# Keleusma's Self-Hosting Strategy

Brendan Sechter

2026-07-12

The V0.3.0 goal of Keleusma is a Keleusma compiler written in Keleusma source, compiled to Keleusma bytecode, running on the Keleusma virtual machine, producing Keleusma bytecode as output. The endpoint is a fixed point. The self-hosted compiler compiled by the Rust-hosted compiler produces bytecode identical to what the Rust-hosted compiler produces from the same source. The self-hosted compiler compiled by itself reproduces its own bytecode. This article describes the strategy that will reach that fixed point.

The strategy has two parts. The first is an incremental migration method that ports the existing Rust-hosted compiler to Keleusma one stage at a time without a big-bang rewrite and without giving up a working reference at any point. The method is not Keleusma-specific. It applies to any staged pipeline whose implementation language is being changed. The second is a three-stage stream-processor pipeline architecture that matches Per Brinch Hansen's pipeline-of-processes model and matches Keleusma's coroutine semantics directly.

The strategy was informed by the stream-based compilers series, which covers the historical demonstrations and the mathematical foundation that the Keleusma design draws on, and by the self-hosted silicon compiler article, which develops the software-to-silicon boundary that a self-hosted software compiler sits above. The article on developments in programming language theory as a historical arc places the self-hosting concept in the broader seventy-year arc.

## What Self-Hosting Means and Why It Matters

Self-hosting a language is the most credible demonstration that the language is expressive enough to write its own toolchain. The signal is twofold. First, it validates the surface language and the type system against a concrete, complex program of substantial size. Second, it removes a dependency. A self-hosted Keleusma can evolve without forcing every change through the Rust-hosted compiler maintainers. Teams that value a short auditable toolchain dependency graph benefit in particular.

For Keleusma specifically, the self-hosted compiler is a precondition for the V0.4.0 native code generation goal, which compiles the self-hosted compiler to native code via LLVM and links it as a static library against a Rust host. Without the V0.3.0 self-hosted compiler, V0.4.0 has nothing to compile to native code.

The self-hosting concept was treated at length for the software case in the streaming compilers series conclusion, which discusses the coalgebraic fixed-point condition that a self-hosted compiler satisfies. It was treated for the silicon case in the recent article on self-hosted silicon compilation. Keleusma sits on the software side of that boundary and offers a candidate example of a compact-toolchain language design that a self-hosting compiler could reasonably compile itself with.

## The Backward Incremental Migration Method

The migration method is a backward variant of what Martin Fowler calls the strangler pattern, specialized for compilers by porting the emit boundary first, because that is where self-hosting efforts most often stall. The method applies when four preconditions hold.

First, a working reference implementation exists. The existing compiler keeps running throughout and serves as the equivalence oracle that every intermediate state is checked against. Without a reference there is nothing to validate against, and the method does not apply.

Second, the pipeline has clean stage boundaries. The compiler is a sequence of stages, for example tokenize, parse, analyze, generate, and emit, each consuming the previous stage's output. Those boundaries are where the seam moves.

Third, the boundaries can be bridged. The migrating engineer can convert one stage's output in the host language into the input the next stage expects in the target language. This is what the adapters do.

Fourth, the backend is the real risk. Producing a valid target artifact at the emit boundary is the hardest and least certain part. This is what justifies going backward.

### Going Backward, Last Stage First

Port the stages in reverse pipeline order. The last stage, the one that emits the target artifact, is ported first. The first stage, usually the lexer, is ported last.

The reason is risk, not convenience. Frontends are well understood and low risk. The wall is the backend. Self-hosting efforts commonly reach a state where the new compiler can read its own source but cannot yet emit a valid artifact, and they stall there. Porting the emit boundary first retires that risk before the engineer invests in the easy stages.

### The Single Moving Adapter

At any moment exactly one adapter sits at the frontier between the still-host-language upstream and the already-target-language downstream. The upstream stages run unchanged and produce their output as before. The adapter converts that output into the input the first target-language stage expects. The downstream stages, already ported, chain directly to one another.

