Morphing Browser I: A Multi-Engine macOS Browser for In-Application JavaScript Engine Benchmarking

Shyamal Suhana Chandra
Sapana Micro Software

June 2026

Abstract

Modern browser engines are optimized under different constraints: operating-system integration, security process models, JIT policy, graphics stacks, and automation protocols. Application developers rarely get a simple way to compare these engines inside one native desktop shell under one user workflow. Morphing Browser I is a macOS SwiftUI browser prototype that embeds or controls three JavaScript execution paths: Nitro through WebKit and JavaScriptCore, V8 through Chromium Embedded Framework (CEF), and SpiderMonkey through a bundled Firefox ESR runtime driven headlessly through Marionette. The system loads a requested origin, benchmarks each engine over repeated runs, records load time, JavaScript execution time, resident memory, and a weighted composite score, then stores the preferred engine per origin using SwiftData. This paper presents the architecture, algorithm, build strategy, preliminary benchmark observations, limitations, and future work for a browser that can morph its engine choice around measured site behavior.

1 Introduction

Browser monocultures simplify implementation but hide engine-specific behavior from users and developers. A page that performs well in one engine may stress a different engine's parser, JIT compiler, networking stack, or memory allocator. Safari's WebKit, Chromium's Blink and V8 stack, and Firefox's Gecko and SpiderMonkey stack each expose different integration models on macOS. Morphing Browser I asks a practical industry question: can a native macOS browser choose a rendering and JavaScript engine per site using local benchmarks rather than fixed policy?

The prototype is not a replacement for a full conformance suite or a scientific browser benchmark lab. Instead, it is an engineering harness inside a usable browser. The user enters a URL. The browser shows a WebKit-backed tab immediately, benchmarks all configured engines in background windows or headless sessions, selects the highest scoring engine for the origin, reloads the tab with that engine, and caches the decision.

The design has three goals:

1. **Comparable in-app measurements.** Each engine is exercised through the same URL and the same JavaScript workload.
2. **Native user experience.** Benchmarking should not block immediate browsing; Nitro acts as the fallback while measurements run.
3. **Reproducible local setup.** Large generated browser runtimes are excluded from git and installed by setup scripts.

2 Background

2.1 JavaScript Engines and Embedding Models

Morphing Browser I maps three engine labels to concrete macOS integration paths:

- **Nitro (JavaScriptCore).** The WebKit path uses `WKWebView`, Apple's supported native view for embedding web content and evaluating JavaScript in app-controlled pages [1].

- **V8 (Chromium).** The Chromium path uses CEF. CEF supports a browser process plus helper subprocesses, and its settings allow applications to specify a separate subprocess executable through `browser_subprocess_path` [2, 3].
- **SpiderMonkey (Gecko).** The Gecko path uses a bundled Firefox ESR runtime. Benchmarking is headless and controlled through Marionette, Mozilla's remote protocol for instrumenting Gecko-based browsers [4].

These paths are intentionally asymmetric. WebKit is a first-class system framework. CEF is an embeddable Chromium distribution with its own subprocess, framework, signing, and message-loop requirements. Firefox/Gecko does not provide the same stable, small, in-process embedding surface for arbitrary macOS apps, so this prototype separates visible helper rendering from headless SpiderMonkey measurement.

2.2 Why Benchmark In-App?

Traditional suites such as JetStream 3 measure broad JavaScript and WebAssembly performance across many workloads and report aggregate scores [6, 7]. Morphing Browser I uses a smaller benchmark because its goal is not to replace JetStream. Its goal is local engine selection for the currently requested origin. The benchmark therefore combines site load time, a deterministic JavaScript computation, and memory use into a per-origin preference.

This distinction matters. Industry benchmark suites are better for comparing browser releases. In-app measurements are better for making a product decision at the moment a user opens a site.

3 System Architecture

The implementation is a SwiftUI macOS application organized around a common `BrowserEngineSession` interface. The app shell owns tabs, address-bar state, history, bookmarks, downloads, and engine statistics. Each tab is controlled by a `TabSessionController`, which can swap from one engine session to another when the benchmark winner changes.

