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

Jai-Chang Park*

February 18, 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×, `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. 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.

*Google Developer Expert (GDE), Dart & Flutter, Google Developer Groups (GDG) Golang Korea, Flutter Seoul

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 separate prior work into (i) academic model/benchmark context and (ii) production ecosystem context.

Academic model and benchmark context.

- **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].

Table 1: Korean morphological analyzer landscape (engineering comparison, snapshot: 2026-02-18).

System	Core model/runtime	Distribution model	Interfaces	Operational notes
Kiwi [2, 40, 41]	C++ analyzer with statistical LM + skip-bigram disambiguation	Open-source library with prebuilt binaries for major OSes	C API; wrappers for Python, Java, C#, Go, R, and WASM	Broad ecosystem and rich POS extensions; baseline engine targeted by `kiwi-rs`
Khaiii [31]	Data-driven CNN-based Korean morphological analyzer	Open-source project repository	Native runtime and tooling documented in the upstream project	Upstream docs describe dictionary and patch workflows for practical correction loops
KOMORAN	Java-based analyzer with dictionary-centered design [32]	Open-source Java library	Java API; ecosystem wrappers (e.g., PyKOMORAN)	JVM-native deployment; custom dictionary workflow emphasized
Open Korean Text (Okt) [33, 28]	Scala/Java Korean text processor	Open-source library	Scala/Java APIs; KoNLPy `Okt` wrapper	Supports normalization, tokenization, stemming, and phrase extraction for practical indexing workflows
MeCab-ko lineage [30, 34, 28]	MeCab-style dictionary analyzer lineage rooted in CRF-era segmentation/POS design	Community distributions (`mecab-ko` and dictionaries)	Standalone deployment; KoNLPy `Mecab` wrapper	Common production choice in indexing/search stacks when dictionary control is required
KoNLPy wrapper layer [27, 29, 28]	Python integration layer over heterogeneous analyzers	Python package	Unified API over Hannanum, Kkma, Komoran, Mecab, and Okt	Strong prototyping ergonomics; runtime behavior depends on selected backend analyzer
Elasticsearch Nori [35]	Lucene Nori module using `mecab-ko-dic`	Elasticsearch analysis plugin	Analyzer components for Elasticsearch index/query pipelines	Search-oriented morphological analysis integration for production retrieval systems
Bareun API [36]	Hosted Korean NLP service (`bareunpy` gRPC client package)	Managed service with client SDK	Python client and service endpoint integration	Provider-managed API operation with credential-based access
ETRI EPRETX [37, 38, 39]	Public hosted AI API platform (language-processing category)	Managed service portal	Key-based API access from EPRETX portal	AI Open site reports closure on 2025-06-30; EPRETX provides the current portal/API list

Production ecosystem context. For practical deployment choices, the current Korean morphology landscape includes open-source analyzers and hosted services. Representative options include KoNLPy's multi-analyzer wrapper (`Hannanum`/`Kkma`/`Komoran`/`Mecab`/`Okt`), MeCab-ko lineage analyzers, Kakao's Khaiii, KOMORAN, Open Korean Text, Elasticsearch Nori integration, and hosted APIs such as Bareun/ETRI platforms [27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]. Table 1 summarizes these options from an engineering perspective.

These two bodies of work are complementary to `kiwi-rs`: the academic line informs model-level context, while the ecosystem line clarifies real deployment alternatives.

Table 1 summarizes widely used Korean morphological analysis options from an engineering deployment perspective.

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, 40]. In practice, Kiwi is commonly consumed through

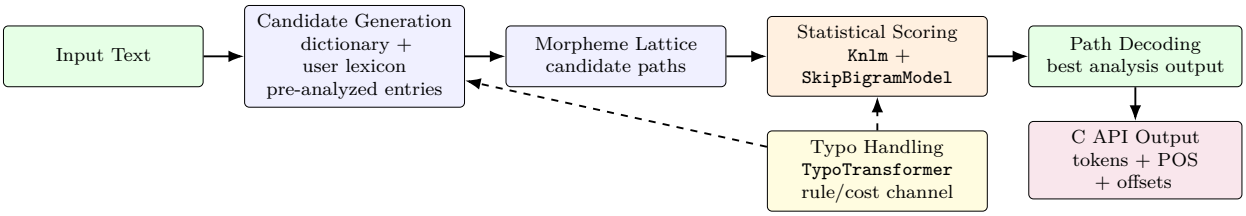

Figure 1: Conceptual Kiwi architecture (upstream-aligned abstraction). Solid arrows: primary inference flow; dashed arrows: typo-control influence.

`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`, it is easy to misread the engine as Transformer NLP. The same risk appears with names like `history_transformer`. Upstream documentation and the project citation, however, describe Kiwi's ambiguity resolution as statistical language-model plus skip-bigram disambiguation [2, 40]. 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.

