

# 온프레미스 리눅스 AI 에이전트 런타임 비교: Node.js·Bun·Deno와 Rust 네이티브 바이너리의 기동·메모리·CPU·지연 측정

임재호

NextgenAI

khan@nextgenai.kr

## 초록

온프레미스 리눅스에서 AI 에이전트를 돌릴 런타임으로 Node.js 계열과 Rust 네이티브 바이너리 중 무엇이 나은지에 대한 측정 근거는 부족하다. 기존 언어 비교 연구는 CPU 바운드 벤치마크를 다루며, LLM 응답 대기가 지배하는 에이전트 루프는 다루지 않았다. 이 연구는 대화 이력 전송, SSE 스트림 파싱, 도구 서브프로세스 실행으로 이루어진 같은 에이전트 루프를 JavaScript 한 벌(Node.js 24, Bun 1.4, Deno 2.9에서 실행)과 Rust 한 벌(tokio 다중·단일 스레드)로 구현하고, 모의 LLM 서버로 서버 시간을 분리해 aarch64 리눅스 컨테이너에서 측정했다. 1세션 기동 시간은 Rust 3.8 ms, Node.js 35.6 ms였다. 동시 500세션에서 Rust의 최대 RSS는 159.7 MB로 Bun 383.9 MB, Node.js 775.8 MB보다 작았고, 오케스트레이션 오버헤드 중앙값은 Rust(다중 스레드) 2.24 ms, Bun 174.09 ms, Node.js 299.58 ms였다. 세션 10개 수준에서는 차이가 작았다. musl 정적 바이너리는 1.44 MiB로 JavaScript 런타임 바이너리보다 50배 이상 작았다. 결과는 가상 머신 한대에서 얻은 것이며 x86 베어메탈에서의 재현은 후속 과제다.

**주제어:** AI 에이전트; 런타임; Node.js; Rust; 온프레미스

## 1. 서론

AI 에이전트는 LLM 호출, 도구 실행, 결과 누적을 반복하는 장시간 상주 프로세스다. 온프레미스 리눅스 서버에 에이전트를 배포하는 조직은 이 루프를 어떤 런타임으로 돌릴지 정해야 한다. 산업계 선택은 엇갈린다. OpenAI는 TypeScript로 작성한 Codex CLI를 Rust로 다시 작성하면서 설치 의존 제거와 메모리 감소를 이유로 들었지만 수치는 공개하지 않았다 [1]. Anthropic의 Agent SDK는 TypeScript와 Python으로 제공되면서 실행 단위로는 세션마다 CLI 서브프로세스를 띄운다 [2].

기존 언어 비교 연구는 이 선택에 바로 쓰기 어렵다. 대표 연구는 CPU 바운드 벤치마크를 다루고 [3], 후속 연구는 측정 방법을 통제하면 에너지 차이가 실행시간과 코어 사용으로 환원된다고 반박했다 [4]. 에이전트 루프는 대부분의 시간을 LLM 응답을 기다리며 보낸다. 이런 I/O 바운드 작업에서 런타임 선택이 지연, 메모리, CPU를 얼마나 바꾸는지는 측정된 적이 없다.

이 연구의 질문은 다음과 같다. *LLM* 대기가 지배하는 에이전트 루프를 리눅스에서 실행할 때, *Node.js·Bun·Deno*와 *Rust* 네이티브 바이너리는 기동 시간, 메모리, *CPU* 시간, 오케스트레이션 오버헤드에서 얼마나 다르며, 그 차이는 동시 세션 수에 따라 어떻게 변하는가? 이 연구의 기여는 다음과 같다.

- 같은 프로토콜과 루프를 JavaScript 한 벌(세 런타임 공용)과 Rust 한 벌로 구현하고, 모의 LLM 서버로 서버 측 시간을 분리하는 측정 방법을 제시한다(절 3).
- 동시 세션 1~500에서 메모리, 오버헤드, CPU 시간을 측정해, 차이가 동시성에 따라 커지며 단일 스레드 Rust도 JavaScript 런타임보다 작은 오버헤드를 보임을 보고한다(표 3, 그림 3, 그림 4).

- 기동 시간, 장시간 상주 RSS, 배포 산출물 크기를 함께 보고하고, 사전에 정한 세 가설의 채택과 기각을 정리한다(표 2, 표 4, 표 5, 절 5).

## 2. 관련 연구

### 2.1 언어·런타임의 성능과 에너지

Pereira et al. [3]은 Computer Language Benchmarks Game의 CPU 바운드 프로그램으로 27개 언어의 에너지·시간·메모리를 측정했다. C를 1.00으로 정규화하면 Rust는 에너지 1.03, 시간 1.04였고 JavaScript는 에너지 4.45, 시간 6.52였다. van Kempen et al. [4]은 이 계열 연구의 측정 방법을 다시 검토했다. 언어와 구현을 분리하고 활성 코어 수와 메모리 활동을 통제하자 언어 구현의 선택은 실행시간 이상의 에너지 차이를 만들지 않았다. 이 결론은 에너지를 직접 비교하기보다 실행시간과 CPU 사용량을 측정하는 쪽이 타당하다는 근거가 된다. JavaScript 런타임 사이의 차이도 작지 않다. Merelo-Guervós et al. [5]는 진화 알고리즘 작업에서 Bun이 Node.js보다 에너지를 3배에서 6배 적게 썼다고 보고했다. 따라서 이 연구는 “JavaScript 런타임”을 하나로 묶지 않고 Node.js, Bun, Deno를 따로 측정한다.

