Ports as Instruments: Comparing Swift with the n0is C++ Engine

Tyler Garrett / 2026-11-08 / Working paper 0.6

Background

I first worked with C++ when I was young, through video game technology. I wondered how games stored their data, what they could process in batches, and what they needed to stream during play. That curiosity led me to study information systems in college and, years later, to investigate how the n0is C++ engine and a Swift port process the same data.

Over the past 15 years, I've worked full time at Tableau and Alteryx. As a senior business consultant in Tableau's post-sales professional services team, I worked on about 80 client engagements and saw teams use scripts and drag-and-drop tools to extract, transform, and load data. Even when a project called for code, I often reached for a visual tool so I could see what each node did to the data. SQL came naturally to me for the same reason. As I write a query, I can picture which rows a filter keeps and how a join brings tables together.

I used SAP Data Services at an SAP BusinessObjects consultancy. There and at Alteryx, I saw how licensing costs shaped what teams could afford to deploy. I also saw how separate scheduling fees and limits on where workflows could run complicated something as ordinary as an overnight job. I gravitated toward the open-source KNIME Analytics Platform because I could lay out a pipeline visually and trace how each step changed the data.

Well before large language models became part of my development process, I was building data pipelines in Node.js and working with the data engineering tools in its ecosystem. JavaScript was a natural choice for the first browser-based n0is engine. Over the past few years, I focused more on streaming. I wanted more control over memory and execution as I pushed for lower latency, so I returned to C++.

The platform took shape over roughly ten years of sketching data flows and turning those sketches into working software. I hired full-stack engineers from Dallas-Fort Worth and spent many late nights implementing parts of it myself. How fast could I make a streaming pipeline, and what could other implementations teach me when they were doing the same work?

Abstract

A faster implementation can point to a better algorithm, less work, or a different stopwatch. We study these explanations by comparing a Swift CSV loader with the corresponding path in the native C++ engine that Tyler Garrett built for n0is. The engine runs interactive extract, transform, and load (ETL) workflows. During the engine rewrite, separate language-model agents implemented four kernels in Rust, Swift, Zig, and Go. We used those ports to propose changes to the C++ implementation, then checked equivalent output and tested candidate changes against preserved baselines.

