	0.4714	17.4%
No GC	mpuz	0.3871	0.3078	0.2782	20.5%
No GC	uf	0.1574	0.0848	0.1126	46.1%
GC	simple	0.5513	0.4699	0.6216	14.8%
GC	mpuz	0.4248	0.4113	0.4244	3.2%
GC	uf	0.1099	0.1076	0.1427	2.1%

bounded continuation duplication moved stores off short branches. Limits were eight pending assignments, 32 duplicated branches, and four continuation statements. Calls, mutation, allocation, and unsupported operations forced commitment; GC loops retained stricter eligibility.

Table 6 records the development-stage comparison with fresh Basis caches. No-GC gains were substantial, although `mpuz` remained 10.6% slower than X64. The 2–3% GC changes are too small for precise attribution from one short session; peak-memory differences within 80 KiB did not establish changed heap requirements. Tests covered spill-heavy loops, nested calls, references, exceptions, overflow, and modulo errors. The final full rerun in Tables 4 and 5 includes these changes.

6.3 A late dynamic-linking failure

The optimised helpers also had to work through dynamic linking, which the static benchmarks did not exercise. Full-Basis REPL testing exposed private helpers receiving a region in `x16` and size in `x17` through a Mach-O stub that overwrote `x16` with the helper address. The helper consequently wrote into executable text. Static linking usually bypassed the stub, hiding the defect. Loading the helper address into `x30` and using `blr x30`, after saving the original return, repaired the transfer. The ABI had already identified `x16/x17` as linker scratch: its contract needed enforcement at the actual boundary.

Static bootstrap and dynamic loading cross different ABI boundaries despite sharing the emitter and runtime. The repair therefore required full-Basis REPL checks with two compiler hosts and both collection policies; successful static self-compilation could not substitute for them.

7 INTERACTION, INITIATIVE, AND LESSONS

The optimisations and late repair show that implementation progress depended on decisions about both design and evidence. We now examine how human intervention and agent initiative shaped those decisions. Passing the return address in `x30`, with the callee saving it in its frame, avoided inheriting X64's stack-based call convention; inline descriptors later exposed the need for matching return instructions. The instruction datatype addressed maintainability before a failing program demanded it. Uniform suffix builders survived a modest incremental gain, while subsequent allocation fast paths reduced code substantially. These connected changes cannot be treated as independent contributions.

The developer requested GC-enabled hosts, phase timings, fresh Basis builds, and twenty-program time/memory reports; Codex investigated hypotheses and implemented bounded changes. Basis reuse isolates benchmark-unit changes, whereas rebuilding measures whole-program effects. Memory evidence changed acceptance: restored `DLX` collection mattered more than its earlier attractive time. Scheduling, broad inlining, and spill-slot sharing were deferred to bound scope.

GC support in the REPL was a byproduct selected and implemented by the agent without interaction with the human codesigner. We interpret it as agent initiative within mixed-initiative development, related to established human–agent collaboration [22] and proactive programming assistance [6]. Proactivity concerns identifying useful work without an explicit request; autonomy concerns executing work independently, a distinction also made in a recent coding-agent position paper [3]. Here the agent implemented an additional capability, beyond suggesting one. The later human question prompted validation and repair of existing support. Initiative broadened functionality; human-directed evaluation exposed its dynamic-loading defect.

These episodes distinguish proposed improvements from established value. Phase timing overstated the instruction datatype's compilation benefit; inline descriptors initially regressed runtime; restored collection deliberately slowed `DLX`. Retaining such outcomes and their baselines makes the collaboration assessable beyond its successful speedups.

Persistent reports recorded hosts, caches, inputs, stages, and samples, but required reconciliation: an early overview described GC returns as `ret`, while later implementation required `br x30`. Reports and tests enabled continued expert direction; they do not establish that the initial specification alone would have produced the same result.

8 RELATED WORK

The MLKit papers cited earlier explain the inherited analyses and runtime contracts. Here we relate the development process to research and engineering reports on coding agents, performance-directed programming, and compiler construction.