### 2.2 가비지 컬렉션 런타임의 메모리

V8 같은 가비지 컬렉션(GC) 런타임에서 힙 한도는 메모리 사용량과 GC 시간을 맞바꾼다. Kirisame et al. [6]은 힙 한도를 제공된 규칙으로 정해 브라우저 워크로드의 메모리를 GC 시간 증가 없이 줄였다. 이 연구는 브라우저를 대상으로 했으며 장시간 상주하는 서버형 에이전트 프로세스의 상주 메모리(RSS)는 측정하지 않았다.

### 2.3 메모리 안전성과 공급망

Google은 Android의 Rust 코드에서 메모리 안전 취약점 밀도를 0.2/MLOC로 추정하고, 과거 C/C++ 코드의 약 1,000/MLOC와 비교했다 [7]. 이 수치는 동료 심사를 거치지 않은 업체 자체 보고이며 비교 대상은 C/C++이다. Rust의 안전성에도 경계가 있다. Cui et al. [8]는 2023년 7월까지 “Rust” 키워드로 검색된 CVE 404건 중 196건이 unsafe 코드 오용에서 비롯됐다고 보고했다. JavaScript 쪽의 위험은 공급망에 있다. Zimmermann et al. [9]에 따르면 2018년 시점에 npm 패키지 하나를 설치하면 평균 79개의 서드파티 패키지와 39명의 관리자를 암묵적으로 신뢰한다. Ohm et al. [10]가 수집한 악성 패키지 174개 중 62.6%가 npm이었고, 56%는 설치 스크립트로 실행됐다. Rust의 build.rs와 절차적 매크로도 빌드 시점에 임의 코드를 실행하므로, 공급망 위험은 어느 한쪽만의 문제가 아니다.

### 2.4 AI 에이전트 런타임 사례

산업계 에이전트 도구는 두 방향을 보인다. OpenAI는 TypeScript로 작성한 Codex CLI를 Rust로 다시 작성하면서, Node.js 설치 의존 제거, 리눅스 샌드박스 바인딩, GC가 없는 데 따른 메모리 감소를 이유로 들었다 [1]. 이 발표는 정량 수치를 제시하지 않았다. Anthropic의 Agent SDK는 TypeScript와 Python으로 제공되지만 세션마다 별도의 CLI 서브프로세스를 띄우며, 대부분의 설치에서 네이티브 CLI 바이너리를 번들한다 [2]. 도구 실행 격리는 언어 런타임이 아니라 OS의 bubblewrap과 seatbelt로 한다 [11]. 즉 격리 경계는 언어 선택과 독립적이며, 런타임 선택이 실제로 바꾸는 것은 기동 시간, 메모리, CPU, 배포 단위다.

### 2.5 연구 공백

앞의 성능·에너지 연구는 모두 CPU 바운드 마이크로벤치마크를 다룬다. LLM 응답 대기 지배하는 I/O 바운드 에이전트 루프에서 런타임별 오버헤드, 동시 세션 수에 따른 메모리 증가, 장시간 상주 시 RSS 추이를 같은 조건에서 측정한 연구는 찾지 못했다. 이 연구는 이 공백을 측정으로 채운다.

### 3. 연구 방법

#### 3.1 작업 모델

이 연구는 에이전트 한 세션을 알고리즘 1의 루프로 모델링한다. 세션은 대화 이력 전체를 JSON으로 직렬화해 모의 LLM 서버에 보내고, 서버-전송 이벤트(SSE) 스트림을 받아 텍스트 조각을 이어 붙인다. 응답 마지막 이벤트가 도구 호출이면 서브프로세스로 도구를 실행하고 출력을 이력에 더한 뒤 다음 턴으로 간다. 한 세션은  $K = 10$  턴이며 앞의 9턴은 도구를 호출하고 마지막 턴은 종료한다. 도구는 16 KiB 텍스트 파일을 `cat`으로 읽는다. 따라서 마지막 요청의 본문은 약 150 KiB까지 커진다. 실제 LLM API 대신 모의 서버를 쓴 이유는 네트워크와 모델 지연의 변동을 없애고 런타임 쪽 비용만 남기기 위해서다. 전체 구성은 그림 1에 보인다.

---

#### 알고리즘 1 세션 루프(모든 구현에 공통)

---

**Require:** 서버 주소  $u$ , 턴 수  $K$

```

1:  $H \leftarrow$  [사용자 메시지]
2: for  $k = 0, \dots, K - 1$  do
3:    $t_0 \leftarrow$  현재 시각
4:   본문  $\{k, K, H\}$ 를 JSON으로 직렬화해  $u$ 에 POST 한다
5:   SSE 이벤트를 파싱해 텍스트를 모으고 마지막 이벤트  $e$ 를 얻는다
6:    $o_k \leftarrow (\text{현재 시각} - t_0) - e.\text{server\_ms}$  ▷ 식 (1)
7:    $H$ 에 응답 텍스트를 추가한다
8:   if  $e$ 가 종료 이벤트 then
9:     break
10:  end if
11:  서브프로세스로 도구를 실행하고 표준 출력을  $H$ 에 추가한다
12: end for

