

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 (cache-active, mixed-reuse) 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 varied-input features (e.g., `join`: 3.85 \times , `tokenize_many_batch`: 23.18 \times , `split_into_sents_with_tokens`: 67.75 \times) while other paths remain close to parity or statistically inconclusive under conservative decision rules. Repeated-input numbers are reported as warm-cache capacity bounds only. We additionally report cache-minimized stress results and path-pinned reproducibility runs to bound interpretation of headline ratios.

1 Introduction

Rust adoption in production NLP systems has increased due to predictable performance, explicit ownership semantics, and strong tooling [1]. However, high-quality language analyzers are often distributed as C/C++ runtimes with Python-first interfaces [2, 3]. This creates a practical gap: teams that build Rust services need bindings that are more than thin FFI wrappers.

1.1 Rust Language Context

Rust is a systems programming language designed for high performance and memory safety without a garbage collector [1]. For NLP service engineering, this combination has practical consequences:

- **Safety at FFI boundaries.** Ownership/borrowing and typed error channels reduce common C interop failure classes (dangling pointers, double free, unchecked null paths).
- **Predictable runtime cost.** Ahead-of-time compilation and zero-cost abstractions fit latency-sensitive inference pipelines.
- **Deployment alignment.** Static binary workflows and mature package tooling (`cargo`) simplify service shipping and CI integration.

In short, Rust is a strong fit when the core analyzer is implemented natively (C/C++) but production orchestration requires safer and more ergonomic application-level APIs.

Existing Rust interop stacks (e.g., generated C bindings, C++ bridge layers, Python bridge toolchains) provide useful foundations [4, 5, 6]. `kiwi-rs` is positioned as an engineering-focused integration package on top of those fundamentals: it emphasizes runtime capability detection, deployment bootstrap behavior, and benchmark artifacts rather than proposing a new FFI theory.

`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 document is intentionally an **engineering technical report**: it prioritizes implementation transparency, reproducibility artifacts, and operational trade-offs over algorithmic novelty claims.

2 Related Work

To position `kiwi-rs` in the broader Korean NLP landscape, we organize representative prior work into eight tracks (20 papers total):

- **Korean evaluation benchmarks.** KLUE defines a multi-task Korean language understanding benchmark suite, while KorQuAD provides a Korean machine-reading comprehension benchmark [7, 8].
- **Korean pre-trained language models.** KR-BERT and KoreALBERT report Korean-specific pretraining strategies and efficiency trade-offs for encoder models [9, 10].
- **Transformer pretraining baselines.** BERT, RoBERTa, ALBERT, ELECTRA, and T5 form widely used encoder/text-to-text pretraining baselines that Korean adaptations commonly compare against [11, 12, 13, 14, 15].
- **Multilingual transfer models.** XLM-R and mT5 provide strong multilingual baselines that include Korean and are frequently used as cross-lingual reference points [16, 17].
- **Tokenization infrastructure.** SentencePiece and BPE-style subword segmentation remain central tokenization choices in practical Korean NLP pipelines [18, 19].
- **Transformer architecture foundation.** The original Transformer formulation remains the architectural basis for most of the model families above [20].
- **Sequence labeling foundations.** CRF-based structured prediction and early unified neural tagging frameworks provide important methodological context for POS/morphological sequence decisions [21, 22, 23].
- **Subword and OOV robustness.** Neural LM and subword representation lines (including WordPiece-style segmentation in Japanese/Korean voice search and subword-enriched embeddings) motivate robust handling of sparse or unseen forms [24, 25, 26].

These studies primarily evaluate model quality and benchmark accuracy. `kiwi-rs` is complementary: it contributes a systems-level engineering layer for safe, reproducible, and high-throughput Rust deployment over the existing Kiwi analyzer.

3 Background: Kiwi

Kiwi is a Korean morphological analyzer implemented in C++ and distributed with a public C API, which makes it usable across multiple language runtimes [2]. In modeling terms, Kiwi is not presented upstream as a Transformer architecture; it is described as statistical LM + skip-bigram based disambiguation [2, 27]. In practice, Kiwi is commonly consumed through `kiwipiepy` in Python workflows, while additional ecosystem bindings target other environments [3, 2].

3.1 Modeling Basis Clarification (Not Transformer Architecture)

Because the Kiwi codebase contains identifiers such as `TypoTransformer` and `history_transformer`, it is easy to misread the engine as Transformer-architecture NLP. Upstream documentation and the project citation, however, describe Kiwi’s ambiguity resolution as a combination of a statistical language model and skip-bigram modeling [2, 27]. In this report, we therefore treat Kiwi as a statistical morphology analyzer with LM-based disambiguation, while Transformer-family papers are used only as adjacent related-work context.

Reviewer-oriented disambiguation (TypoTransformer question). For the specific review concern “`TypoTransformer` exists, so is Kiwi Transformer-based?”, our source audit indicates **no**:

- `TypoTransformer` is an error/typo generation-correction utility class (rule/cost based), not a neural encoder block.
- `history_transformer` in KNLM code is a history-index remapping argument for n-gram construction, not self-attention computation.
- Core LM implementation units are explicitly named `Knlm`, `CoNgramModel`, and `SkipBigramModel`, consistent with statistical LM design.
- Build configuration in the upstream project does not introduce deep-learning runtime dependencies (e.g., PyTorch/TensorFlow/ONNX).

Accordingly, we avoid claiming that Kiwi itself is a Transformer neural architecture and keep the terminology separation explicit throughout this report.

