Choosing an inference library for Lege

tract, RTen and ONNX Runtime in a reproducible OCR pipeline

21CSM · November 2026

Abstract. Lege, a command-line OCR tool, runs three small neural networks on every page and needs an inference library to run them. We compared tract, RTen and ONNX Runtime inside Lege itself, on 25 reference images, on an Apple M3 Pro, GitHub's Linux x86-64 and ARM64 runners, and two emulated older x86 CPUs, under a decision rule committed before the second round of results. All three read the same text everywhere, stayed far under 2 GB, and ran on CPUs without AVX. The deciding requirement was the same output at every thread count: tract and RTen met it everywhere, while ONNX Runtime as built by Nix changed one confidence score at 4 threads on the M3 Pro. Microsoft's official build of the same version met it on every machine. Judged by the build Lege would ship, ONNX Runtime passes every requirement and is fastest on one Linux x86-64 thread (64.5 s for the 25 pages, against 110.5 s for RTen and 123.3 s for tract), so it is chosen, on condition that Lege ships Microsoft's official build. The study also found and fixed a defect in Lege that made tract about a third slower than it is.

1 Introduction

Lege turns a scanned page into ALTO 4.4 XML and plain text. It is a command-line tool written in Rust, built for archives and for anyone who needs the same output every time. For each page it runs three PP-OCRv5 neural networks in turn:

1. a detector finds the text lines on the page;
2. a direction classifier checks whether each line is upside down;
3. a recognizer reads the characters in each line.

Everything else, from decoding the image to writing ALTO, is Lege's own code. The one piece it borrows is the inference library that runs the networks. That library sets most of the speed and memory use, decides whether the output stays the same at every thread count, and decides what has to ship with Lege.

Lege started with tract, a pure-Rust ONNX runtime, without measuring the alternatives. This report measures tract against two others before that choice is made final:

- **RTen**, a younger pure-Rust runtime that loads ONNX files directly;
- **ONNX Runtime**, Microsoft's C++ runtime and the most widely used option, reached from Rust through the `ort` crate.

2 Requirements and decision rule

Lege makes four promises that an inference library must keep. They became hard requirements; failing any one rules a library out:

1. **Same output at any thread count.** On one machine, the ALTO file is byte-identical at 1, 2, 3, 4, 6 and 8 threads, for every reference image.
2. **Runs on old CPUs.** Every page runs, and the text is right, on an x86 CPU without AVX and on one with AVX but no AVX2.
3. **Fits a 4 GB machine.** Peak memory stays under 2 GB on every page.
4. **Same text.** The text matches the approved snapshot on every page.

Among libraries that pass, the ranking is: speed on one thread on Linux x86-64 (the closest stand-in for low-end machines, which have few cores), then speed at 4 threads on the Mac; totals within 10% count as a tie. Ties are broken by nothing extra to ship, then lower memory, then fewer added dependencies.

This rule was written and committed before the second round of Linux and emulation results came in, so that the data could not be read to fit a favourite.[1]

3 Method

One code base, three builds. Lege keeps its inference calls behind four functions in one module. For this study a second implementation of those four functions was written for RTen and for ONNX Runtime, chosen by a build feature. Image preparation, box finding, character decoding and ALTO writing are the same code in all three builds, so the measurements reflect the real pipeline.

Library	Version	How Lege reaches it
tract	0.23.9	Rust crate, compiled in
RTen	0.27.0	Rust crate, compiled in
ONNX Runtime	1.27.1	`ort` 2.0.0-rc.15; C++ library loaded at run time

Table 1: The three libraries. All builds use Rust 1.99.1 in release mode and the same PP-OCRv5 mobile models, pinned by SHA-256.

Workload. The 25 images of Lege's acceptance tests: 20 scanned book pages printed 1871–1920 in English, French, German, Spanish and Polish (1200 pixels wide, 1 to 220 lines each) and 5 synthetic lines. Some images are the same page saved as another TIFF type or turned upside down, so there are about 17 distinct pages.

