kiwi-rs: Ergonomic and Performance-Oriented Rust Bindings for the Kiwi Korean Morphological Analyzer

Jai-Chang Park

February 17, 2026

Abstract

This report presents `kiwi-rs`, a Rust library that exposes the public Kiwi C API through a high-level and safety-oriented interface for Korean NLP. The implementation combines dynamic symbol loading, ownership-safe handle wrappers, runtime capability checks, and auto-bootstrap for library/model assets, and it includes an agent-skill package for reliable LLM-assisted onboarding. We evaluate `kiwi-rs` against `kiwipiepy` under matched conditions using repeated-input (warm-cache) and varied-input (near no-cache) workloads with bootstrap confidence intervals and sink-parity checks. On a dataset of 192 Korean texts across 8 categories, `kiwi-rs` shows measurable speedups on selected features (e.g., `join`: 3.85×, `tokenize_many_batch`: 23.18×, `split_into_sents_with_tokens`: 67.75×) while other paths remain close to parity or statistically inconclusive under conservative decision rules. We report design choices, reproducibility metadata, and parity boundaries.

1 Introduction

Rust adoption in production NLP systems has increased due to predictable performance, explicit ownership semantics, and strong tooling [5]. However, high-quality language analyzers are often distributed as C/C++ runtimes with Python-first interfaces [1, 2]. This creates a practical gap: teams that build Rust services need bindings that are more than thin FFI wrappers.

`kiwi-rs` addresses this gap for Kiwi, a Korean morphological analyzer, by providing:

- idiomatic Rust APIs for common analysis pipelines;
- explicit and safe ownership around FFI handles;
- runtime compatibility checks for optional C API surfaces;
- reproducible benchmarking utilities for Rust-vs-Python comparison.

This technical report describes the system architecture and presents benchmark evidence under externally reviewable conditions.

2 Scope and Contributions

As of version `0.1.4` (snapshot date: 2026-02-17), the project reports complete loader coverage for the published Kiwi C symbols (101/101) and broad support for high-level workflows in Rust [3, 4]. The main contributions are:

1. **A production-oriented Rust surface over Kiwi C API.** The library exposes core and advanced capabilities, including batch APIs, typo models, pretokenization constraints, UTF-16 variants, and semantic CoNg operations.

2. **A safety-first runtime model.** The implementation uses RAII cleanup (`Drop`), typed error propagation, runtime feature probing, and internal compatibility guards across object graphs.

3. **A practical bootstrap path.** `Kiwi::init()` can resolve or download matching runtime assets into cache, reducing setup friction while keeping explicit configuration paths available.

4. **A reproducible benchmark framework.** The repository includes paired Rust/Python harnesses, repeated and varied input modes, sink parity checks, and bootstrap confidence intervals for defensible speed claims.

5. **An agent-skill artifact for developer productivity.** The repository packages an assistant skill that constrains LLM responses to runnable code, explicit initialization choices, one-step validation commands, and request-specific pitfalls.

3 System Design

3.1 Dynamic Loading and Capability Detection

`kiwi-rs` loads Kiwi symbols dynamically at runtime and resolves optional APIs when available [3]. Instead of hard-linking a single ABI assumption, the library checks capability flags (e.g., UTF-16, stream builder init, multi-line UTF-16 analysis support) before exposing optional paths. This design reduces failure modes across heterogeneous deployment environments.

3.2 Ownership and Handle Safety

Core runtime objects (library, analyzer, builder, typo model, tokenizer, and intermediate result handles) are wrapped in Rust structs and released via deterministic `Drop` implementations. Internally shared runtime state uses `Arc`, while cross-object validity checks prevent accidental mixing of handles originating from different loaded libraries. This avoids a common FFI class of undefined behavior.

3.3 Error Model

Public APIs return a typed `Result<T, KiwiError>` with distinct categories:

- `LibraryLoad`, `SymbolLoad`, `NulByte`;
- `InvalidArgument`, `Bootstrap`, `Api`.

This separation improves observability for deployment issues compared to string-only error channels.

3.4 Initialization Paths

The library supports three initialization styles:

1. `Kiwi::init()` for auto-bootstrap and cache-based setup;
2. `Kiwi::new()` for environment-driven explicit paths;

Table 1: Supported API surface in `kiwi-rs` (representative groups).

