flutter_kiwi_nlp: A Native-First, Cross-Platform Korean NLP Plugin for Flutter

Jai-Chang Park
Google Developer Expert (GDE), Dart & Flutter

February 18, 2026

Abstract

This paper presents `flutter_kiwi_nlp`, a production-oriented Flutter plugin for Korean morphological analysis built on Kiwi. The package exposes one stable Dart API while internally operating two runtime stacks: Native (Dart FFI + C bridge + Kiwi shared library) and Web (Dart JS interop + kiwi-nlp WASM). This design enables a single application codebase across Android, iOS, macOS, Linux, Windows, and Web.

The implementation is explicitly aligned with on-device AI requirements: local inference execution, reduced text egress by default, predictable latency without per-request network dependence, and operational fallback controls for model provisioning. Empirically, repeated benchmark trials show that Flutter throughput is lower than the Python-native baseline in current measurements: approximately 0.72x on same-host desktop baseline, and as low as 0.63x in mobile engineering-reference rows that compare emulator/simulator Flutter runs against host macOS Python. The updated benchmark pipeline also adds boundary- decomposed measurements (pure processing vs full JSON path), quantifying serialization/parsing overhead at 11–15% of Flutter full-path elapsed time across measured environments. Cross-runtime linguistic agreement remains close on gold corpora (88.39% vs 88.58% token agreement; 84.90% vs 85.55% POS agreement). On native targets, inference can run fully offline after one-time model provisioning.

This paper provides an implementation-complete, reproducible specification of API contract, runtime parity rules, build automation hooks, security boundaries, failure taxonomy, benchmark protocol, and quantified ecosystem survey signals.

1 Introduction

Korean morphological analysis is a foundational primitive for search, retrieval, classification, and generation workflows. In Flutter environments, application logic can be shared across targets, but language runtime integration remains platform-specific. `flutter_kiwi_nlp` addresses this mismatch by encapsulating runtime diversity behind a single API boundary.

The central engineering problem is not only to expose Kiwi functionality, but to make runtime behavior predictable when deployment environments differ in:

- native dynamic library semantics,
- browser module and WASM loading constraints,
- model-file availability,
- platform build toolchains and artifact formats.

1.1 On-Device AI Positioning

This plugin is designed as an on-device AI integration layer, not a network-first inference client. In this context, "on-device" means that tokenization and morphological analysis execute inside the host application process (native FFI path) or in-browser runtime (WASM path), with optional model download only for provisioning. This positioning provides:

- stronger default data locality for analyzed text,
- lower dependency on network availability during inference,
- deterministic runtime behavior under explicit model/version controls,
- easier integration into privacy- or compliance-sensitive workloads.

For native targets, once model artifacts are bundled or cached locally, analysis can execute without network connectivity (offline-first inference path). For web targets, comparable offline behavior requires deployment-specific caching and self-hosting policy because the default bootstrap uses CDN module/WASM URLs.

2 Related Work

This revision expands the related-work coverage to include core Transformer foundations, compact/efficient variants, on-device inference studies, and Korean-language-specific resources. For foundation and compression lineage, prior work includes BERT, ALBERT, ELECTRA, DistilBERT, TinyBERT, MiniLM, and MobileBERT [1, 2, 3, 4, 5, 6, 7]. These studies establish the main trade-off surface between representational quality and model size/latency.

For edge execution and deployment-oriented efficiency, SqueezeBERT, EdgeBERT, and DeeBERT provide architectural or runtime pathways for lower-latency transformer inference, while pNLP-Mixer explores an all-MLP design for compact on-device language modeling [8, 9, 10, 11]. Application focused on-device work reports practical workloads such as VQA, form filling, and smart-reply code-switching, and personalization-oriented studies analyze device-local vocabulary adaptation under memory/latency constraints [12, 13].

For Korean NLP context, KR-BERT, KLUE, KoBigBird-large, and character-level Korean morphological analysis/POS tagging provide complementary evidence on language-specific tokenization, evaluation, and modeling behavior [14, 15, 16, 17]. These works are primarily model or benchmark contributions; by contrast, this paper focuses on systems integration for production Flutter apps, including runtime selection, artifact provisioning, fallback semantics, and reproducibility constraints.

The Korean morphological analyzer ecosystem itself is also important related work context. Widely used dictionary-centric analyzers and lexicons such as MeCab-ko and MeCab-ko-dic represent a practical baseline in many production pipelines, while Khaiii represents a neural approach with different quality and runtime trade-offs [18, 19, 20]. In Python workflows, KoNLPy is often used as an integration layer over multiple Korean analyzers, affecting how researchers compare or operationalize analyzers in practice [21]. Relative to that landscape, this paper's main contribution is not proposing a new Korean analyzer itself, but delivering a native-first, cross-platform Flutter integration path for Kiwi with explicit reproducibility and deployment controls.

Table 1 summarizes commonly used Korean morphological analyzers/toolkits in practice, with repository-level adoption signals collected on February 17, 2026 via GitHub API snapshots [22].

Table 1: Representative Korean morphological analyzers and usage landscape (snapshot: 2026-02-17)

Analyzer / Toolkit	Primary runtime	Typical usage context	Adoption signal (snapshot)	Source
Kiwi	Native C++ (+ wrappers)	Production morphology/POS pipelines; offline embedding	GitHub stars: 671	[23]
MeCab-ko (+ mecab-ko-dic)	Native C++	Established tokenization/POS baseline; commonly used via KoNLPy Mecab path	MeCab upstream + Eunjeon lineage references; no canonical active GitHub org repo in this snapshot	[18, 19, 21]
Khaiii	Native C++/Python wrapper	Neural Korean analyzer for batch/server usage	GitHub stars: 1,448	[20]
Open Korean Text (Okt)	JVM/Scala (+ wrappers)	Social-text normalization/tokenization in JVM and Python workflows	GitHub stars: 656	[24, 21]
KOMORAN	JVM	Java production pipelines and KoNLPy-integrated experiments	GitHub stars: 311	[25, 21]
KoNLPy (toolkit)	Python wrapper hub	Unified interface over Kkma, Hannanum, Komoran, Mecab, Okt	GitHub stars: 1,486	[21, 26]

For benchmark methodology, GLUE and SuperGLUE establish widely reused task-oriented NLU evaluation structure, and Long Range Arena targets efficiency comparison under long-context settings [27, 28, 29]. For systems-level performance reporting, DAWNBench and MLPerf Inference provide complementary protocol ideas such as time-to-accuracy framing and cross-stack inference benchmarking [30, 31].

From a systems perspective, on-device ML discussions increasingly emphasize data locality, reduced dependency on always-on connectivity, and deployment practicality under edge constraints [32]. These themes align with the operational goals of `flutter_kiwi_nlp`: local inference path, deterministic runtime controls, and explicit provisioning/fallback mechanisms.

Privacy-preserving distributed training literature such as FedAvg also informs the broader motivation for keeping user data local when possible [33]. Although this plugin targets inference rather than training, the same locality principle reinforces its offline-first native execution model.

3 Background: Flutter and Kiwi

3.1 What Flutter Is

Flutter is an open-source UI toolkit for building applications from one Dart codebase across mobile, desktop, and web targets. Its rendering and widget model allows large parts of application logic to be shared, but low-level platform integration is still target-specific.

For systems like NLP analyzers, Flutter integration typically requires one of:

- platform channels (message-based bridge), or
- FFI plugins (direct native library interop from Dart).

`flutter_kiwi_nlp` uses FFI for native platforms and JS interop on web, which is why the plugin architecture is more complex than standard UI-only Flutter packages.

3.2 What Dart Is and Why It Was Created

Dart is an open-source, client-optimized language developed by Google and used as the language foundation of Flutter. The Dart project positions the language goal as productive multi-platform development paired with a flexible runtime platform.[1]

The practical reason this matters for plugin engineering is that Dart is designed for both development-time velocity and production deployment across multiple backends. In current official language positioning, this includes:

- fast iterative development workflows (for example, hot-reload-oriented tooling in Flutter),
- ahead-of-time native compilation for device/desktop targets, and
- web-target compilation paths (JavaScript and WebAssembly).[2]

For this plugin, these Dart properties directly motivate the design choice to publish one stable Dart API while internally dispatching to platform-specific runtime backends.

3.3 What Rust Is and Why It Is Mentioned

Rust is an open-source systems programming language designed around memory safety, strong compile-time guarantees, and predictable performance without a garbage collector.[3]

This repository is not primarily implemented in Rust: native runtime integration here is built on a Dart FFI layer and a C bridge that loads Kiwi artifacts. Rust is still worth documenting in this paper because reviewers often ask about Rust's role in modern WebAssembly ecosystems. In this plugin, consumer builds do not require a Rust toolchain; web execution depends on the distributed `kiwi-nlp` JS/WASM artifacts and native execution depends on platform libraries. This is an intentional interoperability tradeoff: the package prioritizes adopting upstream Kiwi distribution artifacts over introducing an additional language toolchain requirement for plugin consumers.

3.4 What WebAssembly (WASM) Is

WebAssembly (WASM) is a compact binary instruction format and execution model for stack-based virtual machines.[4] On the web it runs inside the browser sandbox, typically loaded by JavaScript bootstrap code, and enables near-native computational kernels while preserving browser security constraints.

In plugin context, WASM is not a replacement for Flutter itself; it is a backend execution target used by the web runtime path to execute Kiwi analysis logic in browser environments.

3.5 What Kiwi Is

Kiwi is a Korean morphological analysis engine distributed primarily as a native library and ecosystem artifacts. In this repository, Kiwi is consumed through:

- native dynamic libraries loaded by a C bridge on Android/iOS/macOS/Linux/Windows,
- the `kiwi-nlp` JavaScript/WASM package on web.

[1] https://dart.dev/overview
[2] https://dart.dev/
[3] https://www.rust-lang.org/
[4] https://webassembly.org/

Operationally, Kiwi performs segmentation and POS-tagged token analysis, and the plugin exposes that capability as: `create`, `analyze`, `addUserWord`, and `close`.