3.1 WebKit Path

`WebKitEngineSession` owns a `WKWebView`, attaches it to the selected tab's `NSView`, observes progress, title, and URL changes, and uses `evaluateJavaScript` for the benchmark script. The implementation also exposes cookie listing, cookie clearing, downloads, and developer-tool hooks for the WebKit-backed path.

3.2 CEF Path

The V8 path has two pieces. Swift code calls `CEFEngineSession`, which delegates rendering and script execution to `CEFBridgeHostView`. The Objective-C++ bridge initializes CEF, creates the browser view, configures cache directories, disables unstable GPU paths for this prototype, builds the helper app during the Xcode build phase, and signs embedded code.

The current JavaScript result path uses explicit CEF inter-process communication. The helper registers a renderer-side V8 function named `__hyperBrowserJavaScriptResult`. The browser process wraps the benchmark script, gives it a token, and waits for the helper to send a `CefProcessMessage` named `HyperBrowserJavaScriptResult`. This replaced an earlier title-polling approach that could time out under CEF's subprocess model.

3.3 Gecko Path

`GeckoHeadlessBenchmarkRunner` starts the bundled Firefox executable with `--headless`, `--marionette`, an isolated profile, and a random local port. A small Swift Marionette client opens a TCP socket, reads the Marionette hello packet, creates a WebDriver session, navigates to the URL, executes the benchmark script, records RSS memory with `ps`, and deletes the session.

4 Benchmark Algorithm

The benchmark runner uses a fixed number of repeated runs and chooses medians to reduce single-run noise. The current implementation uses three runs per engine.