Timing. One `lege` process per page, as people run it. The time is Lege's own per-page timer (decode, recognize, write). Each page has a warm-up run and then three timed runs; the median is kept. The libraries take turns image by image and run by run, so a slowdown partway through affects all three alike. Peak memory is the process's maximum resident set size.

Correctness. Each run's text and ALTO are compared with the approved snapshots: byte for byte on Linux x86-64, the machine the snapshots were made on, and within the acceptance tests' tolerance on confidence scores elsewhere. Thread-count checks run every image once at each of 1, 2, 3, 4, 6 and 8 threads and compare the ALTO bytes with the 1-thread run.

Machines. An Apple M3 Pro laptop; GitHub's hosted Linux runners for x86-64 and ARM64; and QEMU user-mode emulation of two older x86 CPUs, Nehalem (SSE4.2, no AVX) and SandyBridge (AVX, no AVX2). A small program confirmed that each emulated CPU reports the missing instructions as absent. Under emulation only correctness is meaningful, not speed: a missing instruction would crash the run, so a clean run shows the code avoids it.

4 A defect found by the benchmark

Most of tract's slowness in the first round was Lege's own fault: Lege rebuilt the line-reading model for every new line width, and on a typical page that cost more than reading the lines.

tract prepares a model for one exact input shape. Each text line is scaled to 48 pixels high, so its width varies, and almost every line brings a new width. For each new width Lege parsed the ONNX file and ran all of tract's preparation passes again. On one 34-line page, 25 rebuilds took 2.0 s of the page's 3.4 s.

Preparation pass	Time across 25 rebuilds
Parsing the ONNX file	43 ms
Type analysis	289 ms
Decluttering (graph simplification)	1,251 ms
Optimizing for the exact shape	391 ms

Table 2: Where the rebuild time went, measured with temporary timers.

Leaving the width open as a symbol removed the rebuilds but made reading much slower (8.6 s for the page instead of 3.5 s), because tract's fastest kernels need concrete shapes. The fix prepares the model once with the

[1] Commit `63ffb13` in the Lege repository, `docs/spikes/runtimes-decision-rule.md`, 2026-11-08 16:28 EST. The Mac results were already known at that point; the Linux and emulation results were not.

width left open, up to and including decluttering, and for each new width only substitutes the number and runs the final optimization. Output is byte-identical to the old code on all 25 images.

tract on the Mac, 25 pages	Before the fix	After the fix
1 thread	73.3 s	47.5 s
4 threads	60.3 s	35.4 s
Peak memory	0.94 GB	1.06 GB

Table 3: The fix cuts tract's time by more than a third, at a cost of about 0.1 GB of memory.

The fix helps Lege whichever library is chosen, since tract is what Lege uses today.