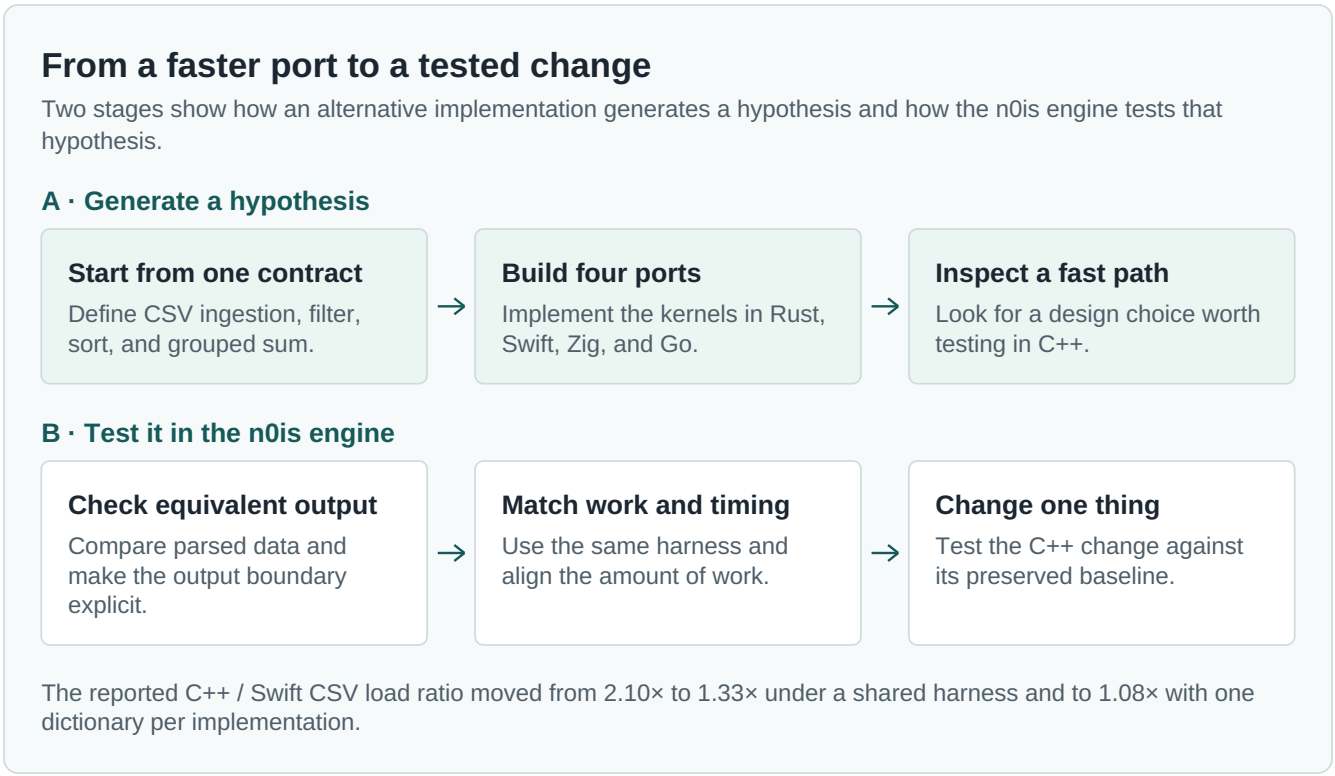


Figure 1. The ports suggest changes. A shared harness tests each C++ change against its preserved baseline.

On a synthetic one-million-row CSV workload, a historical C++/Swift latency ratio of 2.10 fell to 1.33 under a common harness and to 1.08 when the C++ path built the same number of dictionaries. The historical and common-harness measurements also used different C++ revisions, so that first change cannot be attributed to timing alone. The study contributes a practical comparison protocol and illustrates two failure modes: inconsistent output boundaries and overlapping measurements of system and language effects. A later matched rerun with 30 blocks per condition measured C++/Swift load-time ratios from 0.729 to 0.841 across two row counts and two timing modes on an Apple M5 Max.

1. Research question

Can separately authored ports reveal useful optimization opportunities even when their raw times are poor evidence about programming languages? A profile describes an existing implementation. A port supplies an alternative implementation that can be checked, profiled, and selectively copied. The goal is to determine which alternative choices survive a controlled comparison and which advantages disappear once the work and clock agree.

We distinguish three claims: **semantic equivalence** on the exercised inputs, **relative performance** of named implementations under an explicit protocol, and **attribution** to a specific design choice. Passing one claim is insufficient evidence for the next.

2. System and workload

The n0is C++ engine executes graphs of table transformations. The port experiment covered CSV ingestion into a materialized table, a numeric filter, a stable numeric sort followed by materialization, and a grouped sum. The focal experiment used a synthetic, approximately 46.3 MB CSV with one million rows and six columns. The

loader parsed the file, converted the amount column to doubles, and built a region dictionary. The production C++ path also built two additional dictionaries for downstream work. The implementations targeted the n0is engine's parsing and numeric rules, including its treatment of whitespace, quotes, short rows, numeric prefixes, and NaN; the study did not establish equivalence for all CSV dialects.

The focal paired run took place on one Apple M5 Max system. The common harness alternated order and summarized all timed loads rather than taking the fastest. The comparison used 16 threads for the primary Swift comparison; the historical Swift suite had used its default 18. A separate preflight compared fingerprints of parsed string cells and table dimensions. That check did not cover every timed execution or its derived typed and dictionary values. Some earlier kernel results had only partial output checks, so those results are useful for proposing changes rather than ranking implementations.