Model files are external artifacts (for example `cong.mdl`, `default.dict`, `typo.dict`) that must be present locally, packaged as Flutter assets, or obtained through archive fallback.

3.5.1 How Kiwi Models Are Trained (Upstream Summary)

Kiwi's upstream design separates dictionary/rule-driven candidate generation from statistical disambiguation. In practical terms, this means model training is centered on language-model estimation over morphologically analyzed corpora, while lexicon/rule components remain explicit resources [23, 34].

Upstream documentation and the Kiwi paper describe training data composition using large Korean morphological corpora, including Sejong and National Institute of Korean Language resources, and report model-family evolution from KNLM-focused scoring to Skip-Bigram and contextual n-gram embedding variants [23, 34].

For `flutter_kiwi_nlp`, the operational implication is explicit: this package consumes prebuilt Kiwi model artifacts for inference, and does not re-implement Kiwi training in the Flutter build/runtime path.

3.5.2 Kiwi Internal Architecture (From Upstream Source)

Upstream Kiwi implementation and public headers indicate that runtime decoding is organized as a morphology candidate graph/lattice search with language-model scoring, rather than a Transformer encoder stack [23, 35, 36, 37]. At the API/type level, model families are explicitly defined as: `knlm` (Kneser–Ney LM), `sbg` (Skip-Bigram), and `cong/congGlobal` (contextual N-gram embedding LM variants), with no Transformer model type in the exposed enum [35].

At inference-time, Kiwi keeps LM state while evaluating candidate paths over the morpheme graph and applies additional rule-based penalties/bonuses for morphological compatibility and punctuation/quote state handling [36, 37]. This is closer to a graph-decoding + statistical language-model architecture than to deep self-attention sequence encoding.

3.5.3 Conceptual Decoding Objective

For reviewer readability, the upstream behavior can be summarized as a lattice path optimization problem (conceptual/illustrative form; not a claim of reproducing every internal constant/feature exactly):

$$\pi^* = \arg\max_{\pi \in \Pi(G)} S(\pi),$$

where G is the morpheme lattice and $\Pi(G)$ is the set of valid paths. The path score is composed additively:

$$S(\pi) = \sum_{t=1}^{T(\pi)} \left[s_{\text{lm}}(e_t, h_{t-1}) \; + \; \lambda^\top \phi_{\text{rule}}(e_t, c_t) \; + \; \gamma^\top \phi_{\text{lex}}(e_t) \right].$$

Here, e_t denotes the selected candidate edge at step t, h_{t-1} is the LM state carried from prior steps, c_t encodes local context such as quote/punctuation conditions, and $\phi_{\text{rule}}, \phi_{\text{lex}}$ represent rule/lexicon-derived feature contributions.

State progression and pruning can be abstracted as:

$$h_t = F(h_{t-1}, e_t), \qquad \mathcal{B}_t = \text{TopK}(\mathcal{P}_t(G), K_{\text{beam}}, S),$$

where $\mathcal{P}_t(G)$ is the partial-path set at depth t, and $\mathcal{B}_t$ is the retained beam-like frontier with width K_{beam}.

The LM-family switch exposed by Kiwi model type can be written as:

$$s_{\text{lm}} = \begin{cases} s_{\text{knlm}}, & m = \text{knlm}, \\ s_{\text{knlm}} + \Delta_{\text{sbg}}, & m = \text{sbg}, \\ s_{\text{cong}}, & m = \text{cong}, \\ s_{\text{cong-global}}, & m = \text{congGlobal}. \end{cases}$$

This formulation explains why Kiwi behaves as graph decoding with explicit LM state and rule scoring, rather than Transformer self-attention inference.

3.5.4 Skip-Bigram and Contextual N-gram Scoring (Conceptual)

Because the plugin references Kiwi model families `sbg` and `cong/congGlobal`, it is useful to provide explicit conceptual scoring forms for reviewer intuition:

$$\Delta_{\text{sbg}}(e_t, h_{t-1}) \approx \sum_{k=1}^{\min(K_{\text{skip}}, t-1)} \alpha_k \, \psi_k(m_t, m_{t-k}) \,,$$

where m_t is the morpheme realized by edge e_t, k is skip distance, and α_k is a distance-dependent weight. The function ψ_k denotes skip-bigram interaction score components. Here K_{skip} is the maximum skip order considered.

A generic contextual n-gram embedding style score can be expressed as:

$$c_t = \sum_{j=1}^{\min(n-1, t-1)} W_j \, v(m_{t-j}), \qquad s_{\text{cong}}(e_t, h_{t-1}) \approx u(m_t)^\top c_t + b(m_t),$$

where $v(\cdot)$ and $u(\cdot)$ denote context/target embeddings and W_j encodes positional/context projection.

For the long-context variant, a practical abstraction is:

$$c_t^{\text{global}} = \sum_{j=1}^{\min(n_g - 1, t-1)} W_j^{(g)} \, v(m_{t-j}), \qquad s_{\text{cong-global}}(e_t, h_{t-1}) \approx u(m_t)^\top c_t^{\text{global}} + b_g(m_t),$$

with $n_g > n$ to reflect wider context coverage.

These equations are intentionally presented as conceptual abstractions to explain model-family behavior at paper level; exact implementation constants and feature composition are defined by upstream Kiwi internals and model artifacts. They are explanatory model-family formulations and are not used directly to compute the benchmark tables in Section 15.

Figure 1 summarizes this upstream-oriented view of Kiwi internals as used in the plugin context.

3.5.5 Why This Is Not a Transformer

Transformer-based analyzers typically center on stacked self-attention blocks and dense neural sequence representations. Kiwi's upstream architecture, by contrast, is designed around:

- lexicon/trie-driven candidate generation and morphological constraints,
- explicit path search with LM-state progression over candidates,

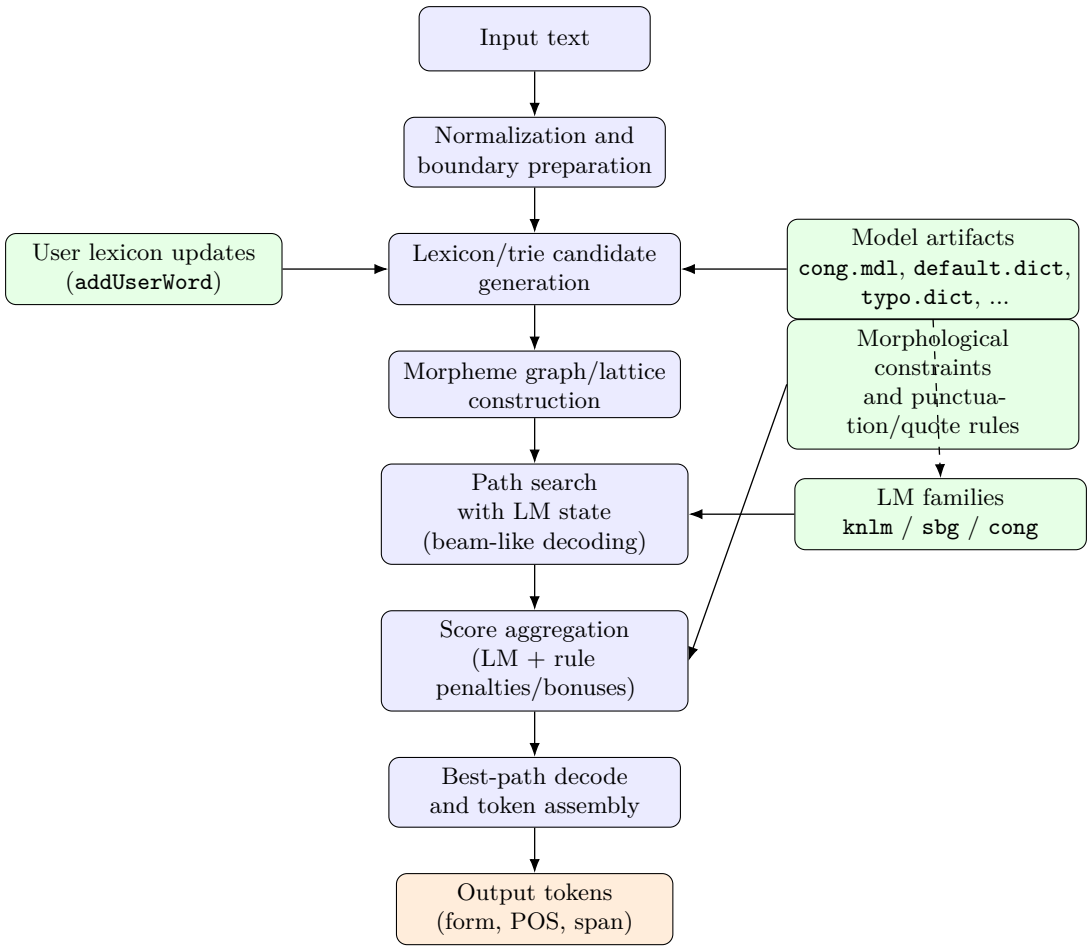

Figure 1: Conceptual Kiwi internal architecture based on upstream source inspection.

- lightweight N-gram-family scoring models (`knlm/sbg/cong`).

This distinction explains why Kiwi integration characteristics differ from Transformer runtimes: lower footprint and predictable CPU inference behavior are prioritized, while broad semantic representation power is not the primary design target in this component.

3.6 Why This Integration Is Non-Trivial

The plugin is not a direct wrapper around one runtime. It is an orchestration layer that has to keep semantics stable across different execution models:

- native C ABI with explicit memory ownership,
- browser JS/WASM with promise-based async semantics,
- platform-specific build systems and artifact formats.

4 Prior Ecosystem Survey: Existing Kiwi Wrappers

4.1 Survey Scope and Method

This paper includes an explicit prior-ecosystem survey to position `flutter_kiwi_nlp` against existing Kiwi bindings. The survey source is the upstream Kiwi repository README and linked binding projects, plus GitHub repository metadata snapshots collected on February 17, 2026.[5]

4.2 Observed Upstream Wrapper/Binding Landscape

Table 2 summarizes the wrappers and binding channels explicitly referenced upstream.