Architectural Clarification: The `TypoTransformer` Component. 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` and `CoNgramModel`; skip-bigram behavior is represented by `SkipBigramModel`, consistent with statistical LM design.
- Build configuration in the upstream project does not introduce deep-learning runtime dependencies.

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

3.2 Kiwi Architecture (Conceptual View)

Figure 1 summarizes a conceptual inference flow for Kiwi based on upstream descriptions (statistical LM + skip-bigram), public C API boundaries, and source-level module naming [2, 40]. This is an explanatory architecture map, not a line-by-line reconstruction of private internal call graphs.

3.3 Technologies Used by Kiwi

From an implementation and modeling perspective, Kiwi combines several practical technologies [2, 40]:

- **C++ native runtime + C API surface:** core analyzer logic is implemented natively and exported through a stable C boundary (`include/kiwi/capi.h`).
- **Statistical language-model scoring:** ambiguity resolution is framed as sequence scoring over morphological candidates (`Knlm` family).
- **Skip-bigram interaction modeling:** non-local morpheme interaction features are included through dedicated skip-bigram model components.
- **Typo-processing utilities:** typo generation/correction utilities (e.g., `TypoTransformer`) provide error-handling support without implying Transformer self-attention architecture.
- **Release-packaged deployment artifacts:** prebuilt binaries and model assets are distributed for major operating systems, enabling multi-language bindings and production integration.

3.4 Conceptual Scoring Formulation

To make the statistical basis explicit, we use the following *conceptual* decomposition for analysis $y = (m_1, \ldots, m_T)$ of input text x:

$$\mathrm{Score}(y \mid x) = \lambda_{\mathrm{lm}} \sum_{t=1}^{T} \log P_{\mathrm{KNLM}}(m_t \mid h_t) + \lambda_{\mathrm{sb}} \sum_{t=2}^{T} s_{\mathrm{SB}}(m_{\pi(t)}, m_t) - \lambda_{\mathrm{ty}} C_{\mathrm{typo}}(x, y) + \lambda_{\mathrm{lex}} B_{\mathrm{lex}}(y).$$

The selected output is then

$$\hat{y} = \arg\max_{y \in \mathcal{Y}(x)} \mathrm{Score}(y \mid x),$$

where $\mathcal{Y}(x)$ is the candidate set induced by dictionary entries, user lexicon additions, and analyzer constraints.

In this formulation, h_t denotes the KNLM history state at position t, $\pi(t)$ denotes a linked predecessor index (adjacent bigram is the special case $\pi(t) = t - 1$), $s_{\mathrm{SB}}(\cdot, \cdot)$ is a skip-bigram interaction score, $C_{\mathrm{typo}}(x, y)$ is a typo-related penalty term, and $B_{\mathrm{lex}}(y)$ is a lexicon/user-dictionary bonus. Coefficients $\lambda_{\mathrm{lm}}, \lambda_{\mathrm{sb}}, \lambda_{\mathrm{ty}}, \lambda_{\mathrm{lex}}$ are non-negative mixture weights.

Note. This equation is intentionally explanatory: it reflects the reported statistical-LM + skip-bigram design direction and clarifies interaction structure, while exact feature templates, predecessor linking policy, and learned/fixed weights remain implementation-specific in upstream Kiwi.

3.5 Ecosystem and Distribution Survey

Upstream Kiwi documentation describes a broad multi-language ecosystem centered on the C API and release-packaged binaries [2, 41, 42]. Table 2 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.

3.6 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 2: Documented Kiwi ecosystem channels and wrappers (upstream snapshot).

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

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 re-implementation 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: 2026-02-18), the project reports complete loader coverage for the published Kiwi C symbols (101/101) and broad support for high-level workflows in Rust [43, 44]. The crate targets Rust edition 2021 with MSRV 1.70, which supports older production toolchains while keeping modern language ergonomics [43]. 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 3: 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 4: 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 [43]. 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-

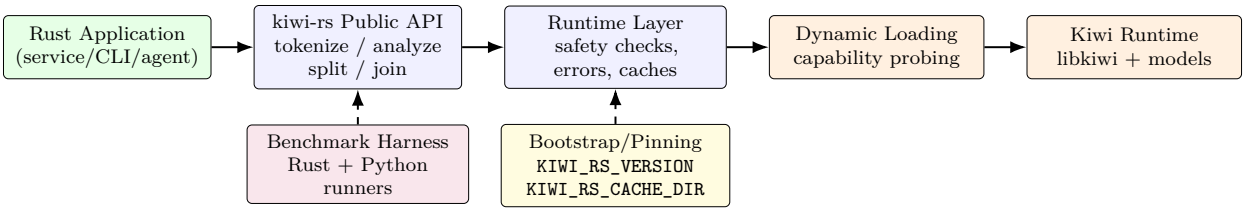

Figure 2: System architecture of `kiwi-rs`. Solid arrows show inference flow; dashed arrows show control flow.