Each time the engineer ports the next stage backward, that stage takes over the adapter's job and the adapter disappears, and a new adapter appears one position further upstream. The seam moves through the pipeline from back to front, and there is never more than one adapter to maintain.

The seam is easiest to see as a picture. The pipeline runs left to right, `H` marks a stage still in the host language, `T` marks a ported target-language stage, and the bar `|` is the single adapter at the frontier.

```
start    H  H  H  H  H      all host, no adapter yet
step 1   H  H  H  H | T     emit stage ported, adapter in front of it
step 2   H  H  H | T  T
step 3   H  H | T  T  T
step 4   H | T  T  T  T
done     T  T  T  T  T      fully self-hosted, adapter gone
```

The two endpoints correspond to the two whole-compiler tombstone diagrams of a bootstrap, the compiler written in the host language at the start and the compiler written in the target language at completion, and the seam is the path between them.

### Adapters as Throwaway Prototypes

The adapter's output and its implementation are separate concerns. The output is the data contract the two adjacent stages agree on, and it is permanent. The implementation, the conversion from the host representation to the target one, is disposable. It is a prototype of the output behavior that the not-yet-written upstream stage will eventually produce for real, and writing that upstream stage is precisely what retires the adapter.

This framing sets the right quality bar. An adapter needs to be good enough to let the engineer build and validate the stage below it, not perfect, not general, and not efficient. When an adapter is getting expensive to fix, that is often the signal to stop and write the real upstream stage that will replace it. The goal is stages, not adapters.

### The Deferral Ledger

Correctness during migration is a judgment call, and the ledger is what keeps it honest. For each corpus program, either the current chain processes it correctly, or its deviation is recorded in a ledger entry that names the upstream stage that will correct it. An entry records three things, the corpus program, the observed deviation from the reference, and the responsible upstream stage.

When that upstream stage lands, the deferred cases are re-run and each is confirmed resolved. A deviation that its responsible stage does not fix was never an adapter limitation. It is a real bug in the stage that already exists, and the ledger is what surfaces it. Without the ledger and the recheck, the judgment call becomes a place for stage bugs to hide until the end.

### The Completion Gate

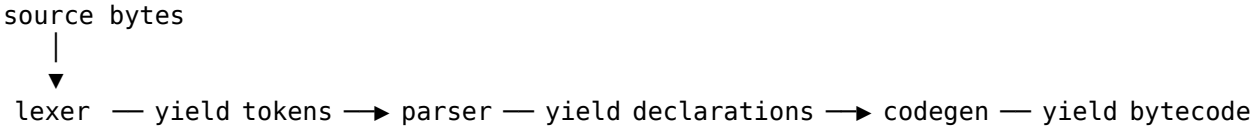
Two engineering modes apply to the process. During migration the engineer defers. The engineer prefers building the next stage to perfecting a throwaway adapter, and records every deferral in the ledger. Once all stages are ported and the adapters are gone, the engineer switches from deferred work to bug fixing. The engineer drives the ledger to empty, and the fully self-hosted pipeline must process the corpus correctly with no adapter left to defer to.

An empty ledger and a passing corpus under the fully self-hosted pipeline is the completion gate for the migration. Every intermediate deferral is a debt against it.

A further gate belongs at the end and is to self-hosting rather than to migration in general. The engineer compiles the compiler's own source with the reference and with the self-hosted compiler and confirms the two artifacts agree, then compiles the compiler with itself and confirms the output is stable across successive generations. This is the staged-bootstrap fixed-point check, the same reproducibility comparison that a multi-stage bootstrap makes between its later stages. The compiler's own source is the most demanding program it will process, and the fixed point is where a subtle miscompilation that the corpus missed will surface.

## The Three-Stage Pipeline Architecture

The V0.3.0 compiler is decomposed into three coordinated stages, each a Keleusma `loop` function.



The lexer consumes source bytes and yields tokens. The parser consumes tokens and yields parsed declarations, where the unit of work is a single top-level declaration rather than the whole program. The codegen stage consumes parsed declarations and yields bytecode chunks plus the auxiliary body that the wire format expects.