3. Protocol

1. **Write the output contract.** Pin the input bytes, type rules, tie behavior, null semantics, order, and materialization boundary. Compare full canonical outputs where feasible. Preserve adversarial cases alongside ordinary ones.
2. **Use a common harness.** Run on the same machine; record revisions, build flags, compiler and runtime versions, thread counts, inputs, and timer boundaries. Interleave arms. Separate fresh-process measurements from repeated operation inside a process. A fresh process does not imply a cold OS file cache.
3. **Equalize work and show both modes.** Keep the as-shipped path as a user-facing measurement. Separately compare the shared contract after disclosing optional work, such as auxiliary dictionaries. Record the end-to-end and phase clocks separately.
4. **Test a mechanism in the host.** Translate the relevant loop or change one design choice while holding the rest of the host implementation steady. Compare with a preserved baseline. Treat any residual gap as unexplained unless another intervention isolates it.
5. **Audit the harness and retain failures.** Ask someone who did not write it to check timer placement, data conversion, excluded work, buffer reuse, and statistics. Keep raw per-run observations and report rejected changes. For a confirmatory study, predeclare the process or paired block as the unit and report uncertainty for the paired effect [1, 2].

4. Results

4.1 A Swift lead shrank as the comparison changed

Historical audit.

Comparison	C++ (ms)	Swift (ms)	C++ / Swift
Historical headline; different revisions or harness conditions	6.33	3.02	2.10×
Common repeated-load harness; C++ builds three dictionaries	4.89	3.69	1.33×
Common repeated-load harness; both build one dictionary	3.72	3.44	1.08×
Common fresh-process harness; C++ builds three dictionaries	6.86	5.94	1.15×

Table 1. The common-harness ratios divide the rounded C++ median by the rounded Swift median. The first row uses the historical headline timings. Timing starts once the input bytes are available. The fresh-process row measures its first parse, conversion, and dictionary build. The two repeated-load rows are descriptive comparisons from the October 11, 2026 study; the notebook records eleven alternating process rounds, with three measured loads per round per implementation, for the initial common-harness comparison. These summaries do not give the paired distribution or a confidence interval.

The 2.10× headline mixed a prior C++ revision with Swift's own best-of-three measurement. The 1.33× and 1.08× rows use a shared timing procedure. Switching C++ from three in-scan dictionaries to a plain parse followed by one dictionary moves the result closer to parity, but the Swift median also moves across the runs. The rows diagnose the size of a comparison problem; subtracting them would not yield a clean causal estimate of timing bias or of dictionary cost. The remaining 1.08 C++/Swift median ratio is unresolved under this design.

Thread count could change which implementation led. In the common repeated-load comparison, Swift's median was 3.69 ms at 16 threads and 6.06 ms at 18, its default. The n0is engine's 4.89 ms median fell between those two results. With one dictionary per implementation, the best observed parse phase took 1.98 ms in C++ and about 2.0 to 2.1 ms in Swift. Building the single dictionary took 0.23 ms and 0.24 ms, respectively. Those phase observations helped locate where the work was going. The whole-load medians in Table 1 summarize a different set of timings.

Matched rerun. On November 8, 2026, we compared pinned C++, Swift, Rust, Zig, and Go builds in 30 balanced blocks per condition on one Apple M5 Max running on battery. Each block ran every implementation once, with balanced execution order across the blocks. Table 2 reports the Swift comparison. Both builds used the same six-column CSV fixtures, requested 16 participants under a common scheduling priority ceiling, converted the amount column to float64, and built one region dictionary. Checks before and after the run matched the parsed-cell fingerprints for both fixtures. We retained every timed observation.

The first-load clock starts in a fresh process after input acquisition and runtime setup. The warmed observation is the median of three timed loads after three in-process warmups. Swift parses its header and gives its worker pool 20 ms to settle before timing. Each process contributes one observation to its block.