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.

Empirical evidence in this report is design-and-test oriented, not formal verification: the repository includes integration tests and a targeted callback-lifetime safety regression (`tests/safety_check.rs`), but a dedicated dynamic-analysis matrix (e.g., Valgrind/Memcheck or sanitizer-gated FFI boundary runs) is not yet part of the release protocol. We therefore scope the safety claim to type-level ownership constraints, deterministic cleanup, runtime handle-origin checks, and currently exercised tests.

5.3 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.

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 2 summarizes the practical layering used by the library in production deployments.

Table 5: Primary in-process caches in `kiwi-rs`.

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

6 Performance Mechanisms

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

6.1 Terminology: Cache and Warm Paths

We use the following terms consistently in this report:

- **Cold path:** execution on a cache miss, including full request preparation and native runtime computation.
- **Warm path:** execution state where reusable artifacts (cache entries, prepared join structures, pair decisions) reduce repeated work.
- **Cache-active varied input:** varied-input runs with a finite dataset pool; partial cache reuse can still occur.
- **Cache-minimized stress:** varied-input runs with a large variant pool to suppress accidental reuse and approximate lower cache benefit.

6.2 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 and then verify full text equality for collision safety. In implementation terms, for text x, the pre-filter is:

$$F(x) = (|x|,\ H_8(x),\ T_8(x))\,,$$

where H_8 and T_8 pack up to the first/last 8 bytes into `u64`. For `glue_cache`, a composite `u64` is additionally mixed from per-chunk $F(\cdot)$ values and newline flags. These fingerprints are only fast sketches; cache acceptance always requires full key equality (options + full text, or full chunk/newline structures), so collisions cannot alter semantic correctness.

6.3 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.

6.4 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.

6.5 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.

6.6 Analytical Cache Model

To formalize mode sensitivity, let C_f be cache capacity for feature f, and let $|U_f^{(m)}|$ be the number of distinct keys observed under mode $m \in \{r, v\}$ (repeated/varied). Under a uniform-reuse approximation:

$$h_f^{(m)} \approx \min\left(1, \frac{C_f}{|U_f^{(m)}|}\right),$$

where $h_f^{(m)}$ is the cache-hit probability.

With hit latency $\tau_{f,\text{hit}}$, miss latency $\tau_{f,\text{miss}}$, and $\kappa_f = \tau_{f,\text{miss}}/\tau_{f,\text{hit}} \geq 1$, expected Rust-side latency is:

$$\tau_{f,R}^{(m)} = h_f^{(m)}\tau_{f,\text{hit}} + \left(1 - h_f^{(m)}\right)\tau_{f,\text{miss}} = \tau_{f,\text{miss}}\left[\left(1 - h_f^{(m)}\right) + \frac{h_f^{(m)}}{\kappa_f}\right].$$

If the Python-side mode shift is relatively small for the same text pool, the repeated-vs-varied amplification can be approximated by:

$$A_f^{\text{model}} \approx \frac{\left(1 - h_f^{(v)}\right) + h_f^{(v)}/\kappa_f}{\left(1 - h_f^{(r)}\right) + h_f^{(r)}/\kappa_f}.$$

This model explains why A_f^{model} grows when repeated-mode key diversity shrinks ($|U_f^{(r)}| \ll |U_f^{(v)}|$) and hit/miss latency contrast is large ($\kappa_f \gg 1$).

Observation scope: C_f values are directly observable from implementation constants (Table 5), and $|U_f^{(m)}|$ can be measured from benchmark key streams. In this report, however, we do not instrument per-feature cache-hit counters or direct hit/miss latency probes; therefore $h_f^{(m)}$, κ_f, and A_f^{model} are explanatory latent quantities rather than directly estimated parameters.

Table 6: 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

7 API Coverage and Parity Boundaries

`kiwi-rs` covers most C API-backed flows and exposes them with Rust-first signatures. Table 6 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 [3, 2, 43].

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.

Maintenance model. The skill package is currently maintained as version-controlled static artifacts (`SKILL.md`, references, and agent config YAML) in the same repository. Updates are release-coupled: when APIs or parity boundaries change, the skill text is revised in the same review cycle to keep prompt guidance aligned with the shipped crate surface.

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 $P(\text{ratio} > 1)$.