Table 2: Upstream Kiwi wrapper/binding survey (as documented by Kiwi)

Category	Location	Notes
C API	`include/kiwi/capi.h`	Core C interface for native embedding.
Prebuilt binaries	Kiwi releases page	Windows/Linux/macOS/Android library and model artifacts are distributed through release assets.
C# wrapper (official GUI usage)	https://github.com/bab2min/kiwi-gui	Upstream points to C# wrapper used by official GUI tooling.
C# wrapper (community)	https://github.com/EX3exp/NetKiwi	Community-contributed multiplatform C# wrapper linked by upstream README.
Python wrapper	https://github.com/bab2min/kiwipiepy	Officially documented Python3 API package.
Java binding	`bindings/java` in Kiwi repository	Java 1.8+ binding path documented upstream.
Android library	Kiwi release asset (`kiwi-android-VERSION.aar`)	Android NDK-based AAR distribution path documented upstream.

[5] https://github.com/bab2min/Kiwi

Category	Location	Notes
R wrapper	https://mrchypark.github.io/elbird/	Community-contributed R wrapper linked by upstream README.
Go wrapper	https://github.com/codingpot/kiwigo	Community Go wrapper linked by upstream README.
WebAssembly binding	`bindings/wasm` in Kiwi repository	JavaScript/TypeScript-facing WASM binding path documented upstream.

4.3 Gap Analysis and Motivation for This Work

The upstream survey shows broad language coverage around Kiwi, but it also shows a practical integration gap for Flutter package consumers:

- no upstream-listed first-class Dart/Flutter wrapper entry,
- no upstream-listed packaging path that unifies native (mobile/desktop) and web (WASM) behind one Dart API contract,
- no upstream-listed Flutter-specific build-hook automation for shipping platform artifacts in plugin workflows.

This gap is the direct motivation for `flutter_kiwi_nlp`: one Flutter-native package that preserves Kiwi backend capability while adding runtime abstraction, model resolution policy, and target-specific packaging automation needed by real Flutter deployments.

4.4 Quantified Maintenance Signals

To reduce purely descriptive bias, this revision adds repository-level maintenance signals collected from GitHub REST API snapshots on February 17, 2026 (`tool/benchmark/collect_wrapper_activity.py`). Table 3 reports latest release date, latest commit date, and commit velocity windows for each wrapper repository.

Table 3: Quantified wrapper maintenance signals (as-of 2026-02-17)

Wrapper	Repo	Latest release	Last commit	90d commits	365d commits	Stars
kiwi-gui	`bab2min/kiwi-gui`	2025-12-22 (57d)	2025-12-22 (57d)	10	18	14
NetKiwi	`EX3exp/NetKiwi`	N/A	2025-02-18 (364d)	0	1	2
kiwipiepy	`bab2min/kiwipiepy`	2025-12-15 (64d)	2025-12-25 (54d)	33	114	357
Kiwi Java binding	`bab2min/Kiwi`	2025-12-15 (64d)	2026-02-02 (15d)	30	279	671
Kiwi WASM binding	`bab2min/Kiwi`	2025-12-15 (64d)	2026-02-02 (15d)	30	279	671
elbird	`mrchypark/elbird`	2025-12-30 (49d)	2025-12-30 (49d)	37	44	34
kiwigo	`codingpot/kiwigo`	N/A	2025-10-15 (125d)	0	5	30

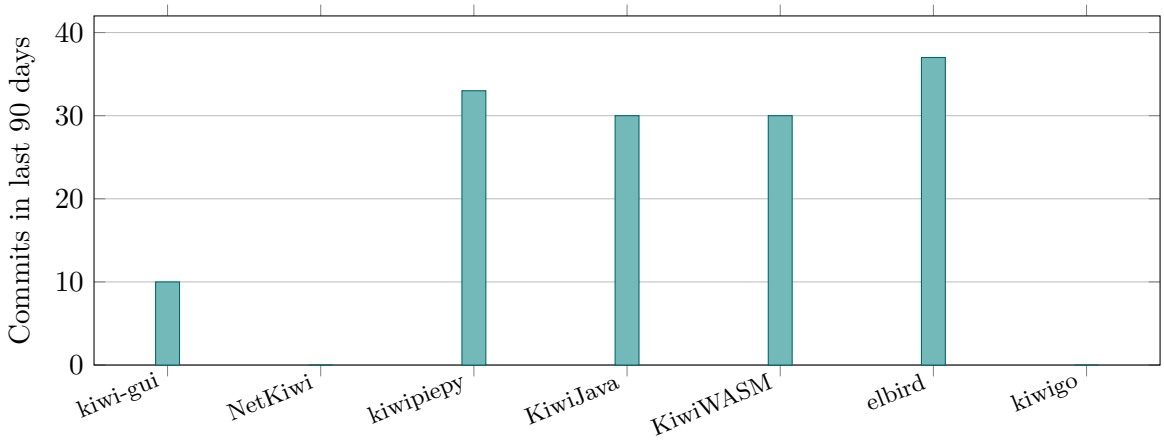

Figure 2: Recent wrapper activity (90-day commit count).

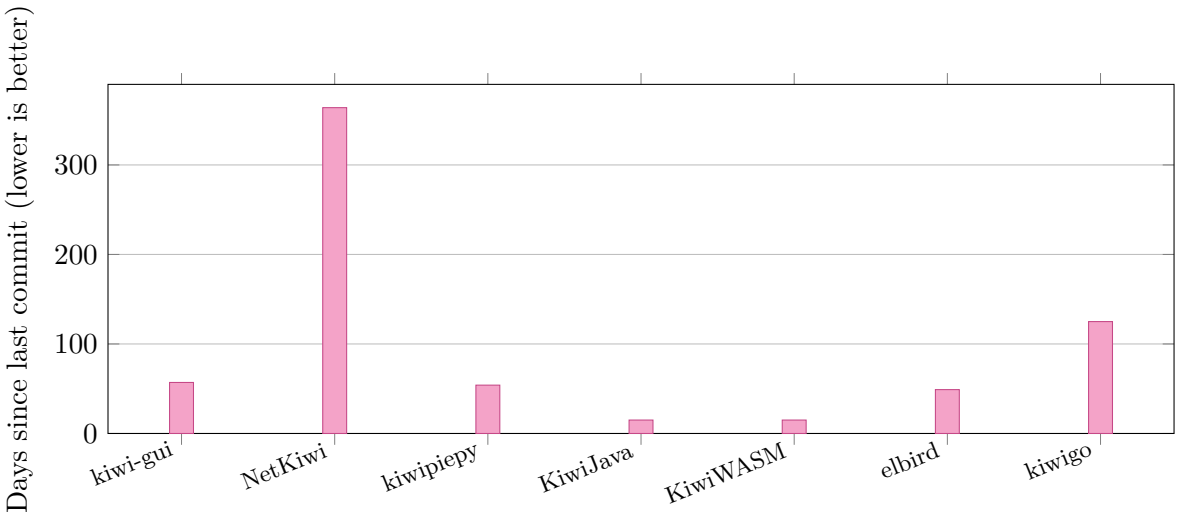

Figure 3: Repository recency profile for upstream wrappers.

4.5 Survey Evidence Limits

These maintenance signals are repository-level proxies. For Java and WASM, the metrics come from the shared Kiwi monorepo and do not isolate directory-level binding effort. Issue response latency and download velocity are also not yet included.

5 Design Goals and Non-Goals

5.1 Primary Goals

1. **Stable API contract**: identical Dart signatures across native, web, and unsupported stubs.
2. **Operational resilience**: layered model resolution, archive fallback, and explicit error propagation.
3. **Build-time ergonomics**: automatic preparation of missing Kiwi artifacts during platform builds.
4. **Typed outputs**: deterministic output schema (result → candidates → tokens).

5.2 Explicit Non-Goals

- Full feature parity with every upstream Kiwi API detail.
- Guaranteed throughput parity with Python-native `kiwipiepy`.
- Support for targets not declared in Flutter plugin metadata (for example, Fuchsia).

6 Supported Platform Matrix

Table 4 summarizes currently supported targets, execution backend, artifact strategy, and packaging hook.

Table 4: Current platform support matrix

Platform	Flutter plugin declaration	Runtime backend	Bundled artifact	Preparation hook
Android	Supported (`ffiPlugin`)	Native FFI + C bridge + `libkiwi.so`	`android/src/main/jniLibs/<abi>/libkiwi.so`	Gradle task `prepareKiwiAndroidLibs` bound to `preBuild`
iOS	Supported (`ffiPlugin`)	Native FFI + C bridge + Kiwi framework	`ios/Frameworks/Kiwi.xcframework`	Podspec hook (`prepare_command`) runs `tool/build_ios_kiwi_xcframework.sh`
macOS	Supported (`ffiPlugin`)	Native FFI + C bridge + libkiwi	`macos/Frameworks/libkiwi.dylib`	Podspec hook (`prepare_command`) runs `tool/build_macos_kiwi_dylib.sh`

Platform	Flutter plugin declaration	Runtime backend	Bundled artifact	Preparation hook
Linux	Supported (`ffiPlugin`)	Native FFI + C bridge + `libkiwi.so`	`linux/prebuilt/libkiwi.so`	CMake custom target `prepare_kiwi_linux_lib`
Windows	Supported (`ffiPlugin`)	Native FFI + C bridge + `kiwi.dll`	`windows/prebuilt/kiwi.dll`	CMake custom target `prepare_kiwi_windows_dll`
Web	Supported (web plugin class)	JS interop + `kiwi-nlp` WASM	Model files loaded by URL or archive bytes in memory	Runtime module/model loader in `kiwi_analyzer_web.dart`
Fuchsia	Not declared	Stub behavior	N/A	N/A

6.1 Architecture Coverage

- Android ABIs default to `arm64-v8a` and `x86_64`.
- iOS framework build includes `iphoneos arm64` and simulator `arm64/x86_64` slices.
- macOS default build targets `arm64` and `x86_64`.
- Linux supports host-driven architecture mapping (`x86_64`, `arm64`, `ppc64le`) in build scripts.
- Windows arch normalization supports `x64` (`x86_64`), `Win32` (`x86`), and `arm64`; prebuilt download path currently covers `x64`/`Win32`.