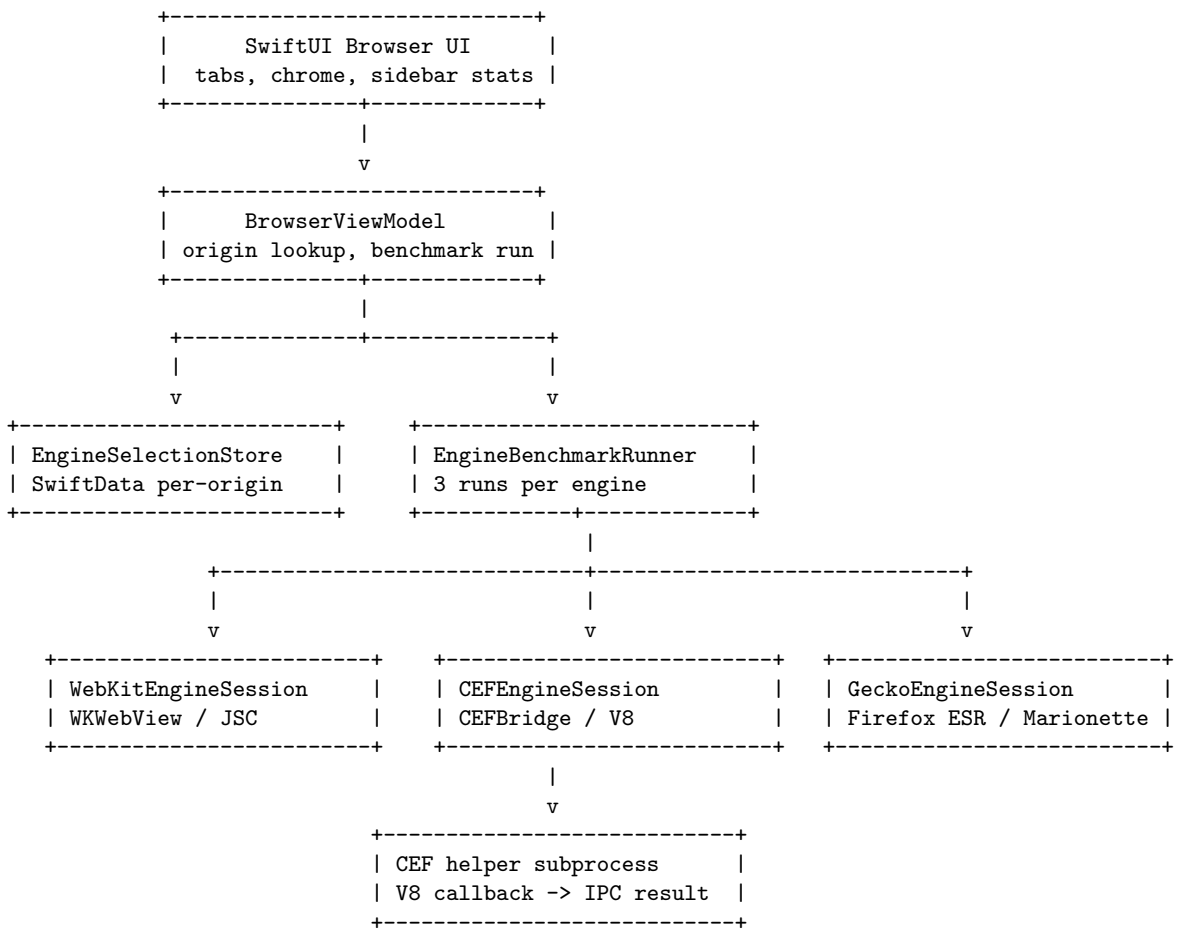

Figure 1: Morphing Browser I system diagram.

4.1 Measurement Inputs

For every engine and target URL, the system measures:

- **Page-load latency** in milliseconds, measured from the call to load or navigate until the engine-specific load completion condition.
- **JavaScript execution latency** in milliseconds, measured inside the page with `performance.now()` around a deterministic loop of 250,000 math operations.
- **Memory** in megabytes, using JavaScript heap data when available for WebKit, process RSS for Gecko, and host process memory plus an engine allowance for CEF.

4.2 Scoring

Lower raw metrics are better. Morphing Browser I converts each metric into a 0–100 component and applies fixed weights:

$$S_{load} = \max(0, 100 - \min(loadMs/50, 100)) \tag{1}$$

$$S_{js} = \max(0, 100 - \min(jsMs/20, 100)) \tag{2}$$

$$S_{mem} = \max(0, 100 - \min(memoryMB/5, 100)) \tag{3}$$

$$S = 0.40S_{load} + 0.35S_{js} + 0.25S_{mem} \tag{4}$$

The winner is the engine with the maximum composite score. If no result is available, Nitro is the fallback.

4.3 Pseudocode

```
function selectEngineForOrigin(url):
    origin = normalizedHost(url)
    cached = lookupPreference(origin)
    if cached exists:
        load(url, cached)
        return cached

    load(url, Nitro)  // immediate user-visible fallback
    results = {}
    failures = {}

    for engine in [Nitro, V8, SpiderMonkey]:
        runs = []
        for i in 1..3:
            session = makeSession(engine)
            try:
                runs.append(session.runBenchmark(url))
            catch error:
                failures[engine] = error
                break
            finally:
                session.detach()

        if runs not empty:
            results[engine] = medianByMetric(runs)

    winner = maxCompositeScore(results) or Nitro
    savePreference(origin, winner, results[winner])
    reloadVisibleTab(url, winner)
    return winner
```

5 Benchmarks

5.1 Methodology

The current benchmark should be read as a product-selection heuristic, not a publication-grade browser shootout. It runs on the user's macOS machine, uses the same application shell for each engine, and measures the current target page. The method intentionally includes page-load overhead because the product decision is about user-perceived site behavior, not only raw JavaScript throughput.

The benchmark has three useful properties:

1. It runs on the real site URL, not a synthetic blank page.
2. It repeats each engine three times and takes medians.
3. It records failures as first-class results rather than hiding them.

It also has known limitations:

1. The JavaScript workload is a small deterministic microbenchmark, not a full suite like JetStream.
2. Memory measurement is not perfectly normalized across engines because embedding models differ.

3. Network state, cache state, and page nondeterminism can dominate single-site runs.

4. Gecko benchmarking is headless, while user-visible browsing uses a helper-rendered view.

5.2 Preliminary Smoke Benchmark

During development, the app was exercised on `www.apple.com`. The UI captured the following preliminary values before the final CEF IPC fix was added:

Engine	Load ms	JS ms	Memory MB	Score	Status
Nitro (JavaScriptCore)	361	2	0.0	97.1	Completed
V8 (Chromium)	–	–	–	–	Timed out waiting for CEF JavaScript result
SpiderMonkey (Gecko)	898	0	511.9	67.8	Completed

This table is useful because it exposed a system bug: V8 page readiness depended on JavaScript completion, and completion was implemented through a fragile title-token side channel. The fix changed CEF JavaScript evaluation to use renderer-to-browser IPC. After the fix, the project builds successfully with `xcodebuild` in Debug configuration. A future benchmark report should re-run the same site after clearing caches and should include V8 values from the corrected IPC path.