8.2 Statistical Definitions

For a feature f, engine $e \in \{R, P\}$ (Rust/Python), and repeat index $i \in \{1, \ldots, n\}$, let $x_{f,e,i}$ denote the measured calls-per-second. We define the per-engine summary as the median over repeats:

$$\tilde{x}_{f,e} = \text{median}_i(x_{f,e,i}).$$

The reported throughput ratio is:

$$\hat{\rho}_f = \frac{\tilde{x}_{f,R}}{\tilde{x}_{f,P}}.$$

Bootstrap ratios are computed with B resamples ($B = 2000$ in this report). For each bootstrap replicate b:

$$\rho_f^{*(b)} = \frac{\tilde{x}_{f,R}^{*(b)}}{\tilde{x}_{f,P}^{*(b)}},$$

and we denote the bootstrap ratio sample by

$$\rho_f^* = \{\rho_f^{*(1)}, \rho_f^{*(2)}, \ldots, \rho_f^{*(B)}\}.$$

and the 95% confidence interval is the percentile interval:

$$[L_f, U_f] = [Q_{0.025}(\rho_f^*),\, Q_{0.975}(\rho_f^*)].$$

We also report the posterior-like bootstrap probability:

$$\hat{p}_f = \frac{1}{B} \sum_{b=1}^{B} \mathbf{1}\left[\rho_f^{*(b)} > 1\right].$$

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.

8.3 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) [45].

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 [43]. We do not claim corpus-level linguistic accuracy gains over upstream Kiwi in this paper; the focus is wrapper behavior and performance.

8.4 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.

8.5 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.

8.6 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` and dirty/clean status (captured in artifact metadata fields `environment.git_sha` and `environment.git_dirty`);
- `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 [43].

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

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

Table 7: Coverage summary from `cargo llvm-cov` (workspace tests).

Profile	Region Coverage	Line Coverage	Function Coverage
Full workspace	43.02%	41.12%	47.45%
Core logic (excluding `runtime.rs`/`native.rs`)	94.69%	92.84%	94.21%

8.7 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 7 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 8 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.

All listed features passed sink parity checks ($1.0000\times$ ratio), indicating equivalent measured workloads between engines for the compared runs. Figure 3 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 9 summarizes representative outcomes.

Table 8: Selected throughput ratios on cache-active varied-input profile (`kiwi-rs` / `kiwipiepy`).

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

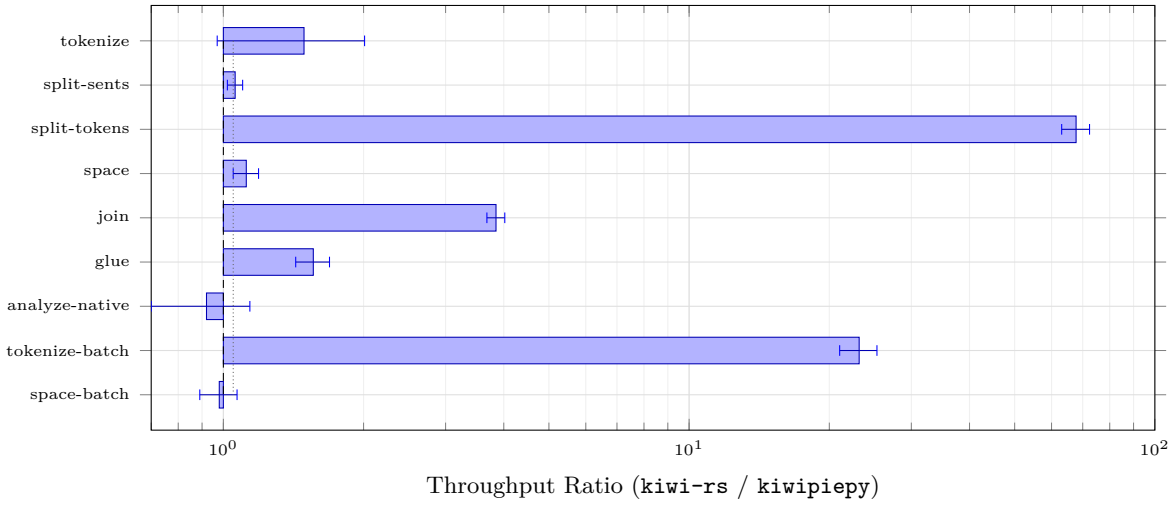

Figure 3: Varied-input throughput ratios with 95% bootstrap confidence intervals. Dashed: parity (1.0×); dotted: practical threshold (1.05×). Feature labels are abbreviated for readability.

Compared to Table 8, 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 10 reports representative ratios.

Startup medians also shift under pinning. The unpinned varied-input run shows 1326.721 ms (`kiwi-rs`) vs 622.918 ms (`kiwipiepy`); the 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.

Table 9: Representative ratios under cache-minimized synthetic stress profile (`kiwi-rs` / `kiwipiepy`, 5 repeats).