3.2 Ecosystem and Distribution Survey

Upstream Kiwi documentation describes a broad multi-language ecosystem centered on the C API and release-packaged binaries [2, 28, 29]. Table 1 summarizes the currently documented channels and wrappers relevant to market and adoption analysis.

This ecosystem breadth suggests that Kiwi demand spans research scripting, server backends, mobile integration, browser runtimes, and non-programmer desktop tooling. From a market-positioning perspective, `kiwi-rs` fills the Rust-native integration slot in this existing multi-language stack rather than competing with upstream analyzers or language-specific wrappers.

3.3 POS Tag Inventory

Kiwi uses a Sejong-based POS scheme with selected extensions and refinements (e.g., web entities, symbol structure, user-defined categories) [2]. This is practically important for `kiwi-rs`, because many downstream integrations consume `Token.tag` and `tag_to_string` outputs as stable labels.

Table 1: Documented Kiwi ecosystem channels and wrappers (upstream snapshot).

Channel	Notes
C API	Public header (<code>include/kiwi/capi.h</code>) as primary native integration boundary.
Compiled binaries	Release artifacts for Windows, Linux, macOS, and Android (library + model files).
C# wrappers	Official GUI wrapper (<code>kiwi-gui</code>) and community wrapper (<code>NetKiwi</code>).
Python wrapper	<code>kiwipiepy</code> as the de facto Python API in production/data workflows.
Java wrapper	<code>KiwiJava</code> (Java 1.8+) via <code>bindings/java</code> .
Android library	NDK-based AAR distribution (<code>kiwi-android-VERSION.aar</code>); documented minimum Android API level 21+ and ARM64 target.
R wrapper	Community wrapper <code>elbird</code> .
Go wrapper	Community wrapper <code>kiwigoo</code> .
WebAssembly	Community JavaScript/TypeScript binding via <code>bindings/wasm</code> .
End-user GUI application	Windows-oriented GUI distribution for non-programmer use cases (<code>kiwi-gui</code>).

A full tag-by-tag inventory is provided in Appendix A, including explicit marking of Kiwi-specific extensions relative to baseline Sejong tags.

For this work, three Kiwi properties are operationally important:

- **C API boundary.** A stable C-layer contract enables dynamic loading and runtime capability probing from Rust.
- **Release-distributed assets.** Platform libraries and model artifacts are versioned together, allowing explicit compatibility control.
- **Feature breadth.** Kiwi exposes not only core tokenization/analysis but also typo handling, sentence processing, and semantic utilities, enabling a broad Rust wrapper surface.

Accordingly, `kiwi-rs` is designed as a C API-centered integration layer rather than a reimplementation of Kiwi internals. This design choice preserves upstream behavior while focusing engineering effort on safety, ergonomics, and deployment reproducibility.

4 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 [30, 31]. The crate targets Rust edition 2021 with MSRV 1.70, which supports older production toolchains while keeping modern language ergonomics [30]. 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 (Kiwi context-graph similarity/prediction APIs).

Table 2: POS tag groups used by Kiwi (summary).

Group	Representative Tags
Nouns and nominals	NNG, NNP, NNB, NR, NP
Predicates	VV, VA, VX, VCP, VCN
Modifiers and particles	MM, MAG, MAJ, IC, JKS–JC
Endings and derivation	EP, EF, EC, ETN, ETM, XPN, XSN, XSV, XSA, XSM, XR
Symbols and scripts	SF, SP, SS, SSO, SSC, SE, SO, SW, SL, SH, SN, SB
Web and special tags	UN, W_URL, W_EMAIL, W_HASHTAG, W_MENTION, W_SERIAL, W_EMOJI, Z_CODA, Z_SIOT, USER0–USER4

Table 3: Kiwi-specific POS extensions relative to baseline Sejong tags.

Extension family	Tags and intent
Derivation and symbol refinement	XSM, SSO, SSC, SB for finer derivational and symbol-structure distinctions.
Unanalyzable and web entities	UN, W_URL, W_EMAIL, W_HASHTAG, W_MENTION, W_SERIAL, W_EMOJI.
Orthographic and user channels	Z_CODA, Z_SIOT, USER0–USER4.

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.

5 System Design

5.1 Dynamic Loading and Capability Detection

`kiwi-rs` loads Kiwi symbols dynamically at runtime and resolves optional APIs when available [30]. 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.

5.2 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. Inter-

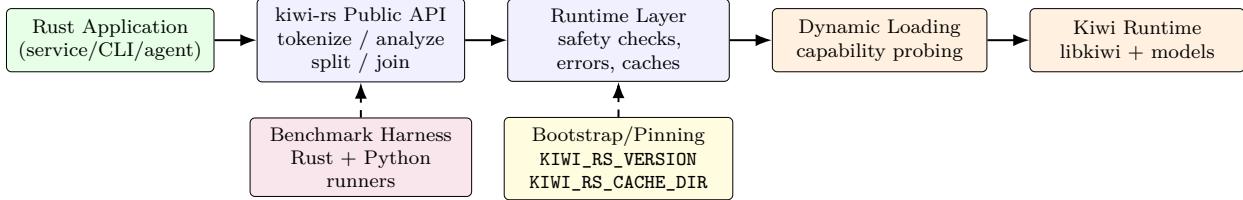


Figure 1: System architecture of `kiwi-rs`. Solid arrows show inference flow; dashed arrows show control flow.

nally 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.

5.3 Error Model

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

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

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

5.4 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;
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).

5.5 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.

Figure 1 summarizes the practical layering used by the library in production deployments.

6 Performance Mechanisms

This section explains *why* the reported throughput numbers can become large on specific features.

Table 4: Primary in-process caches in `kiwi-rs`.