Area	Representative APIs
Initialization and setup	`Kiwi::init`, `Kiwi::new`, `Kiwi::from_config`, `Kiwi::init_direct`
Core inference	`analyze*`, `tokenize*`, `split_into_sents*`, `space*`, `glue*`, `join*`
Batch and native paths	`analyze_many_with_options`, `analyze_many_via_native`, `tokenize_many`, `tokenize_many_with_echo`
Builder and lexicon customization	`add_user_word`, `add_pre_analyzed_word`, `load_user_dictionary`, `add_rule`, `add_re_rule`, `extract_words*`
Constraint and typo models	`MorphemeSet`, `Pretokenized`, `KiwiTypo`, default typo presets
Extended APIs	`SwTokenizer`, CoNg similarity/prediction APIs, UTF-16 variants with runtime support checks

3. `Kiwi::from_config(...)` for fully controlled runtime configuration.

In auto-bootstrap mode, the runtime resolves release metadata, downloads matching archives, and extracts them to an OS-appropriate cache root (or `KIWI_RS_CACHE_DIR` override).

3.5 Inference Hot-Path Caching

The implementation includes lightweight in-process caches for join, tokenize, analyze, split, and glue operations using bounded queues. While this significantly improves repeated-input throughput, the benchmark protocol explicitly separates repeated-input and varied-input runs to prevent cache-inflated claims.

4 API Coverage and Parity Boundaries

`kiwi-rs` covers most C API-backed flows and exposes them with Rust-first signatures. Table 1 summarizes the currently supported API surface by subsystem.

At the C API layer, symbol loading coverage is complete (101/101 loader entries at the reported snapshot). For `kiwipiepy`-surface parity tracking, the current matrix reports 12 `Equivalent`, 19 `Partial`, and 9 `Unavailable` rows (total 40 tracked rows). Full parity is therefore intentionally partial: several Python/C++-specific surfaces (e.g., template layer, dataset/training helpers beyond C API, some utility classes) remain out of scope [2, 1, 3].

4.1 Agent Skill for LLM-Assisted Usage

To reduce prompt ambiguity in AI-assisted development, `kiwi-rs` includes a local skill package (`skills/kiwi-rs-assistant/`) with:

- a structured workflow that routes user intent to API families (tokenize/analyze/split/join, builder, typo, UTF-16, batch, and semantics);

- explicit initialization path rules (`Kiwi::init`, `Kiwi::new`, `Kiwi::from_config`, or builder flow);
- a response contract requiring runnable Rust code, a concrete verification command, and request-specific pitfalls;
- troubleshooting and parity guardrails that map common errors to concrete fixes and prevent unsupported parity claims.

This skill packaging is not presented as a model-quality contribution; instead, it is a reproducible developer-facing artifact that can improve consistency of generated integration code.

5 Evaluation Methodology

5.1 Benchmark Design

The benchmark protocol compares `kiwi-rs` and `kiwipiepy` under aligned settings:

- same text workload and dataset source;
- same warmup/iteration schedules;
- alternating engine order to reduce order bias;
- sink parity checks to validate workload equivalence;
- bootstrap confidence intervals (95%) and $P(\text{ratio} > 1)$.

Decision thresholds follow a practical equivalence band of ±5%: robust wins require confidence intervals entirely above 1.05.

5.2 Dataset and Environment

The dataset benchmark uses `benchmarks/datasets/swe_textset_v2.tsv`:

- 192 rows, 192 unique texts, 8 categories;
- SHA-256: `8c81b8e8d0c4272f96c05e6851da10759f02361caa0a2acb881dd72e642f4696`;
- text length (characters): min 14, median 63, max 192.

Reported runs were executed on macOS 15.7.4 (arm64), Rust 1.93.1, Python 3.14.3, and `kiwipiepy` 0.22.2, with 5 repeats and 2000 bootstrap samples.

6 Results

6.1 Varied-Input (Near No-Cache) Results

Table 2 reports representative features from the varied-input profile. Clear gains appear in `join`, `tokenize_many_batch`, and `split_into_sents_with_tokens`. Other features remain near parity or inconclusive under the strict decision rule.

All listed features passed sink parity checks (1.0000× ratio), indicating equivalent measured workloads between engines for the compared runs.

Table 2: Selected throughput ratios on varied-input profile (`kiwi-rs` / `kiwipiepy`).