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

Table 10: Selected outcomes on path-pinned dataset varied-input run (`kiwi-rs` / `kiwipiepy`, 5 repeats).

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

9.4 Cross-Engine Structural Agreement

Table 11 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 12 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 13 and Figure 4 report exact token-sequence agreement by category.

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

9.5 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.

Table 11: 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 12: Observed POS confusion pairs on shared spans (cross-engine comparison).

`kiwi-rs` tag	`kiwipiepy` tag	Count
`VX`	`VX-R`	3
`VV`	`VV-R`	3

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 14 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:

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

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

For each feature, we define the empirical cache/warm-path amplification factor:

$$A_f^{\text{emp}} = \frac{\hat{\rho}_f^{\text{repeated}}}{\hat{\rho}_f^{\text{varied}}}.$$

Values $A_f^{\text{emp}} \gg 1$ indicate strong repeated-mode amplification relative to the varied profile.

Figure 5 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 16 reports features not shared between both engines in the current harness (4 Rust-only features and 1 Python-only feature).

Table 13: 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%

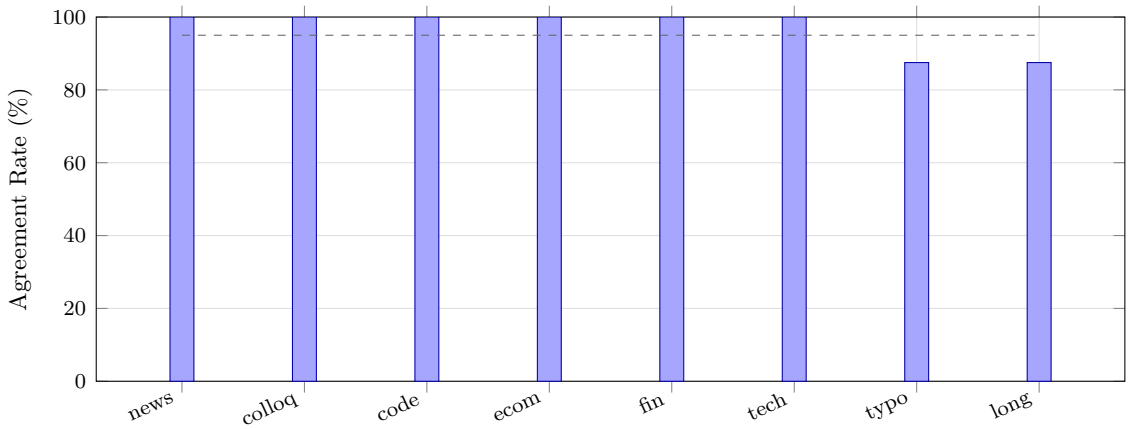

Figure 4: Category-level exact token-sequence agreement rates (`kiwi-rs` vs `kiwipiepy`).

9.9 Startup Cost

Initialization latency is sensitive to pinning profile and initialization mode. Table 17 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.

Startup mitigation options. For production deployments, practical mitigation options are:

- warm process pools that amortize one-time initialization;
- explicit path pinning via `KIWI_LIBRARY_PATH` and `KIWI_MODEL_PATH` to reduce discovery variance;
- explicit constructors (`Kiwi::new` or `Kiwi::from_config`) when auto-bootstrap behavior is unnecessary;
- a future "fast-init" mode that skips discovery/bootstrap by requiring fixed runtime paths at deploy time (targeting short-lived CLI/FaaS processes).

Table 14: 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 15: 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%

More aggressive options (e.g., static packaging or deeper loader specialization) may be constrained by upstream runtime distribution policy, ABI compatibility, and platform-specific linking requirements.

Memory-footprint note. The current benchmark harness is throughput/latency oriented and does not yet report peak RSS or allocator-level memory statistics. We therefore avoid quantitative memory claims in this report and treat memory comparison as an explicit follow-up task.

9.10 Build and Deployment Footprint

To quantify practical adoption cost, we measured three footprint components on the benchmark host: (i) crate package transfer size, (ii) release-binary delta for a minimal consumer application, and (iii) runtime library/model payload size.