```

---

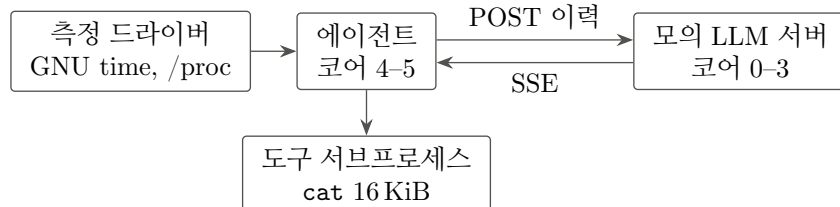


**그림 1.** 측정 구성. 모든 구성 요소는 같은 리눅스 컨테이너에서 실행되며 서버와 에이전트는 서로 다른 CPU 코어에 고정된다.

#### 3.2 비교 대상

비교 대상은 표 1의 다섯 구성과 단일 실행 파일 세 가지다. JavaScript 구현은 TypeScript 원본 한 벌을 `bun build`로 2,418 B의 JavaScript 파일로 변환해 세 런타임에서 그대로 실행한다. 외부 패키지는 쓰지 않으며 HTTP는 각 런타임의 내장 `fetch`, 서브프로세스는 `node:child_process`의 `execFile`을 쓴다. Rust 구현은 `tokio`와 `reqwest`를 쓰고 같은 프로토콜과 루프를 따른다. Rust는 스케줄러 두 가지를 비교한다. 다중 스레드(`rust-mt`)는 `tokio` 기본값이고, 단일 스레드(`rust-st`)는 JavaScript의 단일 이벤트 루프와 같은 조건에서 비교하기 위한 변형이다. 측정 코드와 원자료는 `agent-runtime-bench` 디렉터리에 있다.

표 1. 비교 대상 구성과 버전(컨테이너 안에서 기록)

| 이름              | 런타임·빌드                     | 실행 형태            |
|-----------------|----------------------------|------------------|
| node            | Node.js v24.21.0           | agent.js         |
| bun             | Bun 1.4.2                  | agent.js         |
| deno            | Deno 2.9.7                 | agent.js, -A     |
| rust- <i>mt</i> | rustc 1.99.0, tokio 다중 스레드 | glibc 동적 링크 바이너리 |
| rust- <i>st</i> | rustc 1.99.0, tokio 단일 스레드 | glibc 동적 링크 바이너리 |
| bun-compiled    | bun build --compile        | 단일 실행 파일         |
| deno-compiled   | deno compile               | 단일 실행 파일         |
| rust-musl       | musl 대상, 다중 스레드            | 정적 링크 바이너리       |

### 3.3 지표

턴  $k$ 의 오케스트레이션 오버헤드  $o_k$ 는 클라이언트가 켜진 시간에서 서버가 요청 본문을 받은 뒤 쓴 시간  $s_k$ 를 뺀 값이다.

$$o_k = (t_k^{\text{end}} - t_k^{\text{start}}) - s_k \quad (1)$$

$o_k$ 에는 요청 직렬화, 연결과 전송, SSE 파싱, 그리고 다른 세션 때문에 이벤트 루프나 스케줄러에서 기다린 시간이 들어간다. 도구 지연은 서브프로세스 생성부터 표준 출력을 모두 받을 때까지의 시간이다. CPU 시간은 GNU `time`이 보고한 사용자 시간과 시스템 시간의 합을 전체 턴 수  $S \cdot K$ 로 나눈 값이며, 도구 서브프로세스의 CPU 시간을 포함한다. 최대 RSS는 GNU `time`의 최대 상주 집합 크기다. 에너지는 측정하지 않았다. 가상 머신에서는 RAPL 카운터를 읽을 수 없고, van Kempen et al. [4]에 따르면 에너지 차이는 실행시간과 코어 사용으로 대부분 설명되므로 CPU 시간을 대리 지표로 쓴다.

### 3.4 실험 환경과 절차

호스트는 Apple M3 Pro(12코어, 36 GiB) macOS 기기이며, Docker Desktop 29.8.1의 리눅스 가상 머신(커널 7.0.14-linuxkit, aarch64, vCPU 12개, 메모리 약 15.6 GiB) 안에서 Debian bookworm-slim 컨테이너로 측정했다. 모의 서버는 `taskset`으로 코어 0–3에, 에이전트는 코어 4–5에 고정했다. 모의 서버는 Rust(axum)로 작성했고 첫 토큰까지 50 ms를 기다린 뒤 50개 조각을 2 ms 간격으로 보낸다. 설정상 서버 시간은 턴당 약 150 ms다. 실험은 네 가지다.

- **E1 기동**: 서버 지연을 0으로 두고 1세션 1턴을 실행한 프로세스의 벽시계 시간과 최대 RSS. 예열 3회 뒤 30회 반복했다.
- **E2 동시성**: 동시 세션  $S \in \{1, 10, 100, 500\}$ , 세션당 10턴. 구성마다 5회 반복했고, 반복마다 다섯 구성을 번갈아 실행해 호스트 상태 변화가 모든 구성에 고르게 미치게 했다.
- **E3 상주**: 한 프로세스가 50세션 묶음을 150회 연속 처리하는 동안 1초마다 `/proc/<pid>/status`의 RSS를 기록했다. 구성마다 1회 실행했다.
- **E4 배포**: 실행에 필요한 런타임 바이너리, 프로그램, 단일 실행 파일의 디스크 크기와 공식 컨테이너 이미지 크기.

E1과 E2는 중앙값을 보고하고, 반복 간 범위(최솟값–최댓값) 또는 분위수를 함께 적는다. 지연의 p99는 모든 반복의 턴 표본을 합쳐 계산했다. 통계적 유의성 검정은 하지 않았다.

이 연구는 사전에 세 가설과 기각 조건을 정했다.

**가정 1 (H1)**. 오케스트레이션 오버헤드의 중앙값은 시험한 모든  $S$ 에서 설정 서버 시간(150 ms)의 5%, 즉 7.5 ms 미만이다. 어느 구성이든 7.5 ms 이상이면 그 구성과  $S$ 에서 기각한다.

가정 2 (H2).  $S \geq 100$ 에서 Rust 구성의 최대 RSS는 JavaScript 런타임보다 작다.