The decomposition matches Brinch Hansen's pipeline-of-processes model that the stream-based compilers series covers in its Brinch Hansen article, and it matches Keleusma's coroutine semantics directly. Each stage's working memory is bounded by its local state plus the inter-stage buffer, both of which fit inside the per-Stream-to-Reset arena budget. The whole-program abstract syntax tree is never constructed. The parser yields each declaration as it completes. The codegen stage emits bytecode immediately and forgets the declaration. Symbol tables are per-scope and popped on scope exit.

The host application that drives the pipeline is a Rust program during the V0.3.0 line and a Keleusma program once V0.4.0 lands. The host resumes each stage as needed and handles the inter-stage flow control. The host's responsibilities are minimal. It collects tokens emitted by the lexer, hands them to the parser, collects declarations emitted by the parser, hands them to the codegen stage, collects bytecode emitted by the codegen stage, and assembles the wire-format buffer.

Three reasons recommend this shape.

First, it composes cleanly with Keleusma's existing model. Each stage is a `loop` function, which Keleusma already admits. The yield-and-resume protocol is the inter-stage communication channel. No new language primitives are required. The bounded worst-case-memory-usage guarantee falls out for each stage independently.

Second, it matches the demonstrated prior-art model. Brinch Hansen's compilers were written exactly this way and worked. The pattern is not speculative.

Third, it provides natural test points. Each stage can be tested in isolation by driving it with a synthesized input stream and inspecting the output stream. The Rust-hosted compiler already has a per-stage test surface that the self-hosted version can reuse with minor adaptation.

## The Integrated Single-Pass Alternative

The Wirth tradition produced compilers that did not decompose into stages at all. The Turbo Pascal compiler, the Oberon compiler, and the various Modula-2 compilers ran the entire compile pipeline as a single recursive-descent parser that emitted bytecode or machine code directly during parsing. There was no token stream materialized between the lexer and the parser. The lexer was just a method on the parser that returned the next token on demand. There was no abstract syntax tree. Each syntactic construct emitted its corresponding bytecode at the point in parsing where the construct was recognized.

This is the integrated single-pass alternative that the stream-based compilers series covers across its Wirth-line and Turbo Pascal articles. Its appeal is speed. The Turbo Pascal benchmark of ten thousand to thirty thousand lines per second on a four point seven seven megahertz eight-oh-eight-eight in nineteen eighty-four, and the Oberon compiler at millions of lines per second on modern hardware, are the gold standards. The architecture has no inter-stage buffering and no stage-coordination overhead.

The V0.3.0 strategy documents this alternative but does not recommend it. The reason is that Keleusma's coroutine model rewards the decomposed pipeline shape. Each `loop` function is a natural stage, and the bounded-worst-case-memory-usage guarantee falls out per stage. An integrated single-pass compiler in Keleusma would either be a single very-long `loop` function, which the verifier might admit but which is awkward to test in isolation, or a function-call chain that cannot use recursion, which forces the explicit-stack discipline anyway. Neither

shape is obviously better than the pipeline. Brinch Hansen's pipeline-of-processes maps directly to Keleusma's coroutine pipeline. The integrated single-pass maps to Keleusma awkwardly.

If V0.3.0 implementation surfaces a real reason to prefer the integrated form, the design is on the shelf and the migration is straightforward. Collapse the three `loop` functions into one, drop the inter-stage yield boundaries, and inline the staging.

## The Bootstrap Fixed Point

Three phases apply at the point when all three pipeline stages exist in Keleusma source. The pattern is canonical across Wirth's Project Oberon, LLVM, Rust, and Go.

Phase A cross-compiles. The self-hosted compiler is written in Keleusma source under `compiler/lexer.kel`, `compiler/parser.kel`, and `compiler/codegen.kel`, plus shared abstract-syntax-tree and bytecode-encoding helpers. The existing Rust-hosted compiler produces the bytecode. The output is a Keleusma bytecode artifact, which the strategy calls `kelc.0.kel.bin`, that runs on the virtual machine and accepts Keleusma source as input.