7 Public API Specification

7.1 Entry Point and Conditional Export

Public package entry point is `lib/flutter_kiwi_nlp.dart`. Runtime selection uses conditional exports:

```
export 'src/kiwi_analyzer_stub.dart'
    if (dart.library.io) 'src/kiwi_analyzer_native.dart'
    if (dart.library.js_interop) 'src/kiwi_analyzer_web.dart';
```

This guarantees a single import path for consumers while allowing runtime- specific implementation files.

7.2 Core Analyzer API

Table 5 lists the complete analyzer surface exposed to users.

Table 5: Core public analyzer API

Signature	Contract
`static Future<KiwiAnalyzer> create(...)`	Creates analyzer instance. Parameters include model path inputs, build/match flags, and thread count. Native resolves model path and opens FFI handle; web imports module and builds Kiwi instance (with fallback); stub throws unsupported exception.
`String get nativeVersion`	Returns backend version string. Native reads bridge-provided version. Web prefixes with `web/wasm`. Stub returns unsupported message.
`Future<KiwiAnalyzeResult> analyze(String text, {KiwiAnalyzeOptions options})`	Runs morphological analysis with candidate count and match options. Returns typed result object. Throws `KiwiException` on runtime failure or use-after-close.
`Future<void> addUserWord(String word, {String tag='NNP', double score=0.0})`	Registers user dictionary entry. Native invokes bridge function directly. Web stores word and rebuilds analyzer with accumulated `userWords`.
`Future<void> close()`	Releases resources and closes instance. Native closes bridge handle; web clears in-memory state; repeated use after close is rejected.

7.3 Options and Flags API

7.3.1 `KiwiBuildOption` constants

Table 6: Build option flags

Constant	Meaning
`integrateAllomorph`	Enable allomorph integration.
`loadDefaultDict`	Load default dictionary.
`loadTypoDict`	Load typo dictionary.
`loadMultiDict`	Load multi-word dictionary.
`modelTypeDefault`	Use backend default model type.
`modelTypeLargest`	Select largest model variant.
`modelTypeKnlm`	Select KNLM model variant.
`modelTypeSbg`	Select SBG model variant.
`modelTypeCong`	Select CONG model variant.
`modelTypeCongGlobal`	Select global CONG variant.
`defaultOption`	Recommended bundle: `integrateAllomorph | loadDefaultDict | loadTypoDict | loadMultiDict | modelTypeCong`.

7.3.2 `KiwiMatchOption` constants

Table 7: Match option flags

Constant	Meaning
`url`, `email`, `hashtag`, `mention`, `serial`	Detect corresponding token classes.
`normalizeCoda`	Normalize final consonants.
`joinNounPrefix`, `joinNounSuffix`, `joinVerbSuffix`, `joinAdjSuffix`, `joinAdvSuffix`	Join morphology according to POS-specific rules.
`splitComplex`	Split complex forms.
`zCoda`	Enable coda-related matching behavior.
`compatibleJamo`	Emit compatibility jamo style output.
`splitSaisiot`, `mergeSaisiot`	Control sai-sios split/merge behavior.
`all`	Baseline option bundle.
`allWithNormalizing`	`all` plus `normalizeCoda`.

7.3.3 `KiwiAnalyzeOptions`

`KiwiAnalyzeOptions` carries request-level options: `topN` (candidate count) and `matchOptions` (bitwise flags).

7.4 Result and Exception Models

Table 8: Public model types

Type	Fields and semantics
`KiwiToken`	`form`, `tag`, offsets (`start`, `length`), positions (`wordPosition`, `sentPosition`), confidence metrics (`score`, `typoCost`).
`KiwiCandidate`	`probability` and ordered `List<KiwiToken>` sequence.
`KiwiAnalyzeResult`	`List<KiwiCandidate>` candidates.
`KiwiException`	User-facing failure wrapper with message string; used for runtime, model, and lifecycle failures.

8 System Architecture Overview

Figure 4 summarizes how one Dart API fans out into native and web runtime lanes while preserving a shared contract.

8.1 Architecture Reading Notes

- The API entrypoint is intentionally singular (`KiwiAnalyzer`) to keep application code independent from target runtime.
- Native and web lanes diverge below the API surface, then converge back to the same typed Dart result models.

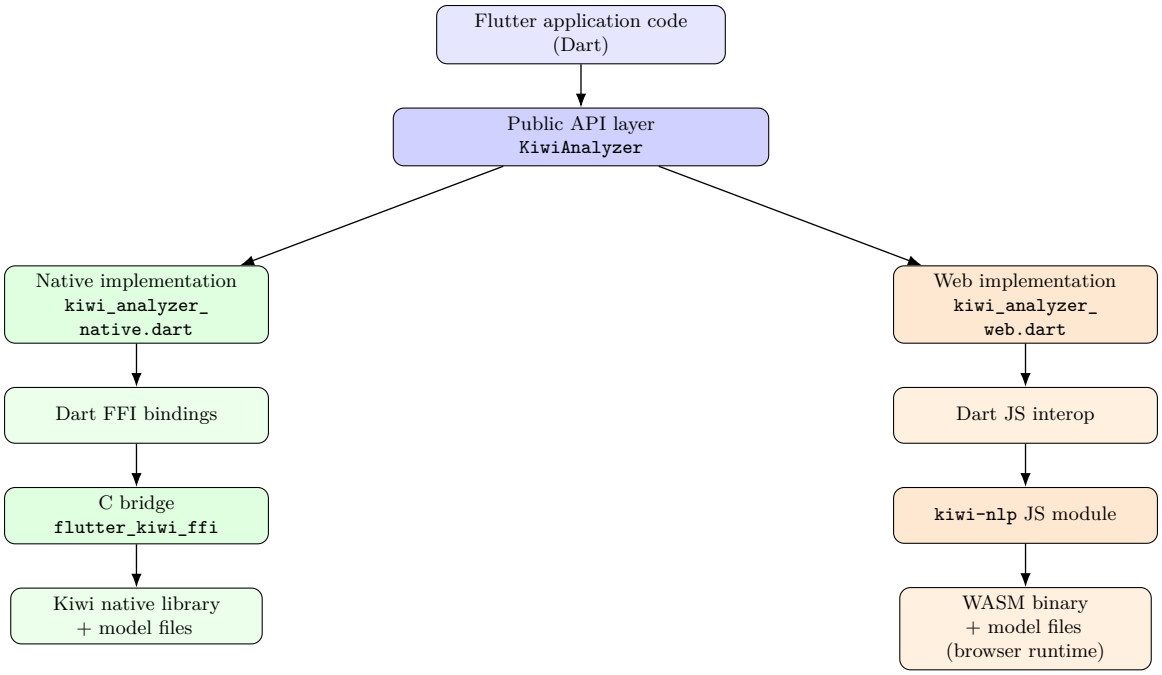

Figure 4: High-level plugin architecture for native and web paths.

- Model artifacts remain external data assets in both lanes, but loading strategies differ by runtime constraints.

9 Native Runtime Implementation

9.1 Layered Architecture

Native execution is deliberately split into three layers:

1. Dart analyzer implementation (`lib/src/kiwi_analyzer_native.dart`).
2. C bridge shared library (`src/flutter_kiwi_ffi.c`, header `src/flutter_kiwi_ffi.h`).
3. Kiwi shared library (loaded dynamically by the bridge).

Dart calls generated bindings in `lib/flutter_kiwi_ffi_bindings_generated.dart`; the bridge owns Kiwi symbol resolution and native error buffering.

9.2 Bridge ABI

The bridge exports a compact ABI: `init`, `close`, `analyze_json`, `add_user_word`, `free_string`, `last_error`, and `version`.

The key design choice is JSON transport for analyze output. This decouples Dart from Kiwi internal structs and simplifies ownership management across language boundaries.

9.3 Dart FFI Binding Layer

The native path uses Dart's `dart:ffi` for symbol binding and `package:ffi/ffi.dart` for UTF-8/pointer utilities. At runtime:

1. `lib/src/kiwi_analyzer_native.dart` opens the bridge dynamic library.
2. `lib/flutter_kiwi_ffi_bindings_generated.dart` binds exported C symbols.
3. `GeneratedKiwiNativeBindings` adapts generated calls behind the `KiwiNativeBindings` interface for testability and swap-in fakes.

This structure keeps API code independent from raw pointer handling while still using zero-copy native handles for analyzer lifecycle management.

9.3.1 FFI Type/Ownership Mapping

- `flutter_kiwi_ffi_handle_t*` → opaque `Pointer<flutter_kiwi_ffi_handle_t>` in Dart.
- `int32_t` and `float` → Dart `int`/`double` via generated `asFunction()` bridges.
- `char*` inputs are allocated from Dart strings, passed as UTF-8, and explicitly released on the Dart side.
- `char*` outputs from `analyze_json()` are bridge-owned and must be released by `flutter_kiwi_ffi_free_string()`.
- Error text from `last_error()` is thread-local on the bridge side and consumed as read-only C strings from Dart.

9.4 ffigen Generation Pipeline

Binding code is generated (not handwritten) from the canonical header `src/flutter_kiwi_ffi.h`. The generation source-of-truth is `ffigen.yaml`, which declares:

- binding class name: `FlutterKiwiFfiBindings`,
- entry/include header: `src/flutter_kiwi_ffi.h`,
- output file: `lib/flutter_kiwi_ffi_bindings_generated.dart`.