For a minimal consumer workflow that calls `Kiwi::init()`, incremental deployment footprint can be approximated as

$$\Delta S_{\text{deploy}} \approx \Delta S_{\text{app\_bin}} + S_{\text{runtime\_payload}},$$

which yields approximately 0.162 MiB + 97.88∼109.38 MiB on this snapshot, i.e., roughly 98.0∼

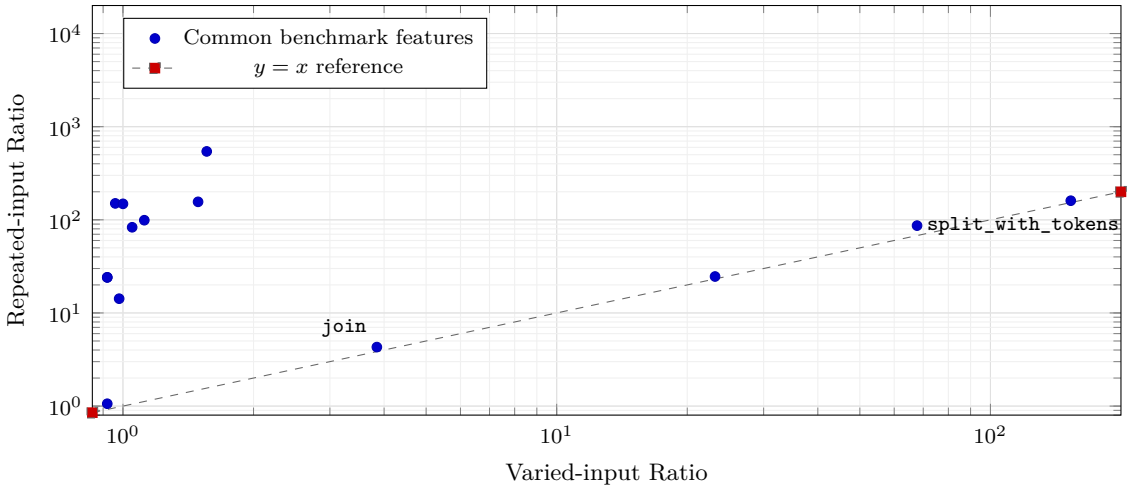

Figure 5: Repeated-input vs varied-input ratio map for all common features. Distance above the $y = x$ line indicates stronger cache/warm-path amplification.

Table 16: 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

109.5 MiB additional footprint for practical runtime use. Bundling static archives or extra model assets increases this bound (full local installation on the measured host: about 194 MiB).

10 Discussion

The results suggest three 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.

3. **Deployment size is runtime-asset dominated.** In our measurement, Rust consumer binary growth is modest (≈ 0.162 MiB), while practical Kiwi runtime assets contribute roughly $98 \sim 109$ MiB.

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.

Table 17: Startup latency summary (`init_ms`, lower is better).

Profile	`kiwi-rs`	`kiwipiepy`
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]

Table 18: Package and binary-size measurements (`kiwi-rs` v0.1.4, macOS arm64 snapshot).

Item	Size (bytes)	Notes
`kiwi-rs-0.1.4.crate` (`cargo package -offline`)	371,465	≈362.8 KiB compressed package archive
Minimal release binary (baseline app)	407,232	`println!` only
Minimal release binary (+ `kiwi-rs`, `Kiwi::init()`)	576,608	dependency linked, init path referenced
Binary delta (consumer app)	**169,376**	+41.6%, **≈0.162 MiB**

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, the packaged native extension in kiwipiepy limits strict binary-level identity with Rust-side dynamic loading paths.
- **Cache interactions.** Varied-input configurations may still retain partial repetition in constrained category pools.
- **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.

Table 19: Runtime payload snapshot (resolved under `$HOME/.local/kiwi`).

Runtime asset	Size (bytes)	Size (MiB)
`lib/libkiwi.0.22.2.dylib`	15,411,424	14.70
`models/cong/base/cong.mdl`	75,667,563	72.16
`models/cong/base/sj.morph`	8,462,892	8.07
`models/cong/base/default.dict`	3,090,954	2.95
`models/cong/base/multi.dict` (optional)	12,064,440	11.50
Core runtime payload (without `multi.dict`)	**102,632,833**	**97.88**
Core runtime payload (+ `multi.dict`)	**114,697,273**	**109.38**

- **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.
- **Qualitative ecosystem survey.** Wrapper/channel landscape is documented, but adoption telemetry (downloads, active dependents) is not yet normalized in this report.
- **FFI boundary verification gap.** A full dynamic-analysis matrix is not yet part of the release gate. Planned evidence includes Valgrind or Memcheck runs, sanitizer-gated CI lanes, and scoped Miri checks where applicable.

Near-term roadmap.

1. **Startup optimization track.** Evaluate constructor-path policies for the `new` and `from_config` initialization APIs, plus prewarmed workers and pinning defaults. Include an optional fast-init profile that disables bootstrap/discovery and report median startup deltas on Tier A/Tier B.
2. **Memory profiling track.** Add peak RSS and allocation telemetry for repeated, dataset-varied, and cache-minimized modes; publish side-by-side Rust/Python summaries.
3. **Cross-host reproducibility track.** Re-run the benchmark matrix on multiple OS/CPU profiles and publish variance envelopes for key features.
4. **Parity and artifact hygiene track.** Continue parity-matrix updates, keep skill documentation release-coupled, and publish immutable runtime lock manifests (release tag + binary/model hashes).
5. **FFI safety verification track.** Add boundary-focused validation lanes (ASan/UBSan/LSan and Valgrind where supported) plus scoped Miri checks for non-FFI ownership invariants; publish crash-free stress logs and failure triage artifacts.