5 Results

5.1 Speed

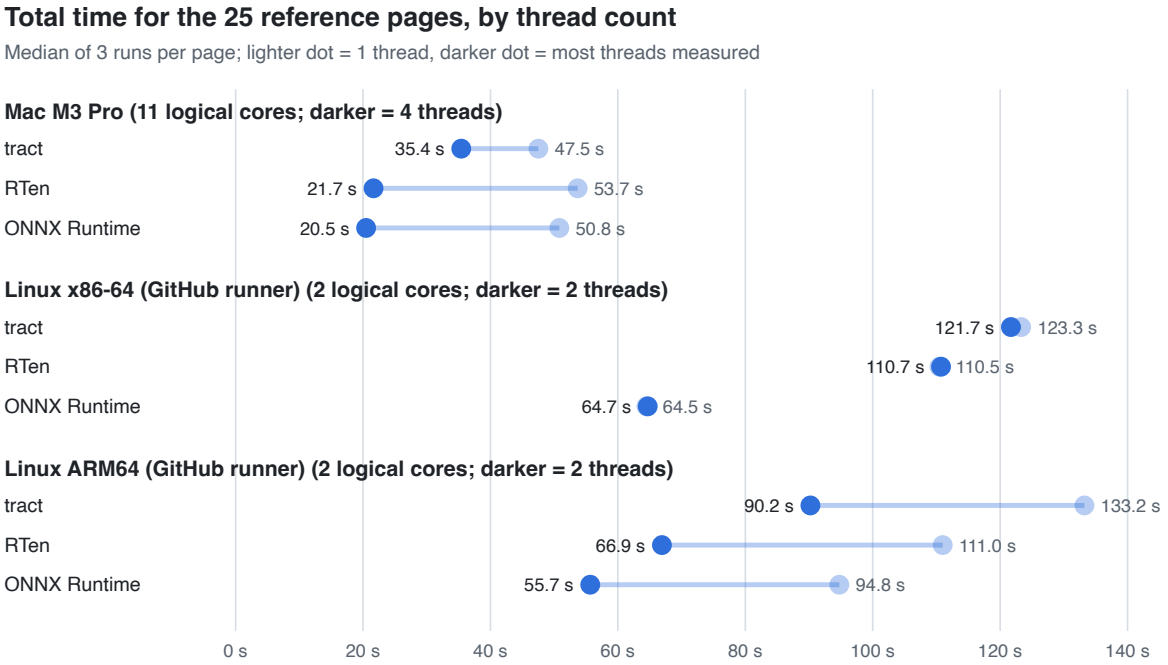


Figure 1: Total time for the 25 pages on each machine. Each dot is the sum of per-page medians of three runs.

Machine, threads	tract	RTen	ONNX Runtime
Mac M3 Pro, 1 thread	47.5 s (47.3–48.4)	53.7 s (53.4–54.1)	50.8 s (50.5–51.4)
Mac M3 Pro, 4 threads	35.4 s (34.7–36)	21.7 s (21.6–21.8)	20.5 s (20.5–20.5)
Linux x86-64, 1 thread	123.3 s (122.3–124.4)	110.5 s (110.2–111.1)	64.5 s (64.2–64.9)
Linux x86-64, 2 threads	121.7 s (121–122.5)	110.7 s (110.5–111.1)	64.7 s (64.5–65.5)
Linux ARM64, 1 thread	133.2 s (132.5–134.1)	111 s (110.8–111.4)	94.8 s (94.5–95.4)
Linux ARM64, 2 threads	90.2 s (89.9–90.5)	66.9 s (66.8–67.1)	55.7 s (55.4–55.8)

Table 4: Total seconds for 25 pages: median, then the sum of the fastest and of the slowest runs in brackets.

On one thread the three are close on the Mac, but on Linux ONNX Runtime is far ahead: on x86-64 it took 64.5 s against 110.5 s for RTen and 123.3 s for tract. With more threads, RTen and ONNX Runtime pull well ahead of tract on the Mac and on ARM64. The x86-64 runner had one physical core with two hardware threads, so a second thread brought nothing there.