가정 3 (H3). 턴당 CPU 시간은 시험한 모든  $S$ 에서 Rust 구성이 가장 작다.

## 4. 실험 결과

### 4.1 E1: 기동 시간과 기본 메모리

표 2과 그림 2은 1세션 1턴을 실행하고 종료하는 데 걸린 시간이다. 중앙값은 rust-musl 3.6 ms, rust-st 3.7 ms, rust-mt 3.8 ms, bun 7.0 ms, deno 22.9 ms, node 35.6 ms였다. node는 rust-mt보다 9.3배 길었다(35.6/3.8). 단일 실행 파일로 묶어도 기동 시간은 원래 런타임과 비슷했다(bun-compiled 6.8 ms, deno-compiled 19.1 ms). 최대 RSS는 Rust가 1.3–3.0 MB, Bun이 19.0–20.2 MB, Deno가 40.3–44.2 MB, Node.js가 61.4 MB였다.

표 2. E1 기동 결과. 1세션 1턴, 서버 지연 0, 30회 반복의 중앙값(p10–p90).

| 구성            | 벽시계 시간 (ms)      | 최대 RSS (MB) |
|---------------|------------------|-------------|
| node          | 35.6 (34.7–36.8) | 61.4        |
| bun           | 7.0 (6.7–7.2)    | 20.2        |
| deno          | 22.9 (22.1–23.9) | 40.3        |
| rust-mt       | 3.8 (3.6–3.9)    | 3.0         |
| rust-st       | 3.7 (3.5–3.8)    | 2.6         |
| bun-compiled  | 6.8 (6.5–7.0)    | 19.0        |
| deno-compiled | 19.1 (18.9–19.7) | 44.2        |
| rust-musl     | 3.6 (3.4–3.8)    | 1.3         |

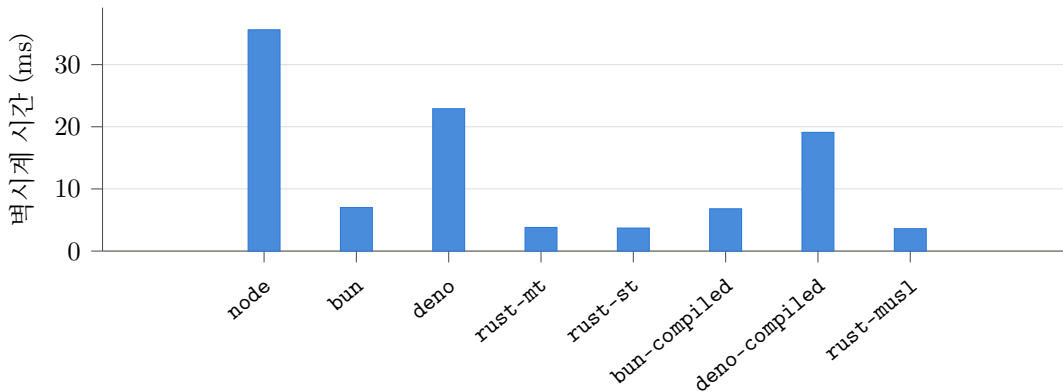


그림 2. E1 기동 벽시계 시간의 중앙값(30회, 표 2).

### 4.2 E2: 동시 세션 수에 따른 변화

표 3은  $S$ 별 전체 결과다. 모든 실행에서 기대한 턴 수( $S \cdot 10$ )의 표본이 기록됐다.

**메모리.** 최대 RSS는 모든 구성에서  $S$ 와 함께 커졌다(그림 3).  $S = 500$ 에서 중앙값은 rust-mt 159.7 MB, rust-st 167.5 MB, bun 383.9 MB, node 775.8 MB, deno 795.2 MB였다.  $S = 100$ 에서  $S = 500$ 으로 늘 때 늘어난 RSS를 추가된 400세션으로 나누면 세션당 증가분은 rust-mt 0.31 MB, rust-st 0.33 MB, bun 0.66 MB, node 1.07 MB, deno 1.41 MB다.

**오케스트레이션 오버헤드.**  $S = 1$ 에서 오버헤드 중앙값은 모든 구성에서 3.1 ms 이하였다(그림 4).  $S$ 가 커지자 JavaScript 런타임의 오버헤드가 먼저 늘었다.  $S = 100$ 에서 node 43.12 ms, bun