Cache	Capacity	Key Sketch
<code>join_cache</code>	16	morph sequence + <code>lm_search</code>
<code>tokenize_cache</code>	256	tokenize options + text fingerprint + text equality
<code>analyze_cache</code>	128	analyze options + text fingerprint + text equality
<code>split_cache</code>	64	<code>match_options</code> + text fingerprint + text equality
<code>glue_cache</code>	64	chunk list + newline flags + glue fingerprint
<code>glue_pair_cache</code>	256	adjacent pair (<code>left</code> , <code>right</code>)

6.1 Recency Caches and Key Design

`kiwi-rs` uses bounded recency caches on multiple paths. Cache hits are promoted and old entries are evicted from bounded queues, yielding an LRU-like policy with small fixed capacities.

Tokenize/analyze/split entries include a compact text fingerprint (length + edge bytes) and then verify full text equality for collision safety. This reduces average key-comparison cost while preserving correctness.

6.2 Hot-Path Specialization

Several implementations reduce overhead beyond basic caching:

- **Top-1 fast paths.** Cache-backed analyze/tokenize paths are specialized for `top_n=1`, the dominant inference mode in production routing.
- **Joiner reuse APIs.** `prepare_join_morphs` and `prepare_joiner` avoid repeated `CString` construction and joiner re-initialization, which explains large gains in join-heavy microbenchmarks.
- **Glue pair scoring via native batch callback.** `glue_with_options` scores only unresolved adjacent pairs using a native multi-text scoring path, then memoizes pair decisions.
- **Sentence structure construction from token metadata.** This path builds sentence structures directly from token metadata after tokenization, avoiding an extra native split pass.

6.3 Cache Coherence and Safety Trade-offs

To prevent stale inference outputs after runtime mutation, option/config updates clear inference caches. This favors correctness but can reduce throughput in workloads that frequently change analyzer settings.

6.4 Implications for Repeated vs Varied Benchmarks

The large repeated-input speedups are expected from recency caches and prepared-join reuse. For varied-input runs, cache effects can still appear when the active text pool fits cache capacity (e.g., a 192-text pool versus 256-entry tokenize cache). Therefore, very large ratios on some features should be interpreted as *pipeline-aware upper bounds under this harness design*, not universal constants.

Table 5: Supported API surface in `kiwi-rs` (representative groups).

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

7 API Coverage and Parity Boundaries

`kiwi-rs` covers most C API-backed flows and exposes them with Rust-first signatures. Table 5 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 `kiwipy`-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 [3, 2, 30].

7.1 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.

8 Evaluation Methodology

8.1 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 $\pm 5\%$ around parity. For ratio CI $[L, U]$:

- **kiwi-rs faster (robust)** if $L > 1.05$;
- **kiwipiepy faster (robust)** if $U < 0.95$;
- **practically equivalent** if $L \geq 0.95$ and $U \leq 1.05$;
- **kiwi-rs likely faster** if $L > 1.00$ and not robust;
- **kiwipiepy likely faster** if $U < 1.00$ and not robust;
- **inconclusive** otherwise.

8.2 Functional Equivalence Scope

Because this is an engineering report about a binding layer, we separate two notions of equivalence:

- **Workload equivalence** for benchmark fairness, validated by sink parity checks in the harness.
- **API-surface equivalence** against `kiwipiepy`, tracked in a maintained parity matrix (12 Equivalent, 19 Partial, 9 Unavailable; total 40 rows) [32].

In addition, integration tests exercise representative end-to-end behaviors (`tokenize`, `analyze`, sentence splitting, spacing, joining, and user-word injection) to catch regressions in practical outputs [30]. We do not claim corpus-level linguistic accuracy gains over upstream Kiwi in this paper; the focus is wrapper behavior and performance.

8.3 Cross-Engine Agreement Protocol

To complement throughput measurements with NLP-relevant structural checks, we added a cross-engine agreement protocol on the same 192-text dataset. This protocol compares `kiwi-rs` against `kiwipiepy` for:

- exact token-sequence agreement (form, POS tag, start, end);
- token-boundary set agreement (precision/recall/F1 on spans);
- token set agreement (span + form + POS tag precision/recall/F1);

- POS agreement rate on shared spans;
- sentence-boundary exact agreement.

The implementation uses a two-step reproducible path:

1. dump `kiwi-rs` structural outputs via `examples/dump_structural_outputs.rs`;
2. compare those outputs against `kiwipiepy` via `scripts/compare_structural_parity.py`.

Sentence boundaries are normalized to character offsets for cross-engine comparability because some Kiwi C-level paths expose byte-indexed boundaries.

8.4 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 a local development host, Rust 1.93.1, Python 3.14.3, and `kiwipiepy` 0.22.2, with 5 repeats and 2000 bootstrap samples.

8.5 Runtime Pinning and Artifact Lock

Captured benchmark artifacts include explicit environment snapshots. We use two reporting tiers:

- **Tier A (cache-active baseline, main tables).** Dataset-based varied/repeated runs with auto-discovery (`KIWI_LIBRARY_PATH`, `KIWI_MODEL_PATH` unset).
- **Tier B (path-pinned reproducibility run).** Explicit pinning with:
 - `KIWI_LIBRARY_PATH=<KIWI_LIB_PATH>`
 - `KIWI_MODEL_PATH=<KIWI_MODEL_DIR>`
 - `-python-model-path <KIWI_MODEL_DIR>`

For Tier A primary runs, the harness recorded:

- `Git SHA: <REPO_COMMIT_SHA> (dirty=<true|false>);`
- `KIWI_LIBRARY_PATH: unset;`
- `KIWI_MODEL_PATH: unset.`

Given this configuration, the runtime was resolved by `Kiwi::init()` fallback logic at execution time. Therefore, Tier A artifacts are reproducible at the harness level but not immutable to future upstream release drift unless explicit pinning is used [30].