5.2 Memory

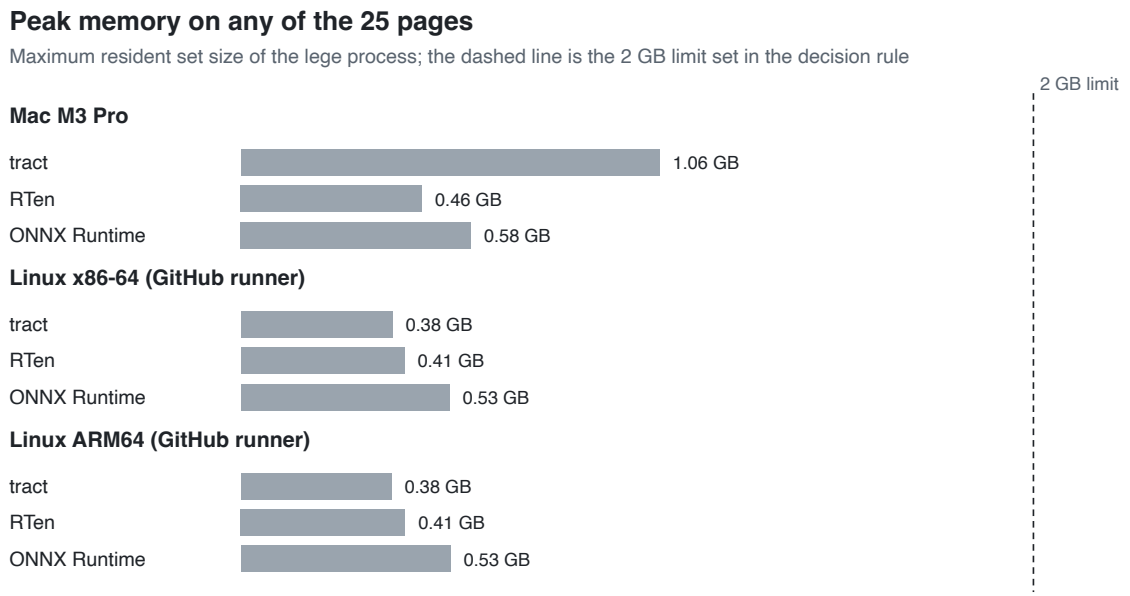


Figure 2: Peak memory on any page. All three stay well under the 2 GB limit on every machine.

RTen used the least memory on the Mac. On Linux all three peaked between 0.38 and 0.53 GB. tract's Mac peak of over 1 GB is not yet explained; macOS counts resident memory differently from Linux, so memory is best compared within one machine.

5.3 Reading the text

Every library read exactly the snapshot text on all 25 images on every machine.

ALTO identical to the snapshot	tract	RTen	ONNX Runtime
Mac M3 Pro	22 of 25	23 of 25	24 of 25
Linux x86-64	25 of 25	23 of 25	24 of 25
Linux ARM64	22 of 25	23 of 25	24 of 25

Table 5: Files that are not identical differ only in confidence scores, within the acceptance tests' tolerance. Linux x86-64 is the machine the snapshots were made on.

5.4 Same output at any thread count

1, 2, 3, 4, 6 and 8 threads	tract	RTen	ONNX Runtime (Nix build)
Mac M3 Pro	same at every count	same at every count	**differs**: `pd-aurelius-meditations-1887-n100-rot180` at 4 threads
Linux x86-64	same at every count	same at every count	same at every count
Linux ARM64	same at every count	same at every count	same at every count

Table 6: Every image run once at each thread count; ALTO compared with the 1-thread run.

tract and RTen gave byte-identical output at every thread count on every machine. ONNX Runtime did too on both Linux machines, but on the Mac one image's ALTO differs at exactly 4 threads, and only there: one confidence score moves from 0.2968 to 0.2967. Three runs at each count gave the same hashes, so this is a fixed property of the thread count, not noise. ONNX Runtime's deterministic-compute option does not change it.

The decision rule judges ONNX Runtime by the build Lege would ship. So the same check was run with Microsoft's official build of ONNX Runtime 1.27.1, downloaded from its release page and pinned by SHA-256, on every machine; on Linux, Lege was also built without Nix for this check.

Official ONNX Runtime, 1–8 threads	Result
Mac M3 Pro (this laptop)	same at every count
Mac M2 Pro (GitHub runner)	same at every count
Linux x86-64, built without Nix	same at every count
Linux ARM64, built without Nix	same at every count

Table 7: Microsoft's official ONNX Runtime build, checked the same way.

The official build gave the same output at every thread count on all four machines, including the M3 Pro where the Nix build did not. On the M3 Pro it also read the snapshot text on all 25 images. The Nix build itself passed on a GitHub M2 Pro, so the mismatch depends on both the build and the chip. Its cause was not investigated.

5.5 Old CPUs

All three libraries ran every page correctly on both emulated CPUs.