Phase B self-compiles. The engineer loads `kelc.0.kel.bin` into a virtual machine instance and invokes it against its own source files as its input. The output is `kelc.1.kel.bin`. If `kelc.0` is correct, `kelc.1` is byte-identical to `kelc.0` modulo non-essential ordering including map iteration order. Any divergence is a bug in `kelc.0`.

Phase C reaches the fixed point. The engineer loads `kelc.1.kel.bin` into a virtual machine instance and invokes it against the same source files. The output is `kelc.2.kel.bin`. `kelc.2` must be byte-identical to `kelc.1`. The fixed point is reached.

Validation runs alongside Phases B and C. Every test in the existing regression corpus is recompiled under both the Rust-hosted compiler and `kelc.1`. The bytecode outputs must be byte-identical modulo the same non-essential ordering. Divergence on the corpus is a bug in the self-hosted compiler.

The bootstrap procedure is mechanical. It does not require additional design work. The risk in the bootstrap is not the procedure but the surface-language gap, namely that the self-hosted compiler may need features the V0.2 surface does not yet provide ergonomically. The strategy addresses this in the required-surface-features section.

## Constraints on the Surface Language

Three surface-language tensions surfaced immediately when the strategy considered writing the compiler in Keleusma.

The first tension is recursion. The self-hosted compiler will want to walk recursive data structures. Parsed declarations contain expressions that contain sub-expressions. Types contain sub-types. Keleusma forbids recursion in `fn` and `yield` categories. Only top-level `loop` admits cyclic execution through productive yield.

The classical resolution is to walk recursive data using explicit stacks rather than recursive function calls. Brinch Hansen's compilers used this technique. The Wirth tradition handled the same constraint with recursive-descent parsers that exploited the fact that the recursion depth was bounded by the language's nesting depth, not the input size. Keleusma's recursion prohibition is stricter and requires explicit stacks. The strategy adopts the explicit-stack pattern, which requires no language surface change against V0.2.

The second tension is Hindley-Milner type inference. The Rust-hosted compiler runs Robinson unification over a constraint graph that spans an entire function. This is a multi-pass procedure within a single function. A pure single-pass compiler in the Wirth tradition does

not perform this kind of inference. The surface language typically requires explicit type annotations.

The realistic V0.3.0 answer is per-function-body inference. The constraint worklist lives in the compiler-loop's persistent data block, bounded by explicit declared limits. Analysis of the Rust-hosted compiler's actual constraint-graph sizes established that a maximum of one thousand twenty-four type variables per function, four thousand ninety-six constraints per function, and sixteen thousand three hundred eighty-four function-body nodes covers the substantial majority of practical programs. The compiler-in-Keleusma is itself written with explicit type annotations so that the self-compilation step exercises the easy inference path.

The third tension is generics and monomorphization. Monomorphization requires the compiler to see every call site of a generic function before it can know which specializations to emit. This is fundamentally a whole-program operation. The realistic V0.3.0 answer is to keep a small specialization table in the compiler's persistent state, across the entire compilation rather than per declaration, and emit specialized chunks lazily as new call sites are discovered. The specialization table grows with the number of distinct specializations, not with the program size. In practice this is a small bound.

## Resolved Design Questions

A dedicated research pass addressed each of these tensions in technical detail and produced recommendations.

The recursion question was resolved with the work-stack pattern. Three worked examples covered the recursion shapes the compiler needs. Pre-order traversal for free-variable collection. Accumulating fold for Robinson unification. Post-order traversal for the bytecode emitter. The pattern requires no language-surface change against V0.2.

The Hindley-Milner inference question was resolved with per-function-body inference bounded by explicit declared limits. Per-function transient memory approximately one hundred thirty kibibytes, plus persistent approximately two hundred fifty kibibytes.

The symbol-table substrate question was resolved with three data structures. A string interner producing `Word` indices. A sorted-array `WordMap<V>` for bulk tables including the function table, the type registry, the specialization table, and the use table. And a linear `LocalScope` for per-scope locals. No new language features required. The design implements directly on the V0.2 surface.