For Tier B, we additionally recorded hash-anchored assets:

- `libkiwi.0.22.2.dylib: e541f158 62e26a44 50f7b0b1 b2355978 c8e498e9 73f9c917 783f2350 81b54bce;`

Table 6: Coverage summary from `cargo llvm-cov` (workspace tests).

Profile	Region Cover	Line Cover	Function Cover
Full workspace	43.02%	41.12%	47.45%
Core logic (excluding <code>runtime.rs</code> / <code>native.rs</code>)	94.69%	92.84%	94.21%

- `cong.mdl`: bd9ca89e e1b72e75 0c8e2166 a17c80a0 fe3fabd8 28c78b1f 0928486a 6b1833a7;
- `sj.morph`: 5e3dab2d ef6d2cc0 79e21d54 77bd610a 391c6904 5d08caf1 e0bbeabd a8db8d1b;
- `default.dict`: d4293e44 b2588d0c 3aabbce6 07a0f41a d3534abd 31b34139 847b1272 54e01549.

We note one remaining boundary for strict cross-engine identity: `kiwipiepy` uses its packaged native extension, so exact binary identity with the Rust-side dynamically loaded `libkiwi` cannot be guaranteed in this harness even when model paths are pinned.

8.6 Test Coverage Measurement

We additionally measured Rust test coverage with `cargo llvm-cov` to report test adequacy in a reproducible form. Coverage was measured using:

- full workspace profile:

```
cargo llvm-cov -workspace -summary-only
- -skip test_add_rule_safety
```

- core-logic profile (excluding FFI boundary-heavy wrappers):

```
cargo llvm-cov -workspace -summary-only
-ignore-filename-regex 'runtime\.rs|native\.rs'
- -skip test_add_rule_safety
```

The second profile is included because this project contains a large volume of thin FFI wrapper code (`runtime.rs`, `native.rs`) whose branch space is dominated by external-library state and optional symbol availability. Concretely, these two files account for 6603/9077 lines (72.75%) of `src/` lines in the measured snapshot. Table 6 summarizes the resulting coverage.

In this paper, when we refer to “core-logic line coverage > 90%”, we mean the filtered profile above, not the unfiltered full-wrapper profile.

9 Results

9.1 Varied-Input (Cache-Active) Results

Table 7 reports representative features from the dataset-based varied-input profile. This profile should be read as *cache-active mixed-reuse*, not cache-free: the 192-entry text pool can still interact with bounded caches and warm paths. 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.

Table 7: Selected throughput ratios on cache-active varied-input profile (`kiwi-rs` / `kiwipiepy`).

Feature	Ratio	95% CI	$P(\text{ratio} > 1)$	Decision
<code> tokenize</code>	1.49x	[0.97, 1.55]	0.955	inconclusive
<code> split_into_sents</code>	1.06x	[1.02, 1.12]	1.000	likely faster
<code> split_into_sents_with_tokens</code>	67.75x	[63.09, 69.85]	1.000	robust faster
<code> space</code>	1.12x	[1.05, 1.21]	1.000	likely faster
<code> join</code>	3.85x	[3.68, 4.40]	1.000	robust faster
<code> glue</code>	1.56x	[1.43, 1.68]	1.000	robust faster
<code> analyze_many_native</code>	0.92x	[0.70, 1.00]	0.029	inconclusive
<code> tokenize_many_batch</code>	23.18x	[21.05, 23.59]	1.000	robust faster
<code> space_many_batch</code>	0.98x	[0.89, 1.47]	0.340	inconclusive

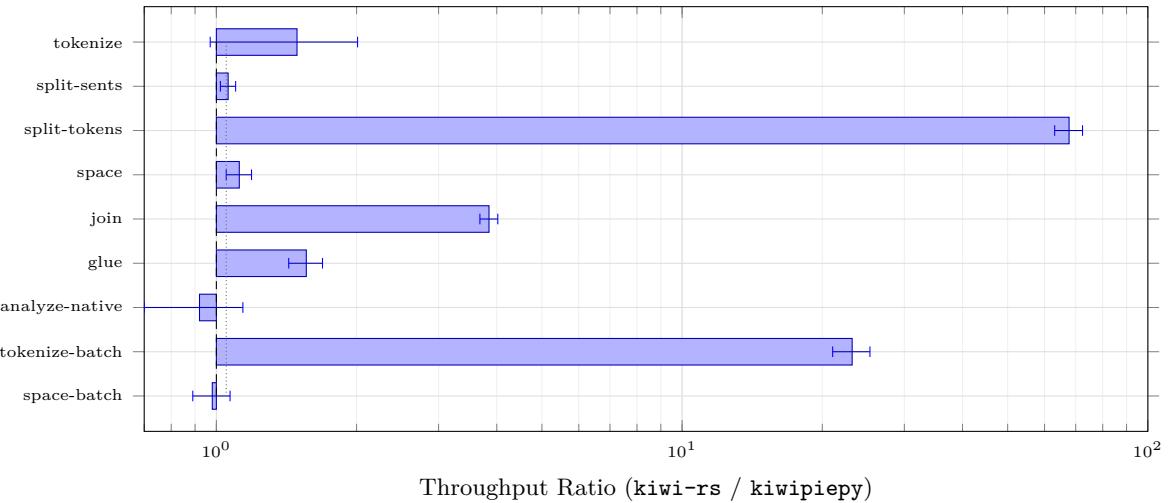


Figure 2: Varied-input throughput ratios with 95% bootstrap confidence intervals. Dashed: parity (1.0×); dotted: practical threshold (1.05×). Feature labels are abbreviated for readability.