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` demonstrates that a Rust-first integration layer over the Kiwi C API can provide stronger safety and deployment ergonomics than thin bindings while preserving practical performance. The report contributes three concrete artifacts: (i) a capability-aware and ownership-safe runtime wrapper, (ii) a transparent multi-profile benchmark protocol (repeated, cache-active varied, cache-minimized), and (iii) reproducibility-oriented evaluation assets including parity checks, command templates, and appendix algorithms.

Results support strong speedups on selected paths and near-parity or inconclusive outcomes on others, which reinforces a central claim of this work: binding-layer performance must be interpreted by workload mode, not a single headline ratio. Startup overhead and missing memory telemetry remain open operational concerns. With the roadmap outlined in Section 11, `kiwi-rs` is positioned as a robust engineering baseline for Rust-native Korean NLP services and reproducible systems research.

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] KoNLPy contributors. Konlpy documentation. https://konlpy.org/en/latest/, 2026. API docs and analyzer integrations (Hannanum, Kkma, Komoran, MeCab, Okt), accessed: 2026-02-17.

[28] KoNLPy contributors. Konlpy tag package api reference. https://konlpy.org/en/latest/api/konlpy.tag/, 2026. Class-level wrappers for Hannanum, Kkma, Komoran, Mecab, and Okt, accessed: 2026-02-17.

[29] Eunjeong L. Park and Sungzoon Cho. Konlpy: Korean natural language processing in python. In *Proceedings of the 26th Annual Conference on Human and Cognitive Language Technology*, Chuncheon, Korea, October 2014.

[30] Eunjeon Project. Mecab-ko. https://bitbucket.org/eunjeon/mecab-ko, 2013. Korean adaptation/distribution of MeCab, accessed: 2026-02-17.

[31] Kakao and contributors. Khaiii: Kakao hangul analyzer iii. https://github.com/kakao/khaiii, 2026. Official repository, accessed: 2026-02-17.

[32] Shineware and contributors. Komoran: Korean morphological analyzer in java. https://github.com/shineware/KOMORAN, 2026. Official repository, accessed: 2026-02-17.

[33] Open Korean Text contributors. Open korean text processor. https://github.com/open-korean-text/open-korean-text, 2026. Official repository, accessed: 2026-02-17.

[34] Taku Kudo, Kaoru Yamamoto, and Yuji Matsumoto. Applying conditional random fields to japanese morphological analysis. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 230–237, 2004.

[35] Elastic. Elasticsearch nori plugin. https://www.elastic.co/guide/en/elasticsearch/plugins/current/analysis-nori.html, 2026. Korean analyzer based on mecab-ko-dic, accessed: 2026-02-17.

[36] Softcore and Bareun team. bareunpy: Python api for bareun korean nlp service. https://pypi.org/project/bareunpy/, 2026. Hosted API client package, accessed: 2026-02-17.

[37] Electronics and Telecommunications Research Institute (ETRI). Etri ai open api service termination notice. https://aiopen.etri.re.kr/, 2025. Notice indicates migration to a successor platform, accessed: 2026-02-17.

[38] ETRI Language Intelligence Research Section. Epretx open api documentation. https://epretx.etri.re.kr/, 2026. Successor public API platform for language services, accessed: 2026-02-17.

[39] ETRI Language Intelligence Research Section. Epretx api list page. https://epretx.etri.re.kr/apiList, 2026. Public API catalog page indicating key-based usage and language-processing category, accessed: 2026-02-17.

[40] 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.

[41] BAB2MIN. Kiwi releases. https://github.com/bab2min/Kiwi/releases, 2026. Compiled binaries and model artifacts, accessed: 2026-02-17.

[42] 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.

[43] 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.