6 Engineering Challenges

6.1 CEF Process Model

CEF is designed around multiple processes. Earlier single-process attempts triggered Chromium failures such as V8 proxy resolver errors. The final design restores a helper subprocess and points `CefSettings.browser_subprocess_path` at a helper app built during the Xcode copy phase. This matches the documented CEF model for separate subprocess executables [2, 3].

6.2 JavaScript Result Transport

`ExecuteJavaScript` schedules execution but does not, by itself, provide a direct Swift result channel equivalent to `WKWebView`'s completion handler. The bridge now creates its own result channel:

1. Browser process generates a token.

2. Browser process wraps the user script.

3. Renderer executes the script.

4. Renderer calls a native V8 function with token, result, and error.

5. Helper sends `CefProcessMessage` to the browser process.

6. Swift continuation resumes with result or error.

This is the most important stability change in the V8 path.

6.3 Code Signing and Runtime Payloads

Large CEF and Firefox runtimes are generated locally. The repository commits setup scripts and README files, not the full downloaded payloads. At build time, CEF is copied into `Contents/Frameworks`, the helper app is compiled, and embedded code is signed. Firefox ESR is copied into `Contents/Resources/Engines` when present.

7 Future Work

1. **Add a benchmark suite mode.** Support JetStream-style workloads or import a fixed local benchmark corpus for broader JavaScript and WebAssembly coverage.
2. **Normalize memory accounting.** Use per-process tree accounting for CEF and Gecko so subprocess memory is included consistently.
3. **Control cache and network state.** Add warm-cache and cold-cache modes, DNS/network timing, and per-origin cache isolation.
4. **Persist benchmark provenance.** Store hardware model, OS version, app build, engine runtime versions, URL, timestamp, cache mode, and failure text.
5. **Improve Gecko embedding.** Investigate deeper GeckoView-style or Servo-style embedding options if stable macOS integration points become practical.
6. **Add confidence intervals.** Increase runs beyond three, compute variance, and avoid switching engines when scores are statistically close.
7. **Expose reproducible reports.** Export benchmark summaries to CSV, JSON, and PDF so engine decisions can be audited.
8. **Harden privacy boundaries.** Make clear which benchmark data is local, avoid uploading URLs by default, and isolate profiles per origin.

8 Conclusion

Morphing Browser I demonstrates a practical architecture for per-origin browser engine selection inside a native macOS app. The prototype combines WebKit, CEF, and Firefox ESR behind a common Swift interface, benchmarks each engine with repeated local measurements, records failures explicitly, and stores the preferred engine per origin. The main engineering lesson is that browser engines cannot be treated as interchangeable libraries. Each engine brings a process model, signing model, JavaScript evaluation path, and measurement semantics. A useful morphing browser must respect those boundaries while presenting a simple user-facing decision: which engine works best for this site on this machine?

References

[1] Apple Developer Documentation. *WKWebView*. https://developer.apple.com/documentation/webkit/wkwebview

[2] Chromium Embedded Framework. *General Usage*. https://chromiumembedded.github.io/cef/general_usage.html

[3] Chromium Embedded Framework. *cef_settings_t Struct Reference*. https://cef-builds.spotifycdn.com/docs/147.0/structcef__settings__t.html

[4] Mozilla Firefox Source Docs. *Marionette*. https://firefox-source-docs.mozilla.org/testing/marionette/

[5] Mozilla Firefox Source Docs. *Introduction to Marionette*. https://firefox-source-docs.mozilla.org/testing/marionette/Intro.html

[6] BrowserBench. *JetStream 3.0 In-Depth Analysis*. https://browserbench.org/JetStream/in-depth.html

[7] WebKit. *Introducing the JetStream 3 Benchmark Suite*. https://webkit.org/blog/17899/introducing-the-jetstream-3-benchmark-suite/

[8] Chromium Project. *Run JavaScript Code*. https://chromium.googlesource.com/experimental/chromium/src/+/refs/tags/87.0.4280.127/chrome/test/chromedriver/docs/run_javascript.md