All listed features passed sink parity checks ($1.0000 \times$ ratio), indicating equivalent measured workloads between engines for the compared runs. Figure 2 provides a visual summary of the varied-input outcomes and uncertainty ranges.

9.2 Cache-Minimized Stress Profile

To isolate cache effects, we ran an additional synthetic varied-input stress profile with a large variant pool (`variant_pool=65536`, no dataset TSV). Under this setting, repeated reuse is strongly reduced for both single-text and batch paths. Table 8 summarizes representative outcomes.

Compared to Table 7, this stress profile shows that several large dataset-varied gains are strongly cache- and pipeline-dependent. The consistently robust gain in this stress setup is `join`, while most other features become statistically inconclusive.

9.3 Path-Pinned Dataset Run

Under Tier B path pinning (`KIWI_LIBRARY_PATH`, `KIWI_MODEL_PATH`, and `-python-model-path` all set to explicit local paths), key directional outcomes remain similar. Table 9 reports representative

Table 8: Representative ratios under cache-minimized synthetic stress profile (`kiwi-rs` / `kiwipiepy`, 5 repeats).

Feature	Ratio	95% CI	Decision
<code> tokenize</code>	1.00x	[0.91, 1.70]	inconclusive
<code> split_into_sents_with_tokens</code>	0.99x	[0.90, 1.17]	inconclusive
<code> space</code>	1.09x	[0.75, 1.36]	inconclusive
<code> join</code>	5.25x	[4.65, 6.81]	kiwi-rs faster (robust)
<code> analyze_many_native</code>	0.91x	[0.69, 1.01]	inconclusive
<code> tokenize_many_batch</code>	0.88x	[0.77, 2.82]	inconclusive

Table 9: Selected outcomes on path-pinned dataset varied-input run (`kiwi-rs` / `kiwipiepy`, 5 repeats).

Feature	Ratio	95% CI	Decision
<code> tokenize</code>	1.31x	[1.13, 2.30]	kiwi-rs faster (robust)
<code> split_into_sents_with_tokens</code>	68.38x	[60.25, 107.79]	kiwi-rs faster (robust)
<code> join</code>	3.88x	[3.39, 4.70]	kiwi-rs faster (robust)
<code> analyze_many_native</code>	0.93x	[0.81, 1.07]	inconclusive
<code> tokenize_many_batch</code>	26.41x	[25.23, 39.38]	kiwi-rs faster (robust)
<code> space_many_batch</code>	0.99x	[0.88, 1.13]	inconclusive

ratios.

Startup medians also shift under pinning: unpinned varied-input run shows 1326.721 ms (`kiwi-rs`) vs 622.918 ms (`kiwipiepy`), while path-pinned varied-input run shows 1461.594 ms vs 764.915 ms (both $n = 5$). This reinforces that startup comparisons should be interpreted with explicit runtime-path metadata rather than as universal constants.

9.4 Cross-Engine Structural Agreement

Table 10 summarizes structural agreement between `kiwi-rs` and `kiwipiepy` on `swe_textset_v2.tsv`. Agreement is high overall. Token-boundary F1 is 1.0000, token (span+form+tag) F1 is 0.9990, and POS agreement on shared spans is 99.90%.

At row level, sentence and token-boundary mismatches were zero after boundary normalization, while POS mismatches occurred in 6 rows. Table 11 lists the observed POS confusion pairs.

In this comparison stream, the `-R` suffix (e.g., `VV-R`, `VX-R`) appears on the `kiwipiepy` side as a tag-variant marker. We treat these as tag-variant disagreements on identical spans, not as boundary mismatches.

These mismatches are concentrated in `typo_noisy` and `longform` categories. They appear as fine-grained tag variant differences (`VV` vs `VV-R`, `VX` vs `VX-R`) rather than segmentation disagreements. Table 12 and Figure 3 report exact token-sequence agreement by category.

Interpretation should remain conservative: these are cross-engine agreement metrics, not human-annotated linguistic accuracy scores.

Table 10: Cross-engine structural agreement (`kiwi-rs` vs `kiwipiepy`, 192 rows).

Metric	Value
Exact token-sequence match rate	186/192 = 96.88%
Exact sentence-boundary match rate	192/192 = 100.00%
Token-boundary precision / recall / F1	1.0000/1.0000/1.0000
Token (span+form+tag) precision / recall / F1	0.9990/0.9990/0.9990
POS agreement on shared spans	5858/5864 = 99.90%

Table 11: Observed POS confusion pairs on shared spans (cross-engine comparison).

<code>kiwi-rs</code> tag	<code>kiwipiepy</code> tag	Count
VX	VX-R	3
VV	VV-R	3

9.5 Repeated-Input (Warm-Cache) Results

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

9.6 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 13 summarizes per-category median relative throughput and the weakest feature in each category. Here, each category’s “Median Ratio” is the median over the 15 common features, where each feature ratio is computed as:

$$\text{ratio}_f = \frac{\text{median calls/sec}_{\text{kiwi-rs}, f}}{\text{median calls/sec}_{\text{kiwipiepy}, f}}.$$

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.

9.7 Repeated vs Varied Snapshot (All Common Features)

To provide full cross-mode visibility, Table 14 reports all 15 common benchmark features shared by both engines. This table includes repeated-input and varied-input ratios, plus $\Delta\%$ relative to parity (1.0 \times).

Figure 4 highlights that several features are near the diagonal (mode-stable), while others move far upward under repeated-input conditions.