Emulated CPU: text matching the snapshot	tract	RTen	ONNX Runtime
Nehalem (SSE4.2, no AVX)	3 of 3	3 of 3	3 of 3
SandyBridge (AVX, no AVX2)	3 of 3	3 of 3	3 of 3

Table 8: Three representative pages (a clean synthetic line, an upside-down page and a fax-compressed TIFF), one thread, under QEMU user-mode emulation. No run failed.

Emulation runs 10–50 times slower than real hardware, so these runs show only that each library avoids instructions an old CPU lacks; a missing instruction would have crashed the run. They say nothing about real speed on old machines.

5.6 Building and shipping

	Crates added	Must ship with Lege	Notes
tract	none (current)	nothing	Pure Rust
RTen	11	nothing	Pure Rust; loads `.onnx` files directly
ONNX Runtime	3	28 MB C++ library, plus protobuf-lite and libiconv	136 MB in Nix; found at run time

Table 9: What each library adds to Lege.

6 GPU and Neural Engine

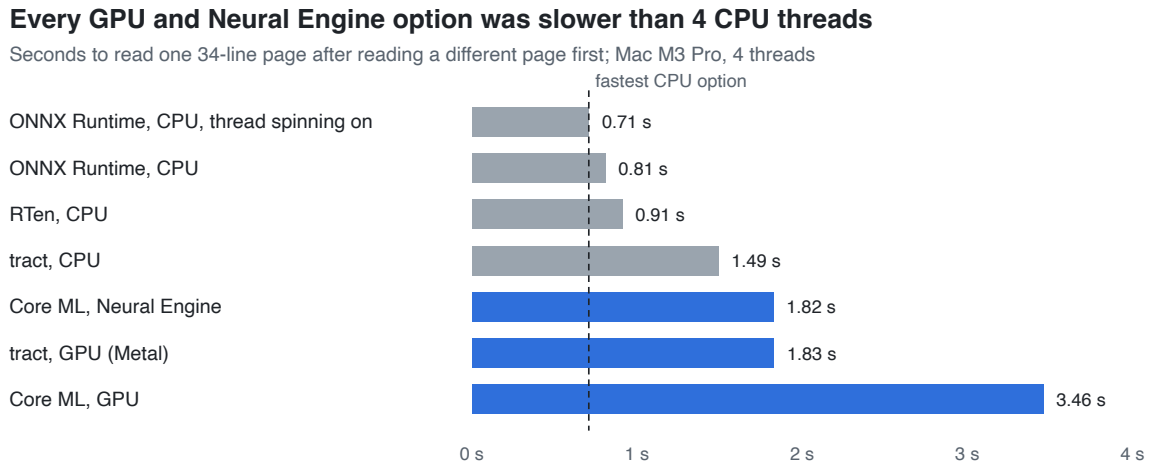


Figure 3: CPU against GPU and Neural Engine options on one page. Recorded in `spikes/runtimes/results/mac-m3pro-gpu.json`.

On the Mac, tract's Metal backend and ONNX Runtime with Apple's Core ML both read exactly the same text as the CPU. None was worth using: on a fresh page, every GPU and Neural Engine option took 1.8 s or longer, while RTen and ONNX Runtime on 4 CPU threads took 0.8–0.9 s. The networks are the small mobile versions and Lege reads one line at a time, so moving each line to the GPU and back costs about as much as the work. Reading many lines in one batch could change that, but batching pads lines to a common width and changes the output, so under Lege's rules it could only be an opt-in setting. tract on Metal also shifted confidence scores slightly, so GPU output would not match the CPU snapshots byte for byte. NVIDIA GPUs were not tested.

7 Threats to validity

An adversarial review of the first round found four problems that changed the numbers. All are corrected in this report.

Problem in the first round	Effect	Correction
Each library ran as one block	RTen at 1 thread measured 74.0 s, later 53.7 s	Libraries take turns image by image
A warm GPU test reread the same page	tract looked faster than it is (1.41 s against 2.06 s)	Warm up on a different page
GPU compared with 1 CPU thread	GPU options looked competitive	Compared with 4 threads
ONNX Runtime thread spinning off	About 12% slower at 4 threads	Measured both ways

Table 10: Corrections from the review.