The byte-iteration question was resolved with a host-side strategy. The host passes source as `[Byte; N]`. The lexer uses array indexing. Three host-registered natives handle the residual `Text` work. `compiler::intern_bytes` returns a `Word` interner index. `compiler::text_from_bytes` constructs a `Text` from a byte range for diagnostic messages. `compiler::text_concat` builds composite messages. No surface-language extension required.

The self-validation question was resolved with three-layered validation integrating into the existing test harness. Layer one is byte-identical comparison after canonicalization of the native-name table, constant pool, specialization chunks, and function-name-to-chunk-index map. A SHA-256 hash over the canonical form provides a fast pass-or-fail signal. Layer two is logical equality with diagnostic when layer one fails. It pretty-prints the bytecode and runs a structural diff against the Rust-hosted output. Layer three is behavioral equivalence over the regression corpus. Every program's runtime output matches between Rust-hosted and self-hosted compilation.

The module-scale compilation question was resolved with Modula-2-style separate compilation. Implementation files carry the `.kel` extension. Interface files carry the `.def.kel` extension. Both files are Keleusma source. Per-module specialization tables reset at module

boundaries. Cross-module specializations land on the consumer side bounded by consumer complexity.

## Open Questions

Three questions remain unresolved after the research pass. None are strategy blockers. Each becomes a V0.3.x or implementation-time concern.

The first is the cross-module monomorphization mechanism. The separate-compilation shape is settled. The strategy does not specify how generic functions specialize across module boundaries when modules are separately compiled. The shape of the per-module specialization table and the cross-module instantiation protocol is open.

The second is the diagnostic quality regression bound. How much diagnostic quality is acceptable to lose in V0.3.0 in exchange for the single-pass streaming architecture? The strategy's "as good as the Rust-hosted, where possible" target is qualitative. Single-pass compilers historically have brittle error recovery, per the prior-art survey in the stream-based compilers series. Some regression is expected.

The third is a V0.2 surface adequacy audit. The universal-expressibility argument for the work-stack pattern is sound in prose but unverified in code. Before V0.3.0 implementation begins, a sample exercise compiling Robinson unification, a recursive abstract-syntax-tree walker, and a monomorphization pass against the V0.2 surface should be conducted and measured. If a load-bearing affordance is missing, the work-stack pattern needs supplementation.

## Prior Art and Lineage

The strategy draws on several traditions in language implementation.

The moving-seam incremental rewrite is Fowler's strangler pattern, applied to compiler porting by running the seam backward so the emit boundary retires first.

Self-hosting and bootstrapping are an old tradition in language implementation. The reproducibility comparison used as the completion gate is the one a multi-stage bootstrap performs between its stages.

The pipeline-of-processes compiler architecture comes from Per Brinch Hansen. His book Brinch Hansen on Pascal Compilers published by Prentice-Hall in nineteen eighty-five is the canonical reference. His SuperPascal compiler was itself written in this style and demonstrated the pattern of stream-processor compiler in a stream-processor language. This is the architectural precedent closest to Keleusma's intended design.

The single-pass tradition comes from Niklaus Wirth. The Pascal compiler of nineteen seventy, Modula-2 of the late nineteen seventies, and Oberon of the late nineteen eighties were each designed to be single-pass compilable, with declare-before-use rules and explicit forward declarations for mutual recursion. The Oberon compiler is approximately four thousand lines of Oberon and is published in full source in Project Oberon, by Wirth and Gutknecht, Addison-Wesley nineteen ninety-two, revised edition two thousand thirteen. Wirth's Compiler Construction, Addison-Wesley nineteen ninety-six, is the canonical pedagogy. Wirth's tradition demonstrates that single-pass discipline survives language evolution.

The commercial demonstration comes from Turbo Pascal one point zero through three point zero, Anders Hejlsberg's compiler of the nineteen eighty-three through nineteen eighty-six period, written in eight-oh-eight-six assembly. Turbo Pascal compiled to memory and ran the code from memory with no traditional link step for the default in-memory build. The internals were never released as open source. Turbo Pascal is the commercial proof that single-pass compilation produces compilers fast enough to change developer workflow.