9.8 Engine-Specific Features

Table 15 reports features not shared between both engines in the current harness (4 Rust-only features and 1 Python-only feature).

Table 12: Exact token-sequence agreement by category.

Category	Matches / Total	Rate
news	24/24	100.00%
colloquial	24/24	100.00%
code_mixed	24/24	100.00%
ecommerce	24/24	100.00%
finance	24/24	100.00%
tech	24/24	100.00%
typo_noisy	21/24	87.50%
longform	21/24	87.50%

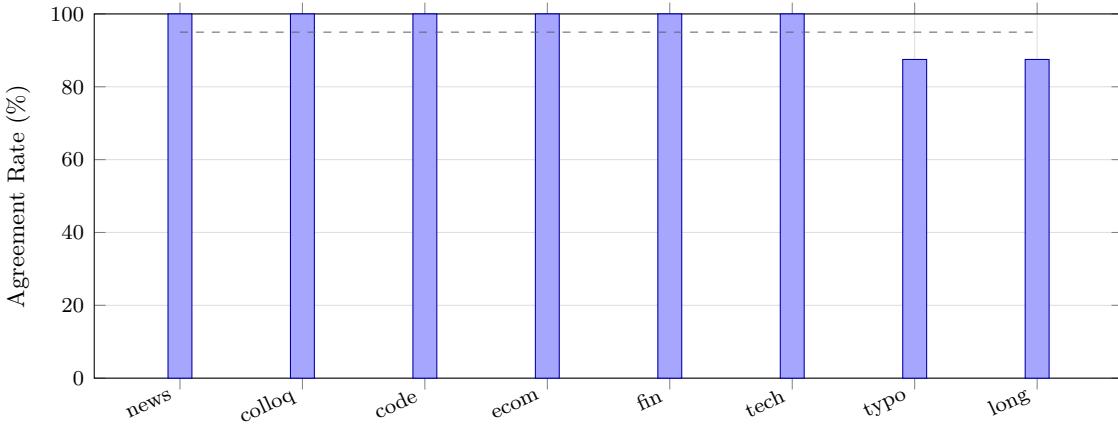


Figure 3: Category-level exact token-sequence agreement rates (`kiwi-rs` vs `kiwipiepy`).

9.9 Startup Cost

Initialization latency is sensitive to pinning profile and initialization mode. Table 16 reports startup medians and IQRs from measured runs.

For Tier A, bootstrap median CIs are [1278.838, 1474.234] ms (`kiwi-rs`) and [617.882, 655.605] ms (`kiwipiepy`). We also probed Rust init paths with a lightweight startup-only workload ($n = 5$): `Kiwi::init()` median 1294.165 ms vs `Kiwi::new()` median 1304.267 ms (no consistent reduction in this setup). Overall, startup is dominated by runtime discovery/loading behavior and should be interpreted with explicit path metadata.

10 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; dataset-varied runs are still cache-active; cache-minimized stress runs provide a stricter bound.

Table 13: 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)

Table 14: 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%

For production systems, this implies a simple policy: report all three modes together (repeated-input, dataset-varied, cache-minimized stress) and use the cache-minimized mode to sanity-check whether large ratios are robust beyond cache/pipeline effects.

11 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.
- **Cross-engine binary identity boundary.** Even with explicit model-path pinning, `kiwipiepy`'s packaged native extension limits strict binary-level identity with Rust-side dynamic loading.
- **Cache interactions.** Even varied-input configurations may retain partial repetition, especially in category-constrained pools.

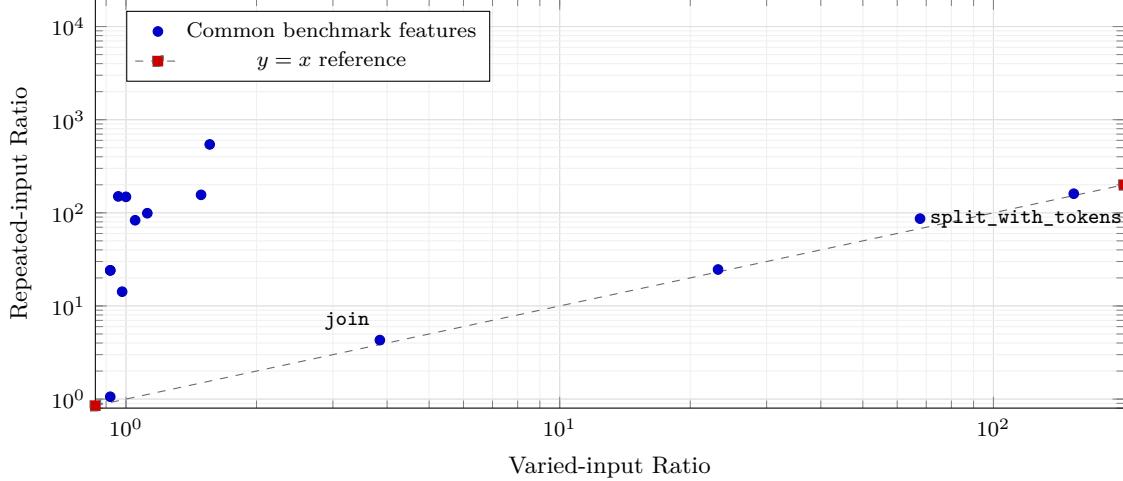


Figure 4: Repeated-input vs varied-input ratio map for all common features. Distance above the $y = x$ line indicates stronger cache/warm-path amplification.

Table 15: Engine-specific benchmark features (median calls/sec).