Other limits remain:

- **Small runners.** GitHub's hosted runners for a private repository had two logical cores: on x86-64 a single physical core with two hardware threads, on ARM64 two cores. Thread scaling beyond two was measured only on the Mac. The first and second rounds also landed on different x86-64 CPUs (AMD EPYC 9V45, then 7763), so Linux times are not comparable between rounds.
- **A small workload.** 25 images, about 17 distinct pages, mostly English-language books printed 1871–1920. Newspapers and dense layouts are not included. The ranking is stable, since runs varied by under 2%, but the absolute times say little about other kinds of page.
- **One page per process.** This matches how the command line is used. A process reading many pages would spread one-off setup across them, which would favour libraries with costly setup.
- **Thread-count checks are finite.** No mismatch was found for tract or RTen at 1, 2, 3, 4, 6 and 8 threads on four machines, but other counts and other machines were not tried. ONNX Runtime's mismatch appeared on one chip at one thread count, which shows how narrow such a problem can be.
- **Emulation proves correctness, not speed.** Real speed on an old machine needs real old hardware, which this study did not have.
- **The decision rule was committed locally before the second round reported, and pushed afterwards.** A commit date can be set by its author, so this is a statement of practice rather than proof.

8 Decision

Applying the rule in Section 2:

- **Requirements.** tract and RTen pass all four on every machine. ONNX Runtime passes all four with Microsoft's official build; the Nix build fails the thread-count requirement on the M3 Pro, so Lege must not ship it.
- **Ranking.** On one Linux x86-64 thread ONNX Runtime took 64.5 s, RTen 110.5 s and tract 123.3 s. ONNX Runtime is ahead by far more than the 10% tie margin, so the ranking ends there. It was also fastest at 4 threads on the Mac.

Lege will use ONNX Runtime, shipping Microsoft's official build, pinned by version and SHA-256. The cost is real: Lege stops being one self-contained binary and must ship a 28 MB C++ library, and its output now depends on a build it does not compile. Two follow-ups protect the promise of identical output: the thread-count check becomes part of Lege's continuous integration for every supported machine, with the official library, and any upgrade of ONNX Runtime must pass it before it lands.

Two limits apply to this decision. The speed figures were measured with the Nix build of the same version; the official build was checked for correctness, not timed, and a gap large enough to change the ranking (over 40%) is unlikely but was not ruled out. And if shipping the C++ library proves impractical on some platform, the rule's next choice is RTen, which beat tract on one thread by 11.6% and passed everything without anything extra to ship.

The tract defect fix in Section 4 is worth keeping in Lege's history even though tract is replaced: it is a reminder to profile the whole program, because the slow part is often our own.

9 Reproducing this report

All code and data are in the Lege repository on the branches `spike/runtimes` and `spike/gpu`, which are not meant to be merged. Inside `nix develop .#runtimes`:

```
cargo build --release -p lege-cli
cargo build --release -p lege-cli --features rt-rten --target-dir target/rt-rten
cargo build --release -p lege-cli --features rt-ort  --target-dir target/rt-ort
python3 spikes/runtimes/bench.py --models MODELS --out bench \
  --threads 1,4 --repeats 3 --bin tract=target/release/lege \
  --bin rten=target/rt-rten/release/lege --bin ort=target/rt-ort/release/lege
python3 spikes/runtimes/threads_check.py --models MODELS --out threads --bin ...
docs/reports/inference-runtimes/build.sh
```

The GitHub Actions workflow `.github/workflows/spike-runtimes.yml` runs the same steps on Linux and under emulation, and `.github/workflows/spike-ort-official.yml` runs the thread-count check with Microsoft's official ONNX Runtime build. To run that check locally, point `ORT_DYLIB_PATH` at the official library first.