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

[45] Jai Chang Park. kiwi-rs 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 20: 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

Major Group	Tag	Description	Scheme
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
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

Figure 6: Algorithm 1: Benchmark Compare Loop (Rust vs Python)

Input: dataset D, feature set F, repeats n, warmup w, iterations t
for all feature $f \in F$ **do**
    for run = 1 to n **do**
        for all engine in alternating_order({rust, python}) **do**
            warmup(engine, f, D, w)
            $m \leftarrow$ measure_calls_per_sec(engine, f, D, t)
            $s \leftarrow$ sink_value(engine, f, D)
            store(engine, f, run, m, s)
        end for
    end for
    $\rho_f \leftarrow \text{median}(M_{rust,f})/\text{median}(M_{python,f})$
    run sink-parity guardrail for f
    compute bootstrap CI and $P(\rho_f > 1)$
end for
Output: per-feature metrics, ratios, CI, decisions

Figure 7: Algorithm 2: Sink-Parity Guardrail

Input: sink_{rust}, sink_{python}, threshold ϵ
if $\text{sink}_{python} = 0$ **then**
    $e \leftarrow |sink_{rust} - sink_{python}|$
else
    $e \leftarrow |sink_{rust} - sink_{python}|/|sink_{python}|$
end if
if $e > \epsilon$ **then**
    mark feature as non-equivalent; fail strict mode
end if

Major Group	Tag	Description	Scheme
Other (Z)	`Z_SIOT`	interfix *saisiot* 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 and Test Pseudocode

This appendix summarizes benchmark/test logic in implementation-agnostic pseudocode for reviewability.

C Implementation Mapping and Traceability

For traceability, lock all appendix command outputs to the submission Git SHA reported by:

```
git rev-parse HEAD
```

Figure 8: Algorithm 3: Bootstrap Ratio Confidence Interval

Input: run vectors R, P; bootstrap samples B
for $b = 1$ to B **do**
    $R_b \leftarrow$ resample_with_replacement(R)
    $P_b \leftarrow$ resample_with_replacement(P)
    $\rho_b^* \leftarrow \text{median}(R_b)/\text{median}(P_b)$
end for
$CI_{95} \leftarrow [Q_{0.025}(\rho^*), Q_{0.975}(\rho^*)]$
$p \leftarrow \text{mean}(\mathbf{1}[\rho^* > 1])$
return CI_{95}, p

Figure 9: Algorithm 4: Structural Agreement (Cross-Engine)

Input: per-row Rust outputs U_r, Python outputs U_p
for all row i **do**
    normalize sentence boundaries to character offsets
    compare exact token sequence (form, tag, start, end)
    compare token-boundary sets (precision/recall/F1)
    compare token sets (span+form+tag precision/recall/F1)
    compute POS agreement on shared spans
end for
aggregate corpus-level rates and confusion pairs

D 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 \
```

Table 21: Pseudocode-to-implementation mapping in this repository.

Pseudocode item	Primary implementation path	Role
Algorithm 1	`scripts/compare_feature_bench.py`	Main benchmark orchestration, ratio/decision reporting, artifact export
Algorithm 2	`scripts/compare_feature_bench.py`	Sink-parity calculation, strict/non-strict guardrails
Algorithm 3	`scripts/compare_feature_bench.py`	Bootstrap resampling, percentile CI, $P(\text{ratio} > 1)$
Algorithm 4	`scripts/compare_structural_parity.py`, `examples/dump_structural_outputs.rs`	Cross-engine structural agreement and confusion extraction

```
  --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"
```

E 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
```

F Build Footprint Command Template

```
cd kiwi-rs

# 1) Crate package size
cargo package --allow-dirty --no-verify --offline
```

```
stat -f "%N %z bytes" target/package/kiwi-rs-0.1.4.crate

# 2) Consumer binary delta (baseline vs kiwi-rs)
mkdir -p /tmp/size_base /tmp/size_with_kiwi

cd /tmp/size_base
cargo init --bin --name size_base --vcs none .
cat > src/main.rs <<'EOF'
fn main() { println!("size probe baseline"); }
EOF
cargo build --release --offline
stat -f "%N %z bytes" target/release/size_base

cd /tmp/size_with_kiwi
cargo init --bin --name size_with_kiwi --vcs none .
cat > Cargo.toml <<'EOF'
[package]
name = "size_with_kiwi"
version = "0.1.0"
edition = "2024"

[dependencies]
kiwi-rs = { path = "/path/to/kiwi-rs" }
EOF
cat > src/main.rs <<'EOF'
use kiwi_rs::Kiwi;
fn main() { let _ = Kiwi::init(); }
EOF
cargo build --release --offline
stat -f "%N %z bytes" target/release/size_with_kiwi

# 3) Runtime payload snapshot (example: local install)
stat -f "%N %z" \
  "$HOME/.local/kiwi/lib/libkiwi.0.22.2.dylib" \
  "$HOME/.local/kiwi/models/cong/base/cong.mdl" \
  "$HOME/.local/kiwi/models/cong/base/sj.morph" \
  "$HOME/.local/kiwi/models/cong/base/default.dict" \
  "$HOME/.local/kiwi/models/cong/base/multi.dict"
du -sh "$HOME/.local/kiwi/lib" "$HOME/.local/kiwi/models" "$HOME/.local/kiwi"
```