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

- **Process-level overhead differences.** Language runtime overheads and bridge paths differ between Rust and Python and can affect per-feature behavior.
- **No corpus-level linguistic metric in this report.** We report wrapper parity/behavior and throughput, not task-level morphological accuracy deltas versus upstream Kiwi.

12 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.
- **Tier-dependent runtime immutability.** Tier B includes explicit binary/model hashes, but the Tier A baseline intentionally uses auto-discovery and is therefore less immutable across future upstream drift.

Table 16: Startup latency summary (`init_ms`, lower is better).

Profile	<code>kiwi-rs</code>	<code>kiwipy</code>
Tier A unpinned, dataset-varied ($n = 5$)	1326.721 ms [Q1 1313.006, Q3 1329.406]	622.918 ms [Q1 618.782, Q3 648.147]
Tier B path-pinned, dataset-varied ($n = 5$)	1461.594 ms [Q1 1442.657, Q3 1471.654]	764.915 ms [Q1 686.409, Q3 839.677]

- **Qualitative ecosystem survey.** Wrapper/channel landscape is documented, but adoption telemetry (downloads, active dependents) is not yet normalized in this report.

Planned work includes broader hardware validation, additional memory profiling, startup optimization, immutable runtime lock manifests (release tag + binary/model hashes), and clearly separated optional modules for non-C-API parity features.

13 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. Kiwi runtime lock (release tag or explicit library/model paths) with binary/model SHA-256 hashes;
6. varied-input and repeated-input outputs;
7. ratio CIs, $P(\text{ratio} > 1)$, and sink parity table.

The repository already includes scripts and generated artifacts for this checklist under `tmp/feature_dataset_matrix_v2_varied_r5_i300` and related directories.

14 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] Rust Project Developers. The rust programming language. <https://www.rust-lang.org/>, 2026. Accessed: 2026-02-17.
- [2] BAB2MIN. Kiwi: Korean intelligent word identifier. <https://github.com/bab2min/Kiwi>, 2026. Public repository including C API source, accessed: 2026-02-17.
- [3] BAB2MIN. kiwipiepy: Python interface for kiwi. <https://pypi.org/project/kiwipiepy/>, 2026. Version 0.22.2 baseline in this report, accessed: 2026-02-17.
- [4] Rust bindgen contributors. rust-bindgen. <https://github.com/rust-lang/rust-bindgen>, 2026. Accessed: 2026-02-17.
- [5] David Tolnay Nayuki and contributors. cxx: Safe interop between rust and c++. <https://cxx.rs/>, 2026. Accessed: 2026-02-17.
- [6] PyO3 contributors. Pyo3 user guide. <https://pyo3.rs/>, 2026. Accessed: 2026-02-17.
- [7] Sungjoon Park et al. Klue: Korean language understanding evaluation. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.
- [8] Seungyoung Lim, Myungji Kim, and Jooyoul Lee. KorQuAD 1.0: Korean QA dataset for machine reading comprehension. Dataset and benchmark release, 2019. KorQuAD project documentation.
- [9] Sangah Lee, Hansol Jang, Yunmee Baik, Suzi Park, and Hyopil Shin. KR-BERT: A small-scale korean-specific language model. Technical report, 2020. Korean-specific pretraining study.
- [10] Hyunjae Lee et al. Korealbert: Pretraining a lite BERT model for korean language understanding. Technical report, 2021. Korean ALBERT variant study.
- [11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT 2019*, pages 4171–4186, 2019.
- [12] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. Technical report, 2019. FAIR pretraining report.
- [13] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations (ICLR)*, 2020.
- [14] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Electra: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations (ICLR)*, 2020.
- [15] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

- [16] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. In *Proceedings of ACL 2020*, pages 8440–8451, 2020.
- [17] Linting Xue et al. mt5: A massively multilingual pre-trained text-to-text transformer. In *Proceedings of NAACL-HLT 2021*, pages 483–498, 2021.
- [18] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, 2018.
- [19] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of ACL 2016*, pages 1715–1725, 2016.
- [20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.
- [21] John Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289, 2001.
- [22] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(76):2493–2537, 2011.
- [23] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, 2018.
- [24] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb):1137–1155, 2003.
- [25] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152, 2012.
- [26] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.
- [27] Min-chul Lee. Kiwi: Developing a korean morphological analyzer based on statistical language models and skip-bigram. *Korean Journal of Digital Humanities*, 1(1):109–136, 2024.
- [28] BAB2MIN. Kiwi releases. <https://github.com/bab2min/Kiwi/releases>, 2026. Compiled binaries and model artifacts, accessed: 2026-02-17.
- [29] BAB2MIN. kiwi-gui: Gui and c# wrapper usage for kiwi. <https://github.com/bab2min/kiwi-gui>, 2026. Official GUI tool repository, accessed: 2026-02-17.

- [30] 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.
- [31] Jai Chang Park. kiwi-rs on crates.io. <https://crates.io/crates/kiwi-rs>, 2026. Version 0.1.4, accessed: 2026-02-17.
- [32] Jai Chang Park. kiwipiepy parity matrix. https://github.com/JAICHANGPARK/kiwi-rs/blob/main/docs/kiwipiepy_parity.md, 2026. Parity status matrix used in this report, accessed: 2026-02-17.

A Full POS Tag Inventory

Table 17: Kiwi POS tags used in practice (Sejong-based with extensions).