Feature	Ratio	95% CI	Decision
`tokenize`	1.49x	[0.97, 1.55]	inconclusive
`split_into_sents`	1.06x	[1.02, 1.12]	likely faster
`split_into_sents_with_tokens`	67.75x	[63.09, 69.85]	robust faster
`space`	1.12x	[1.05, 1.21]	likely faster
`join`	3.85x	[3.68, 4.40]	robust faster
`glue`	1.56x	[1.43, 1.68]	robust faster
`analyze_many_native`	0.92x	[0.70, 1.00]	inconclusive
`tokenize_many_batch`	23.18x	[21.05, 23.59]	robust faster
`space_many_batch`	0.98x	[0.89, 1.47]	inconclusive

Table 3: Category-stratified summary (varied input, per-category runs).

Category	Median Ratio	Weakest Feature (Ratio)
`code_mixed`	40.23x	`join` (4.29x)
`colloquial`	53.59x	`join` (3.98x)
`ecommerce`	53.79x	`join` (4.27x)
`finance`	49.18x	`join` (3.62x)
`longform`	56.26x	`join` (4.70x)
`news`	53.04x	`join` (3.66x)
`tech`	43.72x	`join` (3.12x)
`typo_noisy`	70.02x	`join` (3.97x)

6.2 Repeated-Input (Warm-Cache) Results

Repeated-input measurements show substantially larger speedups for cache-sensitive paths (e.g., `tokenize`: 156.03×, `glue`: 542.54×, `split_into_sents`: 9445.91×). These values are best interpreted as warm-cache upper bounds, not as default deployment expectations.

6.3 Category-Stratified Snapshot

In addition to overall varied-input runs, we used category-stratified evaluation on 8 dataset categories (`code_mixed`, `colloquial`, `ecommerce`, `finance`, `longform`, `news`, `tech`, `typo_noisy`). Table 3 summarizes per-category median relative throughput and the weakest feature in each category.

These category runs should be interpreted as a complementary robustness signal rather than a direct replacement for the overall varied-input baseline. In particular, category-local text pools can still include repeated forms and may amplify cache effects.

6.4 Repeated vs Varied Snapshot (All Common Features)

To provide full cross-mode visibility, Table 4 reports all 15 common benchmark features shared by both engines. This table includes repeated-input and varied-input ratios, plus Δ% relative to parity (1.0×).

Table 4: Full repeated-vs-varied ratio snapshot (`kiwi-rs` / `kiwipiepy`) for all common features.

Feature	Repeated Ratio	Repeated Δ%	Varied Ratio	Varied Δ%
`analyze_many_loop`	150.10x	+14909.5%	0.96x	-3.9%
`analyze_many_native`	24.10x	+2309.9%	0.92x	-7.9%
`analyze_top1`	148.44x	+14744.4%	1.00x	+0.3%
`batch_analyze_native`	24.10x	+2309.9%	0.92x	-7.9%
`glue`	542.54x	+54153.8%	1.56x	+56.2%
`join`	4.30x	+329.9%	3.85x	+285.1%
`space`	99.02x	+9802.0%	1.12x	+11.8%
`space_many_batch`	14.23x	+1322.6%	0.98x	-1.7%
`space_many_loop`	83.17x	+8217.0%	1.05x	+5.0%
`split_into_sents`	9445.91x	+944491.2%	1.06x	+6.4%
`split_into_sents_with_tokens`	86.64x	+8564.1%	67.75x	+6675.4%
`split_many_loop`	1.06x	+6.1%	0.92x	-8.3%
`tokenize`	156.03x	+15503.4%	1.49x	+48.9%
`tokenize_many_batch`	24.62x	+2362.2%	23.18x	+2217.9%
`tokenize_many_loop`	160.42x	+15942.0%	153.21x	+15221.1%

Table 5: Engine-specific benchmark features (median calls/sec).

Feature	Repeated	Varied
`join_prepared` (Rust-only)	142877.01	145837.51
`join_prepared_utf16` (Rust-only)	149083.73	149551.35
`joiner_reuse` (Rust-only)	1841428.46	1836727.20
`joiner_reuse_utf16` (Rust-only)	2289342.35	2047194.66
`split_many_batch` (Python-only)	127.54	100.10

6.5 Engine-Specific Features

Table 5 reports features not shared between both engines in the current harness (4 Rust-only features and 1 Python-only feature).

6.6 Startup Cost

Initialization latency is currently higher in Rust auto-bootstrap mode:

- `kiwi-rs` median init: 1326.721 ms;
- `kiwipiepy` median init: 622.918 ms.