Regeneration command:

```
dart run ffigen --config ffigen.yaml
```

Why this paper uses `ffigen` instead of handwritten bindings:

- ABI drift resistance when C signatures evolve in `src/flutter_kiwi_ffi.h`.
- Deterministic regeneration from one header/config pair, which improves reviewability and incident forensics.
- Lower human error risk in repetitive signature plumbing (`NativeFunction`, `asFunction()`, pointer type mapping).
- Consistent symbol surface between C exports and Dart binding class (`FlutterKiwiFfiBindings`).
- Better maintenance cost profile for long-lived plugin evolution across Android/iOS/desktop targets.

9.4.1 Code-Generation Pipeline Diagram

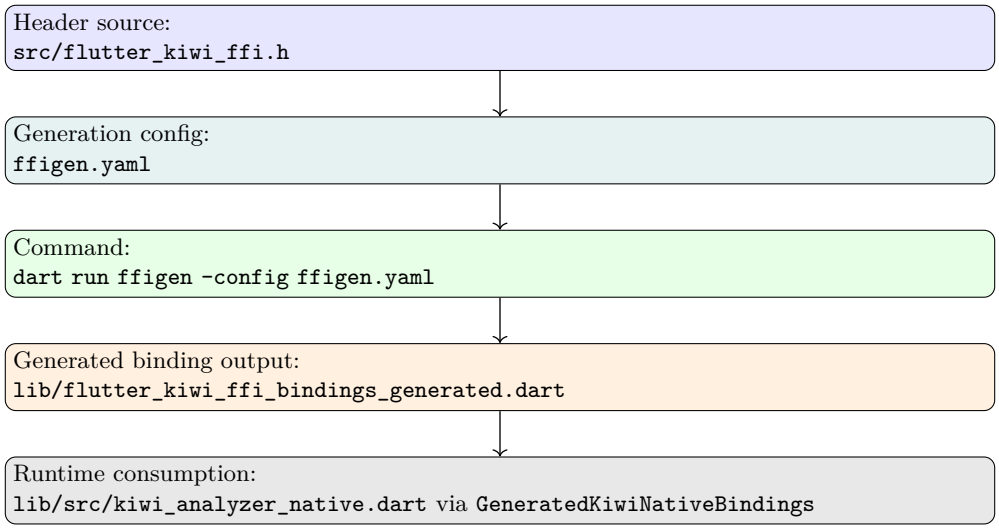

Figure 5: Dart FFI code-generation pipeline used in this plugin.

9.5 Dynamic Library Loading Strategy

9.5.1 Bridge loading (Dart side)

Candidate library names are tried in order by host platform:

- Apple: `flutter_kiwi_nlp.framework/flutter_kiwi_nlp`, then `flutter_kiwi_ffi.framework/flutter_kiwi_ffi`.
- Linux/Android: `libflutter_kiwi_ffi.so`, then `libflutter_kiwi_nlp.so`.
- Windows: `flutter_kiwi_ffi.dll`, then `flutter_kiwi_nlp.dll`.

9.5.2 Kiwi engine loading (C bridge side)

Bridge logic accepts optional override path via environment: `FLUTTER_KIWI_NLP_LIBRARY_PATH` (legacy alias: `FLUTTER_KIWI_FFI_LIBRARY_PATH`). Errors are captured in thread-local storage and exposed through `flutter_kiwi_ffi_last_error()`.

9.6 Analyzer Lifecycle and Memory Ownership

9.6.1 Creation

1. Resolve model directory using layered strategy.
2. Convert path to UTF-8 pointer.
3. Call `flutter_kiwi_ffi_init(...)`.
4. On null handle, read bridge error and throw `KiwiException`.
5. Free temporary path pointer.

9.6.2 Analysis

1. Ensure analyzer is open.
2. Convert input string to UTF-8 pointer.
3. Call `flutter_kiwi_ffi_analyze_json`.
4. Decode JSON into `KiwiAnalyzeResult`.
5. Release bridge-owned response string via `flutter_kiwi_ffi_free_string`.
6. Free input pointer.

9.6.3 Close

`close()` calls bridge close, marks instance closed, and rejects any subsequent operation with explicit use-after-close error messaging.

9.7 Native Model Resolution Algorithm

Native model resolution order is strict and deterministic:

1. `modelPath` argument.
2. `assetModelPath` argument.
3. Environment `FLUTTER_KIWI_NLP_MODEL_PATH` (legacy alias `FLUTTER_KIWI_FFI_MODEL_PATH`).
4. Compile-time define `FLUTTER_KIWI_NLP_ASSET_MODEL_PATH` (legacy alias `FLUTTER_KIWI_FFI_ASSET_MODEL_PATH`).
5. Built-in package asset candidates.
6. Download-and-cache fallback archive.

```
resolveModelPath(modelPath, assetModelPath):
  if modelPath is non-empty: return modelPath
  if assetModelPath is non-empty: return extractAssets(assetModelPath)
  if env MODEL_PATH is non-empty: return env MODEL_PATH
  if compile-time ASSET_MODEL_PATH is non-empty:
    return extractAssets(ASSET_MODEL_PATH)
  for candidate in builtInAssetCandidates:
    if assetExists(candidate):
      return extractAssets(candidate)
  return ensureDownloadedModel()
```

9.8 Native Archive Fallback and Integrity

Fallback archive handling includes:

- configurable archive URL, cache key, and SHA-256 define,
- download timeout guard,
- partial-file strategy with atomic rename,
- extraction retry behavior,
- required-file completeness and minimum-size checks.

Required model files are currently: `combiningRule.txt`, `cong.mdl`, `default.dict`, `dialect.dict`, `extract.mdl`, `multi.dict`, `sj.morph`, and `typo.dict`.

In this plugin, that list is treated as strictly required at initialization time. The native model-file check validates all entries in `kiwiModelFileNames`, including `extract.mdl`, before analyzer construction proceeds.

10 Web Runtime Implementation

10.1 Module and WASM Bootstrap

Web runtime imports `kiwi-nlp` using configurable module and WASM URLs. Promise-like JS values are normalized through helper wrappers and transformed into Dart exceptions on rejection.

10.2 How WASM Runs on the Web in This Plugin

The browser execution path is a staged pipeline:

1. Flutter web code calls `KiwiAnalyzer.create()`.
2. Dart JS interop loads the configured `kiwi-nlp` JavaScript module URL.
3. The JS module resolves and instantiates its WASM binary from `FLUTTER_KIWI_NLP_WEB_WASM_URL` (or module defaults).
4. Model files are loaded by URL map when available; otherwise archive fallback is triggered and extracted in memory.
5. A Kiwi runtime instance is constructed in browser memory and referenced by the Dart wrapper.
6. Each `analyze()` call crosses the Dart-to-JS boundary, executes in WASM-backed logic, then returns JS data converted into typed Dart models.

Important constraint: this plugin does not assume a browser-available native ABI or direct file-system model layout. All web model handling is network and in-memory oriented by design.

10.3 Web Build Modes

The web implementation supports two construction modes:

1. **Direct builder mode**: `KiwiBuilder.build()` returns a Kiwi object.
2. **API bridge mode**: when builder API is available, model files are loaded through API methods and analyzer commands are sent by `cmd(...)` with Kiwi instance id.

   This dual-path logic reduces fragility against upstream module behavior changes.

10.4 Web Fallback Algorithm

If URL-based model loading fails, the runtime attempts archive fallback:

1. try explicit archive URL define,
2. try default release URL composed from repo/version/name,
3. try GitHub releases API metadata lookup,
4. download bytes, optional SHA-256 verification,
5. extract required files in memory,
6. validate completeness/minimum size,
7. retry Kiwi build with in-memory model map.

10.5 Web-Specific Behavioral Notes

- `numThreads` and `matchOptions` are accepted in `create()` for API parity but are not applied at web create phase.
- `addUserWord()` triggers a rebuild with accumulated `userWords` rather than direct mutable insertion.
- `nativeVersion` reports `web/wasm <version>` format.

11 Build and Packaging Pipeline

11.1 Android

- Gradle task `prepareKiwiAndroidLibs` runs before `preBuild`.
- Default ABIs: `arm64-v8a`, `x86_64`.
- Script: `tool/build_android_libkiwi.sh`.
- Output: `android/src/main/jniLibs/<abi>/libkiwi.so`.

11.2 iOS

- Hook: `ios/flutter_kiwi_nlp.podspec` prepare command.
- Script: `tool/build_ios_kiwi_xcframework.sh`.
- Output: `ios/Frameworks/Kiwi.xcframework`.
- Includes device and simulator slices in one XCFramework.

11.3 macOS

- Hook: `macos/flutter_kiwi_nlp.podspec` prepare command.
- Script: `tool/build_macos_kiwi_dylib.sh`.
- Output: `macos/Frameworks/libkiwi.dylib`.
- Supports `arm64`/`x86_64` merge via `lipo`.

11.4 Linux

- Hook: `linux/CMakeLists.txt` custom target `prepare_kiwi_linux_lib`.
- Script: `tool/build_linux_libkiwi.sh`.
- Output: `linux/prebuilt/libkiwi.so`.
- Strategy: prebuilt download first, source build fallback.

11.5 Windows

- Hook: `windows/CMakeLists.txt` custom target `prepare_kiwi_windows_dll`.
- Script: `tool/build_windows_kiwi_dll.ps1`.
- Output: `windows/prebuilt/kiwi.dll`.
- Strategy: prebuilt download first (where available), source build fallback.

11.6 Shared Bridge Build

`src/CMakeLists.txt` builds shared bridge library `flutter_kiwi_ffi`. On Android, linker option `-Wl,-z,max-page-size=16384` is applied for Android 15 page-size compatibility; Linux links `dl` where required.

12 Configuration Surface

12.1 Native Configuration Keys

Table 9: Native runtime configuration keys

Key	Role
FLUTTER_KIWI_NLP_MODEL_PATH	Runtime environment model directory override.
FLUTTER_KIWI_NLP_ASSET_MODEL_PATH	Compile-time default packaged asset base path.
FLUTTER_KIWI_NLP_MODEL_ARCHIVE_URL	Compile-time archive URL for fallback download.
FLUTTER_KIWI_NLP_MODEL_ARCHIVE_SHA256	Optional checksum for archive integrity validation.
FLUTTER_KIWI_NLP_MODEL_CACHE_KEY	Cache directory discriminator for extracted archive assets.
FLUTTER_KIWI_NLP_LIBRARY_PATH	Runtime override for Kiwi shared library location.