A separate line of work pursues bootstrapping with machine-checked correctness rather than migration mechanics. The CakeML verified bootstrapped compiler is the sharpest instance of that program.

Behind all of it sits Ken Thompson's Trusting Trust argument for why the provenance of the seed matters. Thompson's Reflections on Trusting Trust lecture, delivered as the nineteen eighty-three ACM Turing Award lecture and published in Communications of the ACM in August nineteen eighty-four, is the founding statement of the seed-provenance concern that David A. Wheeler's Diverse Double-Compiling countermeasure of two thousand nine addresses. The recent article on self-hosted silicon compilation develops both of these in a hardware context that the software self-hosting strategy sits alongside.

The C-family tradition, namely GCC, Clang, and the Portable C Compiler line, is explicitly not relevant prior art for this strategy. Those compilers are multi-pass and abstract-syntax-tree-based. They optimize for the opposite trade-off, namely heavy optimization at the cost of compilation speed and memory. The lcc compiler, described in Fraser and Hanson's A Retargetable C Compiler, Addison-Wesley nineteen ninety-five, is a useful counter-example to study for what multi-pass design looks like at small scale, but it is not the model V0.3.0 follows.

## Lessons from a Contemporary Attempt

The brief-lang project is a contemporary language that attempted self-hosting and reached, then stalled at, the frontier V0.3.0 approaches. The strategy's engineering choices were informed by a targeted review of that project. The observations reflect that project at a point in time and are cited for their engineering value, not as endorsement.

The frontend is the achievable part. Codegen and host output are the wall. Brief has a working compiler frontend written in Brief, including a lexer, a parser, a type checker, and a contract engine, but is not bootstrapped because the frontend runs inside a host interpreter and its backends are unfinished. The V0.3.0 work sequences codegen and the emit-to-host boundary as the high-risk stages to be de-risked first, not the lexer and parser. This reinforces the backward-migration ordering.

The output capability must be a first-class host native from the start. Brief's self-hosted compiler could read source but had no general facility to write its output, which is a hard blocker to a true bootstrap. Keleusma's host-native surface must expose a deliberate, bounded emit-compiled-bytecode capability to the self-hosted compiler as a designed feature, not an afterthought.

Divergent execution models are a bootstrap hazard. Brief maintained a tree-walking interpreter and a compiled backend that drifted into different runtime semantics, silently miscompiling programs. Keleusma's bootstrap fixed point, `kelc.0` to `kelc.1` to `kelc.2`, is the guard against this. The fixed point converges only if the self-hosted compiler's output is stable across stages, so the byte-identical fixed-point check in the bootstrap procedure is load-bearing, not ceremonial.

The work-stack idiom is independently validated. Brief's compiler used explicit work-stacks and threaded state, for example iterative depth-first call-graph traversal, even though its language permits recursion. That a real compiler was written this way is external evidence that Keleusma's recursion-free work-stack pipeline can express the compiler it needs to.

Only admit surface syntax that will actually be compiled. Brief accumulated parsed-but-uncodegened constructs, each becoming a maintenance stub that generated defects. The strategy grows the self-hosted compiler feature-complete per increment. Parse, type-check, and generate for a construct together, or not at all.

Consume every analysis result on every path. Brief repeatedly computed an analysis including liveness and convergence and then failed to consume it in one of several codegen paths, losing the benefit and creating inconsistency. A staged self-hosted pipeline must ensure each stage consumes the descriptors the prior stage produced, everywhere they apply.

## Success Criteria

V0.3.0 is complete when nine conditions hold.

First, the compiler pipeline exists in Keleusma source at `compiler/lexer.kel`, `compiler/parser.kel`, and `compiler/codegen.kel`, plus shared abstract-syntax-tree and bytecode-encoding helpers.