Major Group	Tag	Description	Scheme
Noun (N)	NNG	common noun	Sejong
Noun (N)	NNP	proper noun	Sejong
Noun (N)	NNB	bound noun	Sejong
Noun (N)	NR	numeral noun	Sejong
Noun (N)	NP	pronoun	Sejong
Predicate (V)	VV	verb	Sejong
Predicate (V)	VA	adjective	Sejong
Predicate (V)	VX	auxiliary predicate	Sejong
Predicate (V)	VCP	positive copula (“to be”)	Sejong
Predicate (V)	VCN	negative copula (“not be”)	Sejong
Determiner	MM	determiner	Sejong
Adverb (MA)	MAG	general adverb	Sejong
Adverb (MA)	MAJ	conjunctive adverb	Sejong
Interjection	IC	interjection	Sejong
Particle (J)	JKS	subjective case particle	Sejong
Particle (J)	JKC	complement case particle	Sejong
Particle (J)	JKG	adnominal/genitive particle	Sejong
Particle (J)	JKO	objective case particle	Sejong
Particle (J)	JKB	adverbial case particle	Sejong
Particle (J)	JKV	vocative particle	Sejong
Particle (J)	JKQ	quotative particle	Sejong
Particle (J)	JX	auxiliary particle	Sejong
Particle (J)	JC	conjunctive particle	Sejong
Ending (E)	EP	pre-final ending	Sejong
Ending (E)	EF	final ending	Sejong
Ending (E)	EC	connective ending	Sejong
Ending (E)	ETN	nominalizing ending	Sejong
Ending (E)	ETM	adnominalizing ending	Sejong
Prefix	XPN	nominal prefix	Sejong
Suffix (XS)	XSN	noun derivational suffix	Sejong
Suffix (XS)	XSV	verb derivational suffix	Sejong
Suffix (XS)	XSA	adjective derivational suffix	Sejong
Suffix (XS)	XSM	adverb derivational suffix	Kiwi extension
Root	XR	root morpheme	Sejong
Symbol/Script (S)	SF	sentence-final punctuation (. ! ?)	Sejong

Major Group	Tag	Description	Scheme
Symbol/Script (S)	SP	separator punctuation (, / : ;)	Sejong
Symbol/Script (S)	SS	quotation/parenthesis symbols	Sejong
Symbol/Script (S)	SSO	opening symbol subset of SS	Kiwi extension
Symbol/Script (S)	SSC	closing symbol subset of SS	Kiwi extension
Symbol/Script (S)	SE	ellipsis	Sejong
Symbol/Script (S)	SO	connector symbol (-, ~)	Sejong
Symbol/Script (S)	SW	other special symbols	Sejong
Symbol/Script (S)	SL	alphabetic string	Sejong
Symbol/Script (S)	SH	Hanja string	Sejong
Symbol/Script (S)	SN	numeric string	Sejong
Symbol/Script (S)	SB	ordered bullet marker	Kiwi extension
Unanalyzable	UN	unanalyzable token	Kiwi extension
Web (W)	W_URL	URL token	Kiwi extension
Web (W)	W_EMAIL	email token	Kiwi extension
Web (W)	W_HASHTAG	hashtag token	Kiwi extension
Web (W)	W_MENTION	mention token	Kiwi extension
Web (W)	W_SERIAL	serial identifier (phone/IP/account)	Kiwi extension
Web (W)	W_EMOJI	emoji token	Kiwi extension
Other (Z)	Z_CODA	appended coda marker	Kiwi extension
Other (Z)	Z_SIOT	interfix <i>saisiot</i> marker	Kiwi extension
User-defined	USER0-USER4	custom user tag slots	Kiwi extension

Note. “Kiwi extension” marks tags that are not in the baseline Sejong tag set.

B Benchmark Command Template

```
cd kiwi-rs
mkdir -p tmp

# Varied-input (headline baseline)
.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

# Repeated-input (warm-cache bound)
.venv-bench/bin/python scripts/compare_feature_bench.py \
```

```

--dataset-tsv benchmarks/datasets/swe_textset_v2.tsv \
--input-mode repeated \
--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_repeated_r5_i300_full/overall.md \
--json-out tmp/feature_dataset_matrix_v2_repeated_r5_i300_full/overall.json

# Cache-minimized varied-input stress profile
.venv-bench/bin/python scripts/compare_feature_bench.py \
--dataset-tsv benchmarks/datasets/swe_textset_v2.tsv \
--input-mode varied \
--variant-pool 65536 \
--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_cache_minimized_varied_r5_i300/overall.md \
--json-out tmp/feature_cache_minimized_varied_r5_i300/overall.json

# Path-pinned Tier B varied-input run (Rust + Python model path)
export KIWI_LIBRARY_PATH=<KIWI_LIB_PATH>
export KIWI_MODEL_PATH=<KIWI_MODEL_DIR>

.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 \
--python-model-path "$KIWI_MODEL_PATH" \
--md-out tmp/feature_pinned_dataset_varied_r5_i300/overall.md \
--json-out tmp/feature_pinned_dataset_varied_r5_i300/overall.json

# Hash lock for pinned assets (Tier B)
shasum -a 256 "$KIWI_LIBRARY_PATH"
shasum -a 256 "$KIWI_MODEL_PATH/cong.mdl"
shasum -a 256 "$KIWI_MODEL_PATH/sj.morph"
shasum -a 256 "$KIWI_MODEL_PATH/default.dict"
```

C Structural Agreement Command Template

```
cd kiwi-rs
mkdir -p tmp/structural_parity

.venv-bench/bin/python scripts/compare_structural_parity.py \
--dataset-tsv benchmarks/datasets/swe_textset_v2.tsv \
--rust-jsonl tmp/structural_parity/kiwi_rs_outputs.jsonl \
--md-out tmp/structural_parity/report.md \
--json-out tmp/structural_parity/report.json
```