Legacy aliases prefixed with FLUTTER_KIWI_FFI_... are retained for backward compatibility.

12.2 Web Configuration Keys

Table 10: Web runtime configuration keys

Key	Role
FLUTTER_KIWI_NLP_WEB_MODULE_URL	JavaScript module URL for kiwi-nlp.
FLUTTER_KIWI_NLP_WEB_WASM_URL	WASM binary URL passed to builder.
FLUTTER_KIWI_NLP_WEB_MODEL_BASE_URL	Base path used to construct per-file model URL map.
FLUTTER_KIWI_NLP_WEB_MODEL_ARCHIVE_URL	Optional explicit archive URL for fallback model download.
FLUTTER_KIWI_NLP_WEB_MODEL_ARCHIVE_SHA256	Optional archive checksum verification value.
FLUTTER_KIWI_NLP_WEB_MODEL_GITHUB_REPO	Repository slug used for release metadata fallback lookup.
FLUTTER_KIWI_NLP_WEB_MODEL_ARCHIVE_VERSION	Release tag used when composing default archive URL.
FLUTTER_KIWI_NLP_WEB_MODEL_ARCHIVE_NAME	Archive filename used in default/fallback URL generation.

13 Failure Taxonomy and Mitigations

Table 11: Failure categories and implemented mitigations

ID	Failure condition	Mitigation strategy
F1	Bridge library not loadable	Multi-candidate load attempt plus aggregated error diagnostics.

ID	Failure condition	Mitigation strategy
F2	Kiwi library unresolved or symbols missing	Runtime path override support and explicit bridge last-error propagation.
F3	Model path unresolved	Ordered fallback chain (arguments, env, defines, assets, archive).
F4	Archive download failure (timeout/HTTP)	Retry path and fallback URL sequence; actionable exception text.
F5	Archive integrity/completeness failure	SHA-256 verification option and minimum-size checks per required model file.
F6	Web module import/promise rejection	Promise normalization wrappers, timeout guards, and error contextualization.
F7	API use after close	Explicit lifecycle state check and deterministic `KiwiException`.
F8	Native library crash (for example segmentation fault in dependent native code)	No in-process isolation in current design; such crashes are fatal to the host Flutter process and must be mitigated by upstream native stability and artifact validation.

14 Performance Evaluation and Benchmark Interpretation

14.1 Evaluation Objective

The benchmark goal is to compare end-to-end analyzer behavior between `flutter_kiwi_nlp` and `kiwipiepy` under a shared corpus and roughly aligned loop structure. The result is intended as an engineering signal, not as a definitive language-model quality ranking.

14.2 Implemented Benchmark Pipeline

The repository executes comparison in three stages:

1. `tool/benchmark/run_compare.py` launches Flutter benchmark app (target: `example/lib/benchmark_main.dart`) and captures JSON payload from benchmark marker lines in stdout/device logs.
2. The same runner executes `tool/benchmark/kiwipiepy_benchmark.py` on Python runtime.
3. `tool/benchmark/compare_results.py` merges both JSON files into one markdown table with mean/stddev and per-trial raw snapshots.

Canonical corpus file: `example/assets/benchmark_corpus_ko.txt`. The runner supports repeated independent trials through `-trials`, producing both per-trial JSON files and aggregated report.

14.3 Reproducibility Manifest

Table 12: Execution metadata used for the benchmark section

Item	Value
Report snapshot date	February 18, 2026
Repository commit base	`0afe90607b9e35758b2836bf36b4fb6aea4af281`
Flutter SDK	`3.41.1` (framework revision `582a0e7c55`)
Dart SDK	`3.11.0`
Python runtime	`3.14.3`
`kiwipiepy`	`0.22.2`
Xcode toolchain	`Xcode 26.2 (17C52)`
Desktop host OS for baseline table	`macOS-15.7.4-arm64-arm-64bit-Mach-O`
Host CPU	`Apple M2 Pro`, 10 logical cores
Host memory	16 GiB (`17179869184` bytes)
Corpus hash (SHA-256)	`0fed28f4601cd577de8ad0f35fbe5bb1827e71931d3c8b19714a9976a12f3c9f`
Desktop benchmark shape	`trials=5, warmup_runs=3, measure_runs=15, top_n=1`
Desktop analyze impl	Flutter: `token_count` (primary), Kiwi: `analyze`

14.4 Recorded Configuration and Workload

From the benchmark trial sets included in this report (February 18, 2026):

- Desktop reference platform: macOS 15.7.4 arm64 (release, `n=5`, `warmup=3`, `measure=15`).
- Mobile platform A: iOS 26.2 simulator (iPhone 17, debug, `n=5`, `warmup=3`, `measure=15`).
- Mobile platform B: Android 16 emulator (API 36, release, `n=5`, `warmup=3`, `measure=15`).
- Corpus sentences: 40.
- Sample POS rows emitted per runtime: `sample_count = 10`.
- Total measured analyses per trial: desktop/iOS/Android all use `15 * 40 = 600`.
- `top_n`: 1.
- Desktop primary analyze path: Flutter `token_count`, Kiwi `analyze` (API path differs; see artifact caution note).
- Build options: `1039` (`integrateAllomorph`, `loadDefaultDict`, `loadTypoDict`, `loadMultiDict`, `modelTypeCong`).
- Analyze match options: `8454175` (`allWithNormalizing` bundle).
- Flutter analyzer setting: `numThreads = -1`.
- Python analyzer setting: `num_workers = -1`.
- iOS Simulator did not support `release/profile` in this setup, so iOS measurements were collected in `debug` mode.

- For mobile runs, `kiwipiepy` comparison values were collected on the host macOS runtime using the same corpus and benchmark loop settings.
- This revision introduces explicit layered timing decomposition (`pure_elapsed_ms`, `full_elapsed_ms`, `json_overhead_ms`) in addition to warm/cold summaries.

14.5 Mobile Test Environment Details

Table 13: iOS/Android benchmark environment details

Item	Value
iOS test target	`iPhone 17` simulator, UDID `<REDACTED\_SIM\_UDID>`
iOS runtime	`com.apple.CoreSimulator.SimRuntime.iOS-26-2`, iOS 26.2 (build `23C54`), supported architecture `arm64`
iOS device type	`com.apple.CoreSimulator.SimDeviceType.iPhone-17`, model identifier `iPhone18,3`
iOS CPU/memory context	`simctl spawn sysctl`: `hw.ncpu=10`, `hw.memsize=17179869184`. In this setup, simulator-reported values match host resources.
Android test target	`<REDACTED_EMULATOR_ID>`, AVD name `Pixel_9`, guest model `sdk_gphone64_arm64`
Android guest OS/ABI	Android 16 (`ro.build.version.sdk=36`), ABI `arm64-v8a`, hardware `ranchu`
Android virtual hardware config	AVD `config.ini`: `hw.cpu.ncore=4`, `hw.ramSize=2048`, resolution `1080x2424`, density `420`, data partition `6G`
Android runtime observation	`/proc/meminfo`: `MemTotal=2017772 kB`; `/proc/cpuinfo`: 4 processors visible in guest

Environment data was captured immediately after benchmark runs using `flutter devices`, `sysctl`, `xcrun simctl`, `adb`, and AVD `config.ini` inspection.

14.6 Metric Definitions

Both benchmark programs produce comparable derived metrics: **Cold start** in this paper means one-time analyzer initialization from a not-ready state (library/model load and analyzer construction). **Warm path** means steady-state `analyze()` execution after initialization has already completed.

These are reported separately because they answer different operational questions: cold start dominates short sessions and first-use latency, while warm path dominates sustained throughput/latency under repeated analysis calls. `Session-length effective throughput` then combines both to show end-to-end impact for finite request counts.

- warm metrics use only measured loop time (`elapsed_ms`) after warmup; cold init is excluded from throughput/latency rows.
- init (ms) = elapsed wall time around analyzer creation call (`KiwiAnalyzer.create(...)`).
- analyses/sec = `total_analyses / elapsed_seconds`

- chars/sec = `total_chars / elapsed_seconds`
- tokens/sec = `total_tokens / elapsed_seconds`
- avg latency (ms) = `elapsed_ms / total_analyses`
- avg token latency (us/token) = `elapsed_ms * 1000 / total_tokens`
- session-length effective analyses/sec (init included) for `N` analyses:

$$\texttt{effective\_analyses\_per\_sec} = \frac{\texttt{N}}{(\texttt{init\_ms}/1000) + (\texttt{N}/\texttt{analyses\_per\_sec})}.$$

Using notation aligned with benchmark payload fields:

$$S = \frac{\texttt{elapsed\_ms}}{1000}, \quad A = \texttt{total\_analyses}, \quad C = \texttt{total\_chars}, \quad U = \texttt{total\_tokens}.$$

$$T_{\text{analyses}} = \frac{A}{S}, \quad T_{\text{chars}} = \frac{C}{S}, \quad T_{\text{tokens}} = \frac{U}{S}.$$

$$L_{\text{avg-ms}} = \frac{\texttt{elapsed\_ms}}{A}, \quad L_{\text{token-us}} = \frac{1000 \cdot \texttt{elapsed\_ms}}{U}.$$

$$T_{\text{eff}}(N) = \frac{N}{(\texttt{init\_ms}/1000) + (N/T_{\text{analyses}})}.$$

14.7 Boundary-Decomposed Measurement (Pure vs Full Path)

To make performance interpretation more granular, the updated benchmark captures both:

- **Pure path**: analyzer compute path using `token_count`-based timing.
- **Full path**: full JSON analyze payload path (`json` path) including bridge serialization/parsing overhead.