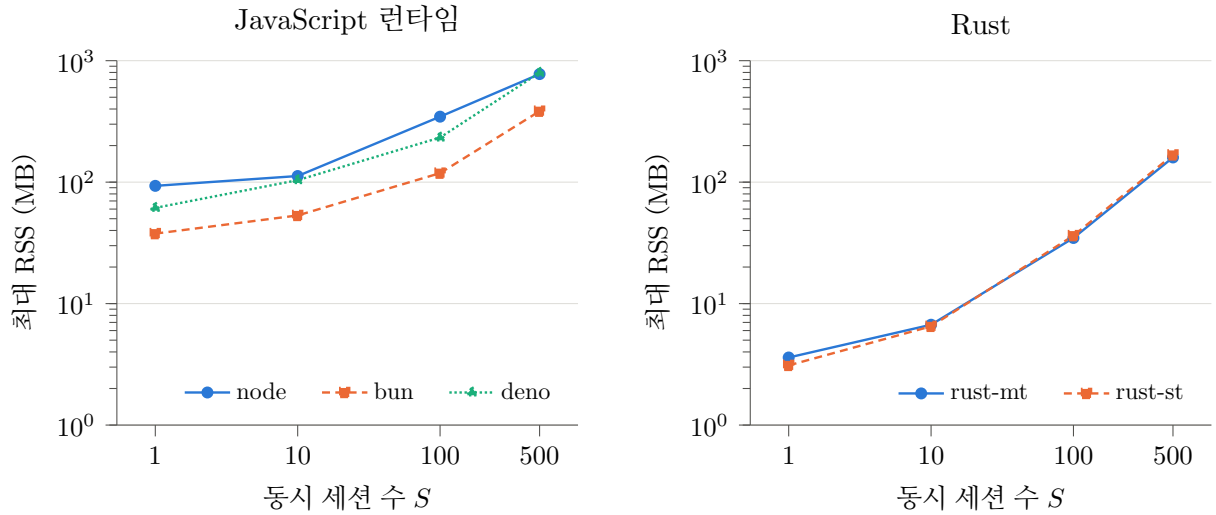


그림 3. 동시 세션 수에 따른 최대 RSS의 중앙값(5회, 표 3). 두 패널은 같은 로그 축을 쓴다.

12.03 ms, deno 7.58 ms였고,  $S = 500$ 에서는 node 299.58 ms, bun 174.09 ms, deno 175.25 ms였다. 같은  $S = 500$ 에서 Rust는 rust-mt 2.24 ms, rust-st 6.26 ms였다. 꼬리 지연도 같은 순서였다.  $S = 500$ 에서  $o$ 의 p99는 node 1012.81 ms, bun 294.43 ms, deno 291.05 ms, rust-mt 91.44 ms, rust-st 118.92 ms였다. 도구 지연 중앙값은  $S = 500$ 에서 node 352.22 ms, bun 48.26 ms, deno 17.04 ms, Rust 2.0 ms 이하였다.

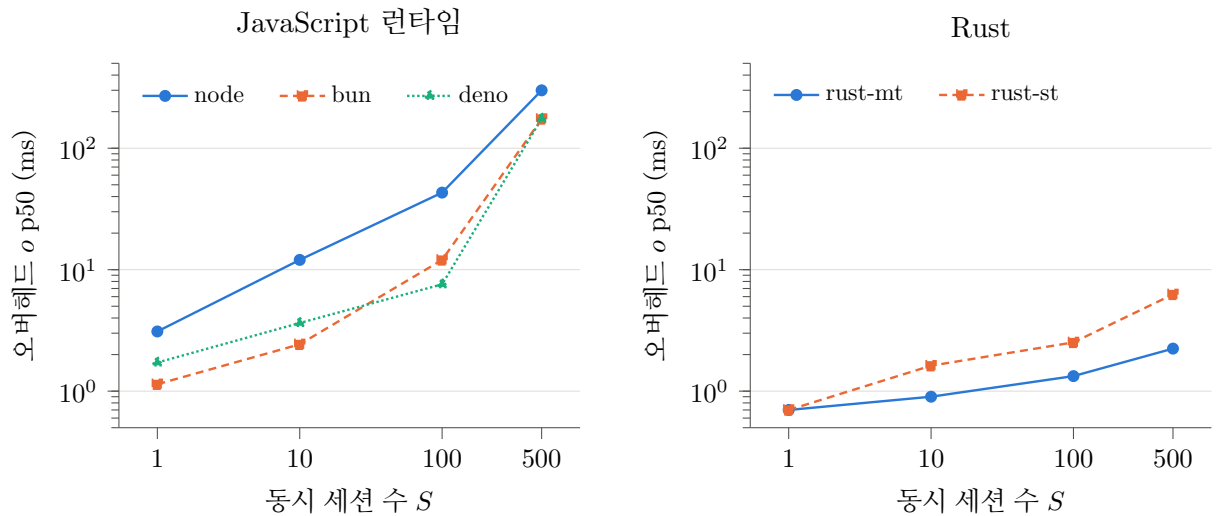


그림 4. 동시 세션 수에 따른 오케스트레이션 오버헤드 중앙값(표 3). 두 패널은 같은 로그 축을 쓴다.

**CPU 시간과 처리 시간.** 톤당 CPU 시간은 모든  $S$ 에서 Rust 구성이 가장 작았다(표 3).  $S = 500$ 에서 rust-st 0.43 ms, rust-mt 0.46 ms, bun 0.78 ms, deno 1.01 ms, node 2.06 ms였다(그림 5).  $S = 1$ 의 값은 기동 비용이 10턴에 나뉘어 들어가 크게 나타나며, GNU time의 10 ms 해상도 때문에 정밀하지 않다. 500세션 묶음 전체를 끝내는 벽시계 시간은 Rust 두 구성이 2.44–2.53 s로  $S = 1$ 과 거의 같았고, deno 4.35 s, bun 4.72 s, node 9.60 s였다.

#### 4.3 E3: 장시간 상주

표 4은 50세션 묶음을 150회 연속 처리하는 동안의 RSS다. 30초 이후 표본의 중앙값은 rust-st 21.1 MB, rust-mt 23.0 MB, bun 76.1 MB, deno 175.5 MB, node 342.8 MB였다(그림 6). 최소제공

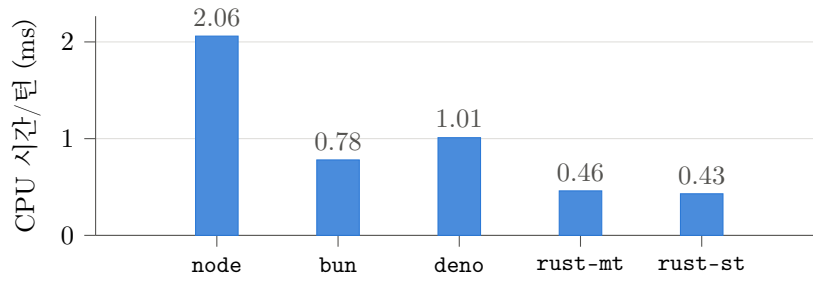


그림 5.  $S = 500$ 에서 턴당 CPU 시간의 중앙값(5회, 표 3). 도구 서브프로세스의 CPU 시간을 포함한다.