CSV load	C++ (ms)	Swift (ms)	Paired ratio (interval)	C++ faster
1M, first load	5.508	6.853	0.841 (0.783-0.912)	28/30
1M, warmed	2.763	3.295	0.829 (0.795-0.854)	30/30
10M, first load	47.246	60.934	0.773 (0.756-0.789)	30/30
10M, warmed	34.355	47.228	0.729 (0.714-0.745)	30/30

Table 2. Times are medians over 30 process observations per build and condition. The paired ratio is the geometric mean of within-block C++ time divided by Swift time. Intervals use 50,000 bootstrap resamples of whole rounds with a Bonferroni adjustment across all 16 comparisons in the five-build study, targeting approximate 95% simultaneous coverage under the assumption that rounds are exchangeable. A ratio below 1 favors the C++ build for this clock.

The C++ build led in 28 of 30 blocks for the 1M first load and in all 30 blocks in each other condition. At 10M warmed, the paired ratio of 0.729 corresponds to 27.1% lower elapsed time. The adjusted interval spans 0.714 to

0.745. These estimates describe the tested builds and fixtures. The parsed-cell check covers the text table, while the numeric and dictionary outputs were outside that check.

4.2 The candidate improved across all tested sizes

The follow-up interleaved the preserved C++ build, the C++ candidate, and the Swift port on the same Mac. The candidate moved dictionary lookup state out of the inner loop, reduced overhead in infrequently called helpers, and ran dictionary planning alongside the quote scan. The C++ builds still produced three dictionaries, and Swift produced one. The Swift column uses its faster tested thread setting for each size. Other work was active on the machine during the runs.

CSV workload	Prior C++ (ms)	Candidate (ms)	Swift (ms)	Candidate faster vs prior / Swift
1M rows, repeated loads	4.26	3.31	3.36	11/11, 8/11
1M rows, first load in a fresh process	6.80	6.11	6.15	13/15, 7/15
10M rows, repeated loads	35.52	29.23	45.20	7/7, 7/7
100M rows, repeated loads	488.26	404.67	415.46	3/3, 2/3

Table 3. Times are medians of measured loads. Each repeated-load process contributed three timings. Each fresh process contributed one. The last column counts the interleaved rounds in which the candidate was faster than each comparator.

At one million rows the repeated-load candidate median was 22% lower than the prior C++ median. Its median was within 0.05 ms of Swift in both comparisons. In fresh processes the candidate had the lower median but was faster in seven of the fifteen individual rounds. At ten million rows it led Swift in all seven rounds, with medians of 29.23 ms and 45.20 ms. At one hundred million rows the two medians were closer again, and that comparison had only three rounds. The pattern varied with input size, and the medians and round counts are descriptive.

A separate 100M-row follow-up targeted memory traffic. The C++ candidate avoided initializing output buffers that were about to be overwritten and changed how it copied dictionary IDs. The load median fell from 410.6 ms to 373.4 ms, with four faster rounds out of four. The recorded numeric conversion and dictionary phases fell from 70 to 42 ms and from 23 to 18 ms, respectively. That comparison tested the two changes together and used a separate baseline from Table 3.

4.3 Output and operator boundaries changed the comparison

One historical table put Zig's payload gathering time beside measurements that included both sorting and gathering. In the Zig port's one-million-row notes, gathering alone took 2.65 ms and the complete permutation and gather took 4.0 ms. The filter table had a similar boundary problem. In one C++ run, a bitmap result took 0.014 ms and an index array took 0.439 ms. The ports in the table produced index arrays. These examples show why the timing boundary has to name the exact output the caller receives.

The grouped-sum comparison revealed a different boundary. A small kernel runs on numeric columns that are already typed. The n0is aggregate step also has to convert cells and manage group state. In a separate interleaved comparison at one million rows, a C++ revision that converted columns once and simplified group bookkeeping

reduced the median for sum by region from 7.46 ms to 1.13 ms across nine process runs per build. All 613 cases in the JavaScript parity suite passed at that revision. Measuring the full step showed where the time went in the engine.