Let E_{pure} and E_{full} denote elapsed times for pure/full paths on the same trial workload. Then:

$$\text{BoundaryLoss} = 1 - \frac{E_{\text{pure}}}{E_{\text{full}}} = 1 - \frac{T_{\text{full}}}{T_{\text{pure}}}.$$

Table 14: Layered boundary decomposition summary (mean ś stddev)

Environment	Pure aps	Full aps	Loss	Overhead/analysis (ms)	Overhead ratio
macOS desktop baseline	2546.24 ś 179.08	2260.25 ś 193.49	11.23%	0.0509 ś 0.0130	11.31 ś 1.69%
iOS simulator (debug)	2422.31 ś 190.28	2105.67 ś 93.14	13.07%	0.0609 ś 0.0202	12.84 ś 4.46%
Android emulator (release)	2214.79 ś 301.30	1877.74 ś 458.67	15.22%	0.1002 ś 0.1321	14.81 ś 18.16%

14.8 Recorded Result Summary: Desktop Baseline (macOS, n=5)

Table 15: macOS desktop warm-path summary (mean ś stddev)

Metric	flutter_kiwi_nlp	kiwipiepy	Ratio (Flutter/Kiwi)
Throughput (analyses/s)	2546.24 ś 179.08	3532.24 ś 167.92	0.72x
Throughput (chars/s)	86317.60 ś 6070.90	119743.02 ś 5692.35	0.72x
Throughput (tokens/s)	40994.49 ś 2883.23	56780.80 ś 2699.25	0.72x
Avg latency (ms, lower better)	0.39 ś 0.03	0.28 ś 0.01	1.39x slower
Avg token latency (us/token, lower better)	24.50 ś 1.89	17.64 ś 0.84	1.39x slower

14.9 Statistical Interval View: Desktop Warm Path (95% CI)

To make variability interpretation more explicit, Table 16 adds 95% confidence intervals for desktop baseline means (`n = 5`, two-sided `t`-interval, `df = 4`).

$$\mathrm{CI}_{95}(\mu) = \bar{x} \pm t_{0.975,\, n-1} \cdot \frac{s}{\sqrt{n}}.$$

Here s denotes the sample standard deviation across trials and $t_{0.975,\, n-1}$ is the Student-t quantile for two-sided 95% intervals. This interval assumes independent repeated trials under the same benchmark configuration.

Table 16: Desktop baseline 95% confidence intervals

Metric	flutter_kiwi_nlp (95% CI)	kiwipiepy (95% CI)
Throughput (analyses/s)	[2323.88, 2768.60]	[3323.75, 3740.74]
Throughput (chars/s)	[78779.58, 93855.62]	[112675.04, 126811.01]
Throughput (tokens/s)	[37414.49, 44574.50]	[53429.24, 60132.36]
Avg latency (ms)	[0.357, 0.432]	[0.267, 0.300]
Avg token latency (us/token)	[22.16, 26.84]	[16.60, 18.69]

14.10 Cold-Start Summary (Median + p95)

Table 17: Cold-start init summary reported separately from warm metrics

Environment	flutter_kiwi_nlp init (ms)	kiwipiepy init (ms)	Ratio (median)
macOS desktop baseline	median 1387.41, p95 1420.06	median 976.95, p95 1104.95	1.42x slower
iOS simulator (debug)	median 1358.27, p95 1496.69	median 677.10, p95 711.15	2.01x slower

Environment	flutter_kiwi_nlp init (ms)	kiwipiepy init (ms)	Ratio (median)
Android emulator (release)	median 3268.46, p95 4028.02	median 847.02, p95 1053.33	3.86x slower

14.11 Additional Mobile Runtime Measurements (iOS n=5, Android n=5)

14.11.1 iOS Simulator (debug)

Table 18: iOS simulator warm-path summary (mean ś stddev)

Metric	flutter_kiwi_nlp	kiwipiepy	Ratio (Flutter/Kiwi)
Throughput (analyses/s)	2422.31 ś 190.28	3352.24 ś 192.79	0.72x
Throughput (chars/s)	82116.41 ś 6450.47	113640.90 ś 6535.66	0.72x
Throughput (tokens/s)	38999.24 ś 3063.50	53887.24 ś 3099.14	0.72x
Avg latency (ms, lower better)	0.41 ś 0.03	0.30 ś 0.02	1.39x slower
Avg token latency (us/token, lower better)	25.76 ś 1.96	18.61 ś 1.11	1.38x slower

14.11.2 Android Emulator (release)

Table 19: Android emulator warm-path summary (mean ś stddev)

Metric	flutter_kiwi_nlp	kiwipiepy	Ratio (Flutter/Kiwi)
Throughput (analyses/s)	2214.79 ś 301.30	3510.74 ś 86.29	0.63x
Throughput (chars/s)	75081.44 ś 10213.94	119014.22 ś 2925.10	0.63x
Throughput (tokens/s)	35658.15 ś 4850.87	56435.21 ś 1387.05	0.63x
Avg latency (ms, lower better)	0.46 ś 0.07	0.28 ś 0.01	1.61x slower
Avg token latency (us/token, lower better)	28.52 ś 4.41	17.73 ś 0.44	1.61x slower

14.12 Session-Length Effective Throughput (Init Included)

Table 20 reports expected analyses/sec when a session runs for fixed analysis counts (`N` in {1, 10, 100, 1000}), combining cold init and warm throughput in one user-facing rate.

Table 20: Session-length effective throughput (mean ś stddev)

Environment	N	flutter_kiwi_nlp	kiwipiepy	Ratio (Flutter/Kiwi)
macOS desktop baseline	1	0.72 ś 0.01	1.06 ś 0.16	0.68x

Environment	N	flutter_kiwi_nlp	kiwipiepy	Ratio (Flutter/Kiwi)
macOS desktop baseline	10	7.16 ś 0.12	10.60 ś 1.57	0.68x
macOS desktop baseline	100	69.84 ś 1.24	103.15 ś 14.81	0.68x
macOS desktop baseline	1000	559.86 ś 15.62	813.32 ś 86.53	0.69x
iOS simulator (debug)	1	0.72 ś 0.04	1.48 ś 0.06	0.49x
iOS simulator (debug)	10	7.21 ś 0.42	14.72 ś 0.59	0.49x
iOS simulator (debug)	100	70.17 ś 4.04	141.59 ś 5.39	0.50x
iOS simulator (debug)	1000	556.34 ś 32.34	1025.03 ś 30.53	0.54x
Android emulator (release)	1	0.30 ś 0.05	1.16 ś 0.18	0.26x
Android emulator (release)	10	3.04 ś 0.46	11.53 ś 1.82	0.26x
Android emulator (release)	100	30.01 ś 4.50	111.90 ś 17.23	0.27x
Android emulator (release)	1000	267.39 ś 39.28	866.99 ś 106.91	0.31x

14.13 Visual Summary Charts (Desktop Baseline)

To improve reviewer readability, this paper includes normalized benchmark charts. Figures 6 and 7 use Kiwi mean as baseline index 100.

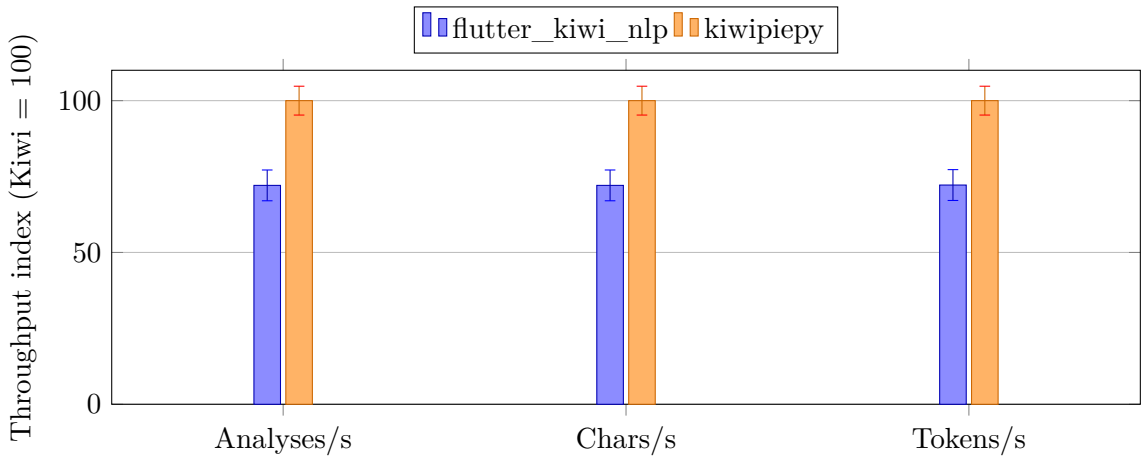

Figure 6: Throughput comparison for macOS baseline (normalized index, mean ś stddev).

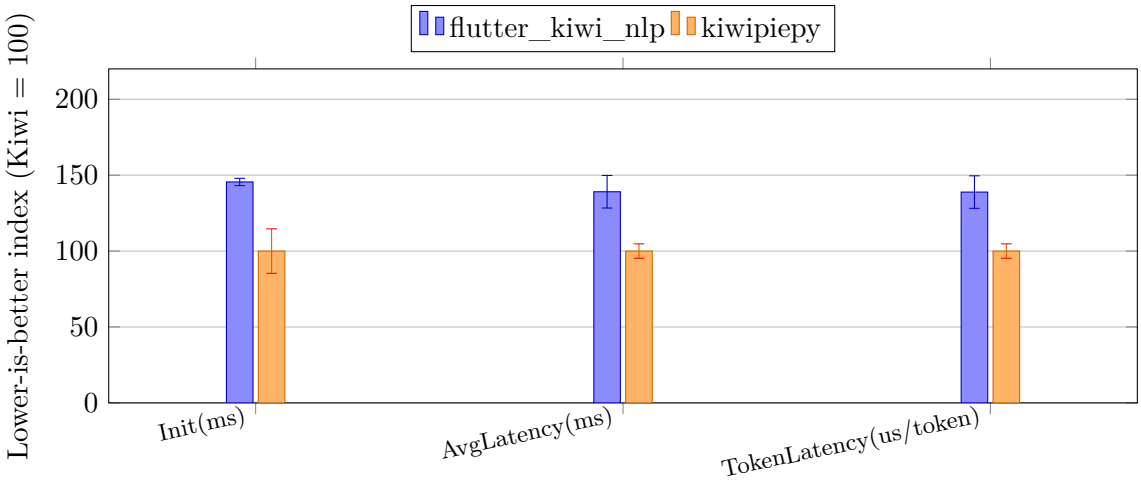

Figure 7: Lower-is-better metrics for macOS baseline (normalized index, mean ś stddev).