표 3. E2 동시성 결과. 세션당 10턴, 구성· $S$ 마다 5회 반복. 벽시계·CPU·RSS는 반복 간 중앙값(최솟값-최댓값), 오버헤드  $o$ 와 도구 지연은 모든 반복의 턴 표본을 합친 분위수(ms).

| 구성      | $S$ | 벽시계 (s) | CPU/턴 (ms)          | 최대 RSS (MB)         | $o$ p50 | $o$ p99 | 도구 p50 | 도구 p99  |
|---------|-----|---------|---------------------|---------------------|---------|---------|--------|---------|
| node    | 1   | 2.51    | 41.00 (35.00–45.00) | 93.2 (93.0–94.8)    | 3.10    | 28.02   | 4.91   | 7.56    |
|         | 10  | 2.66    | 8.30 (7.90–9.10)    | 112.3 (109.5–119.2) | 12.04   | 33.82   | 10.52  | 32.35   |
|         | 100 | 3.32    | 2.20 (2.17–2.26)    | 346.0 (341.6–356.9) | 43.12   | 105.15  | 54.03  | 104.73  |
|         | 500 | 9.60    | 2.06 (1.97–2.19)    | 775.8 (722.3–799.4) | 299.58  | 1012.81 | 352.22 | 1021.02 |
| bun     | 1   | 2.45    | 20.00 (17.00–23.00) | 37.7 (37.7–37.8)    | 1.14    | 3.74    | 3.22   | 22.89   |
|         | 10  | 2.46    | 4.80 (4.60–5.40)    | 53.2 (51.3–53.5)    | 2.43    | 25.38   | 4.61   | 20.23   |
|         | 100 | 2.69    | 1.24 (1.21–1.26)    | 118.6 (115.7–122.7) | 12.03   | 50.13   | 22.56  | 41.35   |
|         | 500 | 4.72    | 0.78 (0.73–0.81)    | 383.9 (375.6–393.1) | 174.09  | 294.43  | 48.26  | 155.60  |
| deno    | 1   | 2.49    | 27.00 (23.00–31.00) | 61.4 (60.6–62.2)    | 1.72    | 5.75    | 3.40   | 22.08   |
|         | 10  | 2.53    | 6.90 (6.80–7.30)    | 103.8 (103.2–106.3) | 3.64    | 25.64   | 3.13   | 16.91   |
|         | 100 | 2.59    | 1.61 (1.54–1.62)    | 232.9 (222.1–247.4) | 7.58    | 72.43   | 5.73   | 47.58   |
|         | 500 | 4.35    | 1.01 (1.01–1.06)    | 795.2 (722.0–860.7) | 175.25  | 291.05  | 17.04  | 158.38  |
| rust-mt | 1   | 2.39    | 5.00 (4.00–5.00)    | 3.6 (3.5–3.8)       | 0.70    | 1.51    | 1.76   | 3.17    |
|         | 10  | 2.40    | 2.50 (2.00–2.80)    | 6.7 (6.6–6.8)       | 0.90    | 5.14    | 1.95   | 6.17    |
|         | 100 | 2.44    | 0.97 (0.97–1.05)    | 34.7 (33.0–36.2)    | 1.33    | 43.15   | 1.90   | 24.50   |
|         | 500 | 2.44    | 0.46 (0.44–0.47)    | 159.7 (147.7–160.6) | 2.24    | 91.44   | 1.97   | 38.97   |
| rust-st | 1   | 2.42    | 4.00 (4.00–6.00)    | 3.1 (3.1–3.1)       | 0.70    | 1.45    | 2.01   | 3.33    |
|         | 10  | 2.43    | 1.90 (1.70–2.10)    | 6.5 (6.3–6.8)       | 1.62    | 6.35    | 1.87   | 5.68    |
|         | 100 | 2.41    | 0.77 (0.75–0.88)    | 36.3 (35.5–38.8)    | 2.52    | 36.28   | 1.74   | 12.46   |
|         | 500 | 2.53    | 0.43 (0.42–0.44)    | 167.5 (166.2–168.0) | 6.26    | 118.92  | 1.93   | 85.10   |

기울기는 node 1.40 MB/분, deno 0.75 MB/분, rust-st 0.13 MB/분, rust-mt 0.03 MB/분, bun –0.03 MB/분이었다. node는 기울기가 양수였지만 30초 시점(339.4 MB)과 마지막 표본(339.5 MB)이 거의 같았고, 그 사이 최대 357.1 MB까지 오르내렸다. 같은 작업량을 끝내는 데 node는 438 s, 나머지 구성은 372–380 s가 걸렸다.

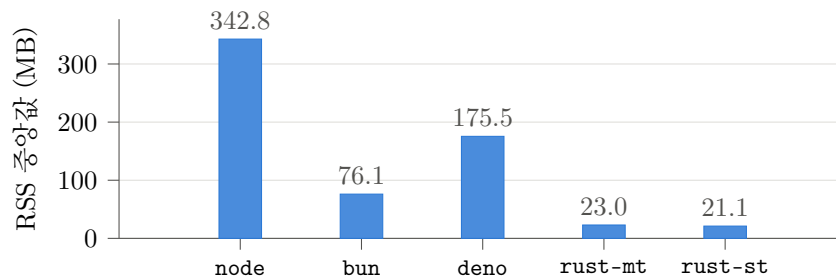


그림 6. E3에서 30초 이후 RSS 표본의 중앙값(구성마다 1회, 표 4).