Original Codex research [5] evaluates function synthesis through HumanEval; our use concerns a tool-using agent. Bolin [2] describes that agent loop. Choi [7] reports roughly 25 hours of development guided by persistent specifications and validation; Lopopolo [24] emphasises repository knowledge and enforced constraints. These engineering articles contextualise our practices but are not controlled productivity studies.

ParaCodex [23] combines a Codex-based agent with hotspot analysis, data planning, compilation, testing, and profiling for OpenMP GPU offload. It reports valid results on 31 eligible kernels and faster GPU execution on 27. The methodological connection is measured performance after functional validation. Our task extends a compiler backend and its runtime contracts, with the human repeatedly choosing which bottleneck to investigate.

Carlini [4] reports sixteen Claude agents building a C compiler from scratch, emphasising testing, reference-compiler comparisons, and remaining performance limits. Our port retains a mature compiler's analyses and regression baseline. Both experiences distinguish compiling substantial programs from delivering consistently good generated code.

Paraskevopoulou [25] reports Claude-assisted CertiCoq proof development guided by an existing CPS proof and human review. The report describes roughly 96 elapsed hours and stresses scrutiny of the proof statements themselves. Its terminating-program theorem is machine-checked; our backend evidence is empirical. Neither its elapsed time nor our human-effort estimate supports a cross-project productivity comparison.

9 LIMITATIONS AND REPRODUCTION

Our results establish a concrete development experience, with limits on both causal attribution and generalisation. This is one expert developer, compiler family, and model session. The two-day estimate concerns retrospective human effort, without an unaided baseline; domain knowledge was a substantial resource. Tests and bootstrap cannot exhaust compiler/runtime semantics.

Measurements use one machine, short runs, and a Rosetta baseline. Focused sample counts and Basis policies varied, so stage-local improvements cannot be accumulated. The final benchmark matrix covers one source snapshot, but does not replace the historical regression suites or establish hosted CI completion. Small timing differences remain uncertain even when their output checks pass.

Unsupported cases include colon-based foreign-symbol resolution, exception unwinding through arbitrary C frames, parallel GC, and parallel image registration. Long callbacks can retain allocations. General scheduling, broad inlining, and spill-slot sharing remain opportunities rather than delivered functionality.

The companion directory contains the milestone issue, source map, and final scripts, manifests, and raw samples in `evaluation/final20`. The compiler guide records checks; the return and focused reports document earlier experiments. Reproduction requires Apple Silicon, adapted paths, stated inputs, and fresh caches. The final campaign was rerun for this paper; historical comparisons retain their original measurements.

10 CONCLUSION

Approximately two days of human work produced native compilers, runtime integration, bootstrap evidence, and competitive measured performance. Expert redirection, agent initiative, and complementary validation were all consequential. Failures crossed boundaries between frames and returns, roots and regions, deferred collection and pending requests, and calls and linker stubs. Making those boundaries explicit and testing them turned executable code into a usable backend; correctness and interpretation remained engineering responsibilities.

REFERENCES

[1] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From Region Inference to von Neumann Machines via Region Representation Inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 171–183. https://doi.org/10.1145/237721.237771

[2] Michael Bolin. 2026. Unrolling the Codex Agent Loop. OpenAI Engineering, January 23. https://openai.com/index/unrolling-the-codex-agent-loop/

[3] Nghi D. Q. Bui and Georgios Evangelopoulos. 2026. Agentic Coding Needs Proactivity, Not Just Autonomy. Position paper, arXiv:2605.06717. https://arxiv.org/abs/2605.06717

[4] Nicholas Carlini. 2026. Building a C Compiler with a Team of Parallel Claudes. Anthropic Engineering, February 5. https://www.anthropic.com/engineering/building-c-compiler

[5] Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374. https://doi.org/10.48550/arXiv.2107.03374

[6] Valerie Chen, Alan Zhu, Sebastian Zhao, Hussein Mozannar, David Sontag, and Ameet Talwalkar. 2025. Need Help? Designing Proactive AI Assistants for Programming. CHI 2025; arXiv:2410.04596, revised February 28. https://arxiv.org/abs/2410.04596

[7] Derrick Choi. 2026. Run Long Horizon Tasks with Codex. OpenAI Developers, February 23. https://developers.openai.com/blog/run-long-horizon-tasks-with-codex

[8] Martin Elsman. 1998. Polymorphic Equality - No Tags Required. In *Proceedings of the Second International Workshop on Types in Compilation (TIC '98)*. Springer-Verlag, Berlin, Heidelberg, 136–155.