14.14 Per-Trial Raw Trace (Desktop Baseline)

Table 21: Per-trial raw snapshot for macOS baseline traceability

Trial	Flutter init (ms)	Kiwi init (ms)	Flutter analyses/s	Kiwi analyses/s
1	1423.93	810.77	2231.43	3316.65
2	1387.41	1067.51	2615.10	3487.51
3	1362.52	1114.31	2680.97	3669.04
4	1404.55	815.85	2608.83	3456.25
5	1385.88	976.95	2594.89	3731.77

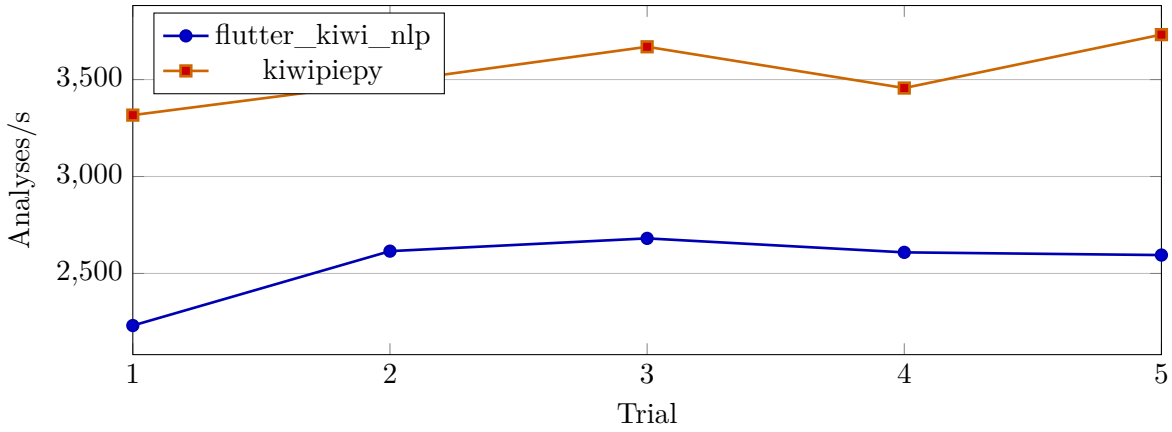

Figure 8: Trial-by-trial throughput trace for macOS baseline (analyses per second).

14.15 Detailed Interpretation (Honest Reading)

Across desktop and mobile trial sets, warm-path throughput and latency are consistently stable, and warm per-call latency remains sub-millisecond in these configurations (roughly 0.3–0.6 ms). This matches the practical impression that repeated in-app analysis is responsive after initialization. At the same time, repeated measurements still show slower init and lower throughput on the Flutter path relative to the Python reference runtime, with magnitude varying by environment (desktop native, iOS simulator debug, Android emulator release). Compared with the previous mobile rows in this report, the updated run profile shows improved Flutter warm throughput on both mobile targets (iOS: 2065.58 → 2422.31 analyses/s, +17.27%; Android: 1981.83 → 2214.79 analyses/s, +11.75%), while desktop warm throughput moved slightly downward (2659.19 → 2546.24 analyses/s, -4.25%). The revised methodology therefore surfaces a more nuanced trend: mobile warm-path behavior improved under the updated setup, but cross-runtime gaps still remain.

New boundary-decomposed instrumentation (Table 14) adds direct evidence for where overhead accumulates. Flutter boundary loss (full JSON path vs pure processing path) is 11.23% on desktop, 13.07% on iOS simulator, and 15.22% on Android emulator, with per-analysis overhead of roughly 0.05–0.10 ms.

Based on code inspection and payload values, the most likely contributors are:

1. **Boundary conversion overhead (evidence-based):** native Flutter path returns JSON from C bridge (`flutter_kiwi_ffi_analyze_json`) and then parses it in Dart (`jsonDecode`). Python path consumes native results via a different binding stack without this exact JSON roundtrip. In bridge code, JSON is assembled through repeated `sb_append(...)` operations and escaped field appends (`sb_append_json_escaped`), which implies dynamic allocation/reallocation and additional O(n)-scale string-copy work on top of base inference.

2. **Per-call async boundary cost (evidence-based):** Flutter benchmark loop awaits `Future` analysis call for each sentence, adding runtime overhead at Dart async/FFI boundaries.

3. **Residual cross-runtime semantic gap (evidence-based):** this benchmark now passes explicit build and match bitmasks to both runtimes, reducing prior configuration mismatch risk. However, output token totals still differ (Flutter 9660 vs Python 9645), indicating backend-wrapper semantic differences even under aligned knobs.

4. **Worker/thread auto mode mismatch (evidence-based):** `numThreads = -1` and `num_workers = -1` are both auto modes, but they are not guaranteed to map to equivalent parallel execution strategy.

5. **Model loading path variability (inference):** unless `-model-path` is forced, each runtime may initialize using different default lookup paths/caches, potentially affecting cold-start cost.

14.16 Threats to Validity

Current benchmark limitations that should be stated explicitly:

- Trial counts are improved but still modest for strong inference (desktop/iOS/Android all `n=5`).
- No explicit CPU pinning or thermal-state control.
- Bitmask parity is enforced, but internal backend semantics can still differ.
- Init metric combines multiple concerns (library load, model resolution, analyzer construction) rather than isolated micro-phases.
- iOS measurements in this report were collected on simulator in `debug` mode, because `release/profile` was unavailable in the current simulator setup.
- iOS simulator reports host-shared CPU/memory context (10 cores, 16 GiB RAM).
- These values are not equivalent to physical-device thermal or power constraints.
- Android measurements were collected on emulator, not on real devices.
- Android emulator guest was configured with 4 vCPUs and 2 GiB RAM, which can amplify startup variance and may not represent flagship physical devices.
- For mobile rows, `kiwipiepy` values come from host macOS runtime; therefore cross-runtime ratios on mobile rows should be interpreted as engineering reference, not strict same-device head-to-head evidence.
- Web runtime functionality is validated in tests, but sustained web throughput benchmark numbers are not included in this report.
- Gold-corpus quality evaluation in this revision covers two Kiwi-provided evaluation sets (191 sentences total). Broader domain coverage still requires larger and more heterogeneous corpora.

14.17 Gold-Corpus Linguistic Agreement Evaluation

To close the prior quality-evidence gap, this revision adds a reproducible gold-corpus evaluation using:

- `example/assets/gold\_eval\_web\_ko.txt` (158 sentences),
- `example/assets/gold\_eval\_written\_ko.txt` (33 sentences),
- shared options for both runtimes: `top_n=1` and `build_options=1039`.
- match options: `create_match_options=8454175` and `analyze_match_options=8454175`.

Evaluation command:

```
uv run python tool/benchmark/gold_corpus_compare.py \
  --device macos --mode release
```

Metrics are sequence-level agreements based on normalized Levenshtein distance: token agreement compares `form` sequences, POS agreement compares `form/tag` sequences, and sentence exact match requires full sequence identity. Let M be the number of evaluated sentences. For sentence i, g_i^{form} and p_i^{form} are gold/predicted token-form sequences, and g_i^{pair}, p_i^{pair} are gold/predicted `form/tag` pair sequences.

$$A_{\text{token}} = 1 - \frac{\sum_{i=1}^{M} d_{\text{Lev}}(g_i^{\text{form}}, p_i^{\text{form}})}{\sum_{i=1}^{M} \max(|g_i^{\text{form}}|, |p_i^{\text{form}}|)}.$$

$$A_{\text{pos}} = 1 - \frac{\sum_{i=1}^{M} d_{\text{Lev}}(g_i^{\text{pair}}, p_i^{\text{pair}})}{\sum_{i=1}^{M} \max\left(\left|g_i^{\text{pair}}\right|, \left|p_i^{\text{pair}}\right|\right)}.$$

$$E_{\text{token-seq}} = \frac{1}{M}\sum_{i=1}^{M} \mathbf{1}\left[g_i^{\text{form}} = p_i^{\text{form}}\right], \quad E_{\text{sentence}} = \frac{1}{M}\sum_{i=1}^{M} \mathbf{1}\left[g_i^{\text{pair}} = p_i^{\text{pair}}\right].$$

Table 22: Gold-corpus overall agreement (191 sentences, 7,990 gold tokens)

Metric	flutter_kiwi_nlp	kiwipiepy	Delta (pp)
Token agreement	88.39%	88.58%	-0.19
POS agreement	84.90%	85.55%	-0.65
Token-sequence exact match	3.66%	3.66%	+0.00
Sentence exact match	1.57%	2.62%	-1.05

Table 23: Per-dataset agreement breakdown

Dataset	Runtime	Token (%)	POS (%)	Token exact (%)	Sentence exact (%)
web_ko	flutter_kiwi_nlp	87.79	84.04	3.80	1.90
web_ko	kiwipiepy	88.03	84.80	3.80	2.53
written_ko	flutter_kiwi_nlp	90.86	88.42	3.03	0.00
written_ko	kiwipiepy	90.86	88.61	3.03	3.03

14.18 Interpretation of Gold-Corpus Results

Aggregate token and POS agreement are close between runtimes (sub-1pp gap), but strict sentence exact match remains low on both sides. This is expected for Korean morphological pipelines where small boundary/tag normalization differences across wrappers can fail whole-sentence exactness despite high token-level agreement. The combined predicted token totals (Flutter 8,076; Kiwi 8,103) confirm minor segmentation divergence under aligned option bitmasks.