**표 4.** E3 상주 결과. 50세션 묶음 150회 연속 처리, RSS 1초 간격 표본. 기울기는 30초 이후 표본의 최소제곱 직선.

| 구성      | 실행 (s) | RSS (MB) |       |       |       | 기울기 (MB/분) |
|---------|--------|----------|-------|-------|-------|------------|
|         |        | 30 s 시점  | 중앙값   | 최대    | 마지막   |            |
| node    | 438    | 339.4    | 342.8 | 357.1 | 339.5 | 1.40       |
| bun     | 379    | 89.2     | 76.1  | 95.6  | 81.8  | -0.03      |
| deno    | 380    | 177.5    | 175.5 | 189.7 | 179.0 | 0.75       |
| rust-mt | 372    | 22.8     | 23.0  | 29.2  | 29.2  | 0.03       |
| rust-st | 374    | 22.8     | 21.1  | 25.8  | 21.2  | 0.13       |

#### 4.4 E4: 배포 산출물

표 5 왼쪽은 에이전트를 실행하는 데 필요한 파일 크기다. JavaScript 프로그램 자체는 2,418 B 이고, 크기는 런타임 바이너리가 결정한다. Node.js는 117.20 MiB, Deno는 81.04 MiB, Bun은 75.74 MiB였다. 단일 실행 파일로 묶으면 bun-compiled 77.55 MiB, deno-compiled 100.13 MiB 였다. Rust 바이너리는 glibc 동적 링크 1.38 MiB, musl 정적 링크 1.44 MiB였고, musl 바이너리는 ldd에서 동적 실행 파일이 아니라고 보고됐다. 표 5 오른쪽의 컨테이너 이미지에서 가장 작은 공식 JavaScript 런타임 이미지는 oven/bun:1-distroless 146.44 MiB였고, musl 바이너리만 담은 scratch 이미지는 2.20 MiB였다.

**표 5.** E4 배포 산출물 크기(linux/arm64, MiB). 왼쪽은 실행 파일의 디스크 크기, 오른쪽은 docker image inspect의 Size다. 공식 이미지는 에이전트 코드를 포함하지 않는다.

| 파일            | 크기     | 컨테이너 이미지                 | 크기     |
|---------------|--------|--------------------------|--------|
| node-runtime  | 117.20 | node:24-bookworm-slim    | 332.27 |
| bun-runtime   | 75.74  | node:24-alpine           | 226.47 |
| deno-runtime  | 81.04  | oven/bun:1-slim          | 261.78 |
| bun-compiled  | 77.55  | oven/bun:1-distroless    | 146.44 |
| deno-compiled | 100.13 | denoland/deno:distroless | 171.91 |
| rust-gnu      | 1.38   | arb-rust-scratch         | 2.20   |
| rust-musl     | 1.44   |                          |        |

## 5. 논의

### 5.1 가설 검증

가정 1은 Rust 구성에서만 시험한 전 범위에 걸쳐 성립했다. Rust의 오버헤드 중앙값은 최대 6.26 ms(rust-st,  $S = 500$ )로 기준 7.5 ms보다 작았다. node는  $S = 10$ (12.04 ms)부터, bun은  $S = 100$ (12.03 ms)부터, deno는  $S = 100$ (7.58 ms)부터 기준을 넘어 기각됐다. 가정 2는 채택됐다.  $S = 100$ 에서 Rust의 최대 RSS는 34.7–36.3 MB로 가장 작은 JavaScript 런타임인 bun의 118.6 MB 보다 작았고,  $S = 500$ 에서도 같았다. 가정 3도 시험한 네  $S$  모두에서 채택됐다.

### 5.2 해석

오버헤드 차이는 스레드 수만으로 설명되지 않는다. rust-st는 JavaScript 런타임처럼 단일 스레드에서 모든 세션을 처리했지만  $S = 500$ 의 오버헤드 중앙값이 6.26 ms였다. 한 가지 설명은 터당 CPU 시간의 차이다.  $S = 500$ 에서 한 차례의 턴을 모든 세션이 수행하는 데 드는 CPU 시간은 터당 CPU 시간에 500을 곱해 node 약 1,030 ms, deno 약 505 ms, bun 약 390 ms, rust-st 약 215 ms 다. 서버 시간이 터당 약 150 ms이므로, 이 값이 클수록 이벤트 루프에 일이 쌓이고 그 대기가  $o$ 에

더해진다. `node`의 도구 지연(352.22 ms)이 다른 런타임보다 큰 것도 자식 프로세스 종료 콜백이 같은 큐에서 기다리기 때문일 수 있다. 이 설명은 추론이다. 이벤트 루프 지연과 스케줄러 대기 시간은 직접 측정하지 않았다.

실무 판단은 동시 세션 수에 달려 있다. 호스트당 세션이 10개 안팎이면 JavaScript 런타임의 오버헤드는 톤당 2.43–12.04 ms이고, 실제 LLM 응답이 수 초 걸린다면 체감 차이는 작다. 이 측정의 서버 시간 150 ms는 실제보다 짧으므로 상대 오버헤드를 크게 보이게 하는 쪽으로 치우쳐 있다. 반면 수백 세션을 한 호스트에 모으면 Rust는 같은 작업에 메모리를 2.4–5.0배 ( $S = 500$ ,  $383.9/159.7$  및  $795.2/159.7$ ), CPU 시간을 1.8–4.8배 (0.78/0.43 및 2.06/0.43) 적게 썼다. 런타임 사이의 차이도 크다. Bun은 같은 JavaScript 코드로 Node.js보다 기동이 5배 빠르고 (35.6/7.0)  $S = 500$ 의 RSS가 절반이었다. 이는 JavaScript 런타임을 하나로 묶지 말라는 Merelo-Guervós et al. [5]의 관찰과 맞는다.