Therefore, one-shot command-line workloads may see weaker end-to-end gains than steady-state service workloads.

7 Discussion

The results suggest two key points:

1. **Steady-state behavior is feature-dependent.** Several operations show strong relative gains, while some batch paths remain near parity.

2. **Evaluation mode materially affects interpretation.** Warm-cache runs can overstate general speedups; varied-input runs offer a stricter baseline.

For production systems, this implies a simple policy: use varied-input statistics for headline claims, and include repeated-input results as supplemental capacity bounds.

8 Threats to Validity

- **Single host profile.** Results were collected on one machine and one OS; cross-hardware variance is not yet quantified.
- **Version snapshot effects.** Results depend on specific versions of Kiwi, `kiwi-rs`, `kiwipiepy`, Rust, and Python.
- **Cache interactions.** Even varied-input configurations may retain partial repetition, especially in category-constrained pools.
- **Process-level overhead differences.** Language runtime overheads and bridge paths differ between Rust and Python and can affect per-feature behavior.

9 Limitations and Future Work

- **Single-machine benchmark scope.** Current results are from one hardware/OS stack.
- **Startup gap.** `Kiwi::init()` convenience introduces startup overhead that should be reduced or amortized.
- **Parity gaps.** Python/C++-specific layers remain out of scope under a strict C API binding strategy.
- **Thread-safety assumptions in upstream runtime.** Some test paths are intentionally serialized around initialization due to observed instability in concurrent setup/teardown.

Planned work includes broader hardware validation, additional memory profiling, startup optimization, and clearly separated optional modules for non-C-API parity features.

10 Reproducibility Checklist

For external review, the following should be published with any claim:

1. exact benchmark commands and flags;
2. dataset path and SHA-256 hash;
3. Rust/Python/package versions;
4. Git SHA and dirty/clean status;
5. varied-input and repeated-input outputs;
6. ratio CIs, $P(\text{ratio} > 1)$, and sink parity table.

The repository already includes scripts and generated artifacts for this checklist in `tmp/feature_dataset_matrix_v2_*`.

11 Conclusion

`kiwi-rs` shows that a Rust-first binding over the Kiwi C API can provide practical ergonomics and explicit safety boundaries. The benchmark evidence supports substantial gains for selected features while also highlighting near-parity regions and startup trade-offs. Overall, the library is a practical option for Rust-native Korean NLP services when paired with transparent, workload-aware evaluation.

Artifact and License Notes

`kiwi-rs` is released under LGPL-2.1-or-later. This manuscript reports software benchmark data; it does not contain user-identifying or sensitive human-subject data.

References

[1] BAB2MIN. Kiwi: Korean intelligent word identifier. https://github.com/bab2min/Kiwi, 2026. Public repository including C API source, accessed: 2026-02-17.

[2] BAB2MIN. kiwipiepy: Python interface for kiwi. https://pypi.org/project/kiwipiepy/, 2026. Version 0.22.2 baseline in this report, accessed: 2026-02-17.

[3] Jai Chang Park. kiwi-rs: Ergonomic rust bindings for the kiwi korean morphological analyzer c api. https://github.com/JAICHANGPARK/kiwi-rs, 2026. Accessed: 2026-02-17.

[4] Jai Chang Park. kiwi-rs on crates.io. https://crates.io/crates/kiwi-rs, 2026. Version 0.1.4, accessed: 2026-02-17.

[5] Rust Project Developers. The rust programming language. https://www.rust-lang.org/, 2026. Accessed: 2026-02-17.

A Benchmark Command Template

```
cd kiwi-rs
mkdir -p tmp
.venv-bench/bin/python scripts/compare_feature_bench.py \
  --dataset-tsv benchmarks/datasets/swe_textset_v2.tsv \
  --input-mode varied \
  --warmup 20 --iters 300 \
  --batch-size 128 --batch-iters 60 \
  --repeats 5 \
  --engine-order alternate \
  --sleep-between-engines-ms 100 \
  --sleep-between-runs-ms 200 \
  --sink-warning-threshold 0.05 \
  --bootstrap-samples 2000 \
  --equivalence-band 0.05 \
  --strict-sink-check \
  --md-out tmp/feature_dataset_matrix_v2_varied_r5_i300/overall.md \
  --json-out tmp/feature_dataset_matrix_v2_varied_r5_i300/overall.json
```