Second, the first migration step intermediate validation passes. Every program in the regression corpus compiles to byte-identical bytecode under the Keleusma-lexer-plus-Rust-parser-plus-Rust-codegen configuration as under the all-Rust baseline.

Third, the second migration step intermediate validation passes. Every program in the regression corpus compiles to byte-identical bytecode under the Keleusma-lexer-plus-Keleusma-parser-plus-Rust-codegen configuration as under the all-Rust baseline.

Fourth, the Rust-hosted compiler produces `kelc.0.kel.bin` from the full Keleusma source without error. The existing test suite continues to pass.

Fifth, the Phase B fixed point holds. `kelc.0.kel.bin` recompiles its own source to produce `kelc.1.kel.bin`. `kelc.1` is byte-identical to `kelc.0` modulo non-essential ordering, formally documented.

Sixth, the Phase C fixed point holds. `kelc.1.kel.bin` recompiles the same source to produce `kelc.2.kel.bin`, byte-identical to `kelc.1`.

Seventh, regression corpus equivalence holds. Every script in `examples/scripts/` and the workspace tests compiles to byte-identical bytecode under both the Rust-hosted compiler and `kelc.1`.

Eighth, the command-line frontend gains a `--self-hosted` flag that routes through `kelc.1` instead of the Rust-hosted compile path. Programs compile and run end to end.

Ninth, documentation acknowledges the self-hosted compiler as an alternative path. The Rust-hosted compiler continues to ship. V0.3.0 does not retire it.

The dual-compiler period is intentional. The Rust-hosted compiler remains the reference implementation. The self-hosted compiler is the validation that the language admits its own toolchain.

## Conclusion

The V0.3.0 strategy combines a general-purpose migration method with a Keleusma-specific pipeline architecture. The migration method is a backward incremental rewrite that ports the emit boundary first, maintains a single moving adapter at the frontier, tracks intermediate deviations in a deferral ledger, and drives the ledger to empty at the completion gate. The architecture is a three-stage stream-processor pipeline matching Brinch Hansen's pipeline-of-processes model and Keleusma's coroutine semantics. Both parts draw on established compiler-implementation traditions that the stream-based compilers series covers in depth.

The endpoint is a fixed point. The self-hosted compiler compiled by itself reproduces its own bytecode. Reaching that fixed point is the V0.3.0 goal. The V0.4.0 native-code-generation goal depends on it. The V0.5 and beyond Keleusma-in-Keleusma-runtime aspiration depends on V0.4.0. Each step retires a dependency that the current shape carries.

The strategy is documented in full in the Keleusma repository as an incremental self-hosting reference document that states the migration method independent of Keleusma and a V0.3.0 self-hosting strategy document that applies the method to the case. The subproject scaffold described in the recent V0.2.2 getting-started article is where the strategy is being realized.

## References


- Brinch Hansen, Per, Brinch Hansen on Pascal Compilers, Prentice-Hall, 1985
- Bootstrapping Compilers
- CakeML Verified Bootstrapped Compiler
- Fowler, Martin, Strangler Fig Application
- Fraser, Christopher W. and Hanson, David R., A Retargetable C Compiler, Design and Implementation, Addison-Wesley, 1995
- GCC Multi-Stage Bootstrap and Stage Comparison
- Keleusma, GitHub Repository
- Keleusma, Incremental Self-Hosting Reference Document
- Keleusma, V0.3.0 Self-Hosting Strategy Document
- Reproducible Builds
- Self-Hosting Compilers
- Thompson, Ken, Reflections on Trusting Trust, CACM, 1984
- Tombstone and T-Diagrams
- Wheeler, David A., Diverse Double-Compiling, arXiv, 2009
- Wirth, Niklaus, Compiler Construction, Addison-Wesley, 1996
- Wirth, Niklaus and Gutknecht, Jürg, Project Oberon, Addison-Wesley, 1992, revised 2013
- Related Post, Streaming Compilers Series Conclusion
- Related Post, The Self-Hosted Silicon Compiler
- Related Post, Developments in Programming Language Theory, A Historical Arc
- Related Post, Getting Started with Keleusma 0.2.2