배포에서 단일 파일 여부는 결정 요인이 아니었다. Bun과 Deno도 단일 실행 파일을 만들었다. 차이는 크기와 `libc` 의존이다. `musl` 정적 바이너리는 1.44 MiB였고 동적 의존이 없었다. 에어갭 환경에서 전송하고 검증할 산출물의 크기는 Rust가 두 자릿수 배 작다. 보안 측면의 득실 (메모리 안전성 [7, 8], 패키지 공급망 [9, 10])은 이 연구에서 측정하지 않았다. 도구 격리는 어느 런타임이든 OS 계층에서 해야 한다 [11].

### 5.3 한계

- **환경.** 측정은 Apple M3 Pro 위 Docker Desktop의 aarch64 리눅스 가상 머신에서 했다. 온프레미스 x86 베어메탈 서버에서 같은 결과가 나오는지는 확인하지 않았다. vCPU가 호스트의 성능 코어와 효율 코어 중 어디에 배정됐는지도 통제하지 못했다.
- **작업 모델.** 서버 시간은 톤당 약 150 ms로 실제 LLM보다 짧고, TLS 없이 평문 HTTP를 썼다. 도구는 `cat` 하나다. 컴파일러나 브라우저 같은 무거운 도구는 런타임보다 훨씬 많은 메모리를 쓰므로 런타임 간 차이가 상대적으로 작아진다.
- **구현.** 언어마다 구현이 하나다. Node.js의 `worker_threads`나 다중 프로세스 구성, 다른 HTTP 클라이언트·JSON 라이브러리는 시험하지 않았다. 런타임의 GC·힙 설정도 기본값을 썼다 [6].
- **측정.** E2는 5회, E3는 구성마다 1회 약 6–7분 실행했다. 이 길이로는 메모리 누수를 판단할 수 없다. E3 동안 지연 표본 배열이 약 1.1 MB (표본 수  $\times$  8 B로 추정)까지 커져 모든 구성의 RSS 기울기에 더해졌다. CPU 시간의 해상도는 10 ms이고, 에너지와 통계적 유의성은 측정·검정하지 않았다.

## 6. 결론

LLM 대기가 지배하는 에이전트 루프에서도 런타임 선택은 동시 세션 수가 커질수록 측정 가능한 차이를 만들었다. 시험한 1–500세션에서 Rust 구현은 오케스트레이션 오버헤드 중앙값을 6.26 ms 이하로 유지했고, 기동 시간, 메모리, 톤당 CPU 시간, 배포 크기에서 모두 가장 작았다. JavaScript 런타임은 세션 10개 안팎에서는 오버헤드가 작았지만 100세션 이상에서 사전 기준을 넘었고, 그 정도는 Node.js, Bun, Deno 사이에서도 크게 달랐다. 후속 연구로는 x86 베어메탈 온프레미스 서버에서의 반복 측정, 실제 LLM 지연과 TLS를 포함한 작업, 수 시간 이상의 상주 측정, Node.js 다중 프로세스 구성과의 비교가 필요하다.

**재현 자료.** 측정 코드, Dockerfile, 원자료, 요약 스크립트는 저자가 `agent-runtime-bench` 디렉터리에 보관한다.

## 참고문헌

- [1] OpenAI. Codex CLI is going native. <https://github.com/openai/codex/discussions/1174>, 2025. GitHub Discussion #1174, 30 May 2025. Accessed 2026-10-10.
- [2] Anthropic. Hosting the Agent SDK. <https://code.claude.com/docs/en/agent-sdk/hosting>, 2026. Product documentation. Accessed 2026-10-10.
- [3] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021. doi: 10.1016/j.scico.2021.102609.
- [4] Nicolas van Kempen, Hyuk-Je Kwon, Dung Tuan Nguyen, and Emery D. Berger. It’s not easy being green: On the energy efficiency of programming languages. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1553–1565, 2025. doi: 10.1109/ASE63991.2025.00131.
- [5] Juan J. Merelo-Guervós, Mario García-Valdez, and Pedro A. Castillo. An analysis of energy consumption of JavaScript interpreters with evolutionary algorithm workloads. In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT)*, pages 175–184, 2023. doi: 10.5220/0012128100003538.
- [6] Marisa Kirisame, Pranav Shenoy, and Pavel Panchekha. Optimal heap limits for reducing browser memory use. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):986–1006, 2022. doi: 10.1145/3563323.
- [7] Google Security Blog. Rust in Android: Move fast and fix things. <https://blog.google/security/rust-in-android-move-fast-fix-things/>, 2025. Vendor report, not peer reviewed. Accessed 2026-10-10.
- [8] Mohan Cui, Suran Sun, Hui Xu, and Yangfan Zhou. Is unsafe an Achilles’ heel? A comprehensive study of safety requirements in unsafe Rust programming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–13, 2024. doi: 10.1145/3597503.3639136.
- [9] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium*, pages 995–1010, 2019. URL <https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman>.
- [10] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber’s knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science, pages 23–43, 2020. doi: 10.1007/978-3-030-52683-2\_2.
- [11] Anthropic. Making Claude Code more secure and autonomous with sandboxing. <https://www.anthropic.com/engineering/claude-code-sandboxing>, 2025. Engineering blog. Accessed 2026-10-10.