5. Related work

Knight and Leveson's multi-version programming experiment warns that separately authored implementations can have correlated errors [3]. Our ports share a brief, input, and target implementation, so agreement is a source of hypotheses, not independent evidence. Raasveldt et al. describe unfair tuning, unverified results, and other database benchmarking pitfalls [1]. Kalibera and Jones give a framework for uncertainty across repeated executions and builds [2]. Mytkowicz et al. show that apparently irrelevant experimental choices can alter a performance conclusion [4]. The proposed protocol applies these lessons to alternative implementations used as a tool for optimization.

DuckDB and MonetDBLite give precedents for embedded analytical execution [5, 6]. ParPaRaw and speculative distributed CSV parsing establish prior art for resolving chunk context without a simple sequential parse [7, 8]. The present study therefore makes no priority claim for parallel CSV parsing or column-oriented execution. TRACE measures efficiency of LLM-generated translations over a task suite [9], while Shypula et al. study performance-improving edits proposed by models [10]. Nanz et al. compare multiple implementations of multicore benchmarks with expert review [11]. Our narrower question is whether a port's choices can improve the original host after work and measurement are aligned.

6. Limits and replication

The original port study was retrospective and the agent briefs were not preserved. The main comparison uses one synthetic file shape and one machine with heterogeneous cores. Compiler effort, process state, cache state, and residual system load were not independently randomized or fully controlled. Some original observations survive as notebook summaries and are reported descriptively. The November 8 intervals describe the tested builds and workload on one machine. DuckDB, Polars, and ClickHouse require separate experiments with matched input, result, and timing boundaries.

A broader replication should add a public wide CSV slice and a quote-heavy edge corpus to the 1M- and 10M-row generated fixtures. Record full canonical outputs before timing, interleave a prespecified number of fresh-process pairs, and publish paired ratios with intervals and separately labeled repeated-load results. Retain raw timing rows with machine and build configuration. Repeat on a second host before making a portability claim. Reproduce the protocol with a small, independently written reference loader and generator that can be shared without distributing the n0is engine. Independent replication of the historical measurements requires access to the private engine and the original implementation artifacts.

7. Conclusion

The useful result was the change in the question. The initial headline invited a judgment about Swift and C++. The audited comparisons instead exposed extra dictionary work, inconsistent timing, and distinct materialization contracts. Ports are useful as instruments for generating and testing such hypotheses when the study reports the contract and tests the proposed mechanism in the host.

Further reading

- First and warmed CSV loads in Zig and n0is C++
- Parallel evaluation throughput and latency in n0is
- ET1 Basic Training from Dev3lop

References

1. Raasveldt, M., Holanda, P., Gubner, T., and Mühleisen, H. Fair Benchmarking Considered Difficult: Common Pitfalls in Database Performance Testing. DBTest, 2018.
2. Kalibera, T., and Jones, R. Quantifying Performance Changes with Effect Size Confidence Intervals. arXiv, 2020.
3. Knight, J. C., and Leveson, N. G. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. IEEE TSE, 1986.
4. Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. Producing Wrong Data Without Doing Anything Obviously Wrong!. ASPLOS, 2009.
5. Raasveldt, M., and Mühleisen, H. DuckDB: An Embeddable Analytical Database. SIGMOD, 2019.
6. Raasveldt, M., and Mühleisen, H. MonetDBLite: An Embedded Analytical Database. arXiv, 2018.
7. Stehle, E., and Jacobsen, H.-A. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. PVLDB, 2020.
8. Ge, C., Li, Y., Eilebrecht, E., Chandramouli, B., and Kossmann, D. Speculative Distributed CSV Data Parsing for Big Data Analytics. SIGMOD, 2019.
9. Gong, Z., Sun, Z., Huang, D., Liang, Q., Zhang, J. M., and Hao, D. TRACE: Evaluating Execution Efficiency of LLM-Based Code Translation. ACL, 2026.
10. Shypula, A., et al. Learning Performance-Improving Code Edits. ICLR, 2024.
11. Nanz, S., West, S., Soares da Silveira, K., and Meyer, B. Benchmarking Usability and Performance of Multicore Languages. arXiv, 2014.