[9] Martin Elsman. 2003. Garbage Collection Safety for Region-Based Memory Management. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (New Orleans, Louisiana, USA) *(TLDI '03)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/604174.604190

[10] Martin Elsman. 2008. *A Framework for Cut-Off Incremental Recompilation and Inter-Module Optimization.* Technical Report. IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark.

[11] Martin Elsman. 2023. Garbage-Collection Safety for Region-Based Type-Polymorphic Programs. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 115 (June 2023), 23 pages. https://doi.org/10.1145/3591229

[12] Martin Elsman. 2024. Double-Ended Bit-Stealing for Algebraic Data Types. *Proceedings of the ACM on Programming Languages* 8, ICFP, Article 239 (Aug. 2024), 33 pages. https://doi.org/10.1145/3674628

[13] Martin Elsman. 2024. Explicit Effects and Effect Constraints in ReML. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 79 (Jan. 2024), 25 pages. https://doi.org/10.1145/3632921

[14] Martin Elsman. 2025. Compositional Deep Argument Flattening. ML Family Workshop. Extended abstract, 18 pages. https://elsman.com/pdf/ml25-deep-flattening.pdf

[15] Martin Elsman. 2026. *Crafting a REPL for HOT Compiled Execution.* DIKU Technical Report. Department of Computer Science, University of Copenhagen. September 19.

[16] Martin Elsman and Niels Hallenberg. 1995. An Optimizing Backend for the ML Kit Using a Stack of Regions. Student Project 95-7-8, University of Copenhagen (DIKU).

[17] Martin Elsman and Niels Hallenberg. 2002. *A Region-Based Abstract Machine for the ML Kit.* Technical Report TR-2002-18. Royal Veterinary and Agricultural University of Denmark and IT University of Copenhagen. IT University Technical Report Series.

[18] Martin Elsman and Niels Hallenberg. 2021. Integrating region memory management and tag-free generational garbage collection. *Journal of Functional Programming* 31 (2021), e4. https://doi.org/10.1017/S0956796821000010

[19] Martin Elsman and Troels Henriksen. 2023. Parallelism in a Region-Inference Context. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 142 (June 2023), 23 pages. https://doi.org/10.1145/3591256

[20] Emden R. Gansner and John H. Reppy (Eds.). 2004. *The Standard ML Basis Library.* Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9780511546846

[21] Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining Region Inference and Garbage Collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/512529.512547

[22] Eric Horvitz. 1999. Principles of Mixed-Initiative User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Pittsburgh, Pennsylvania, USA, 159–166. https://www.erichorvitz.com/uiact.htm

[23] Erel Kaplan, Tomer Bitan, Lian Ghrayeb, Le Chen, Tom Yotam, Niranjan Hasabnis, and Gal Oren. 2026. ParaCodex: A Profiling-Guided Autonomous Coding Agent for Reliable Parallel Code Generation and Translation. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, San Diego, California, United States, 16113–16136. https://doi.org/10.18653/v1/2026.acl-long.732

[24] Ryan Lopopolo. 2026. Harness Engineering: Leveraging Codex in an Agent-First World. OpenAI Engineering, February 11. https://openai.com/index/harness-engineering/

[25] Zoe Paraskevopoulou. 2026. Machine-Generated, Machine-Checked Proofs for a Verified Compiler (Experience Report). arXiv preprint arXiv:2602.20082. https://doi.org/10.48550/arXiv.2602.20082

[26] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (Sept. 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

[27] Mads Tofte and Jean-Pierre Talpin. 1994. Implementing the Call-By-Value Lambda-Calculus using a Stack of Regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 188–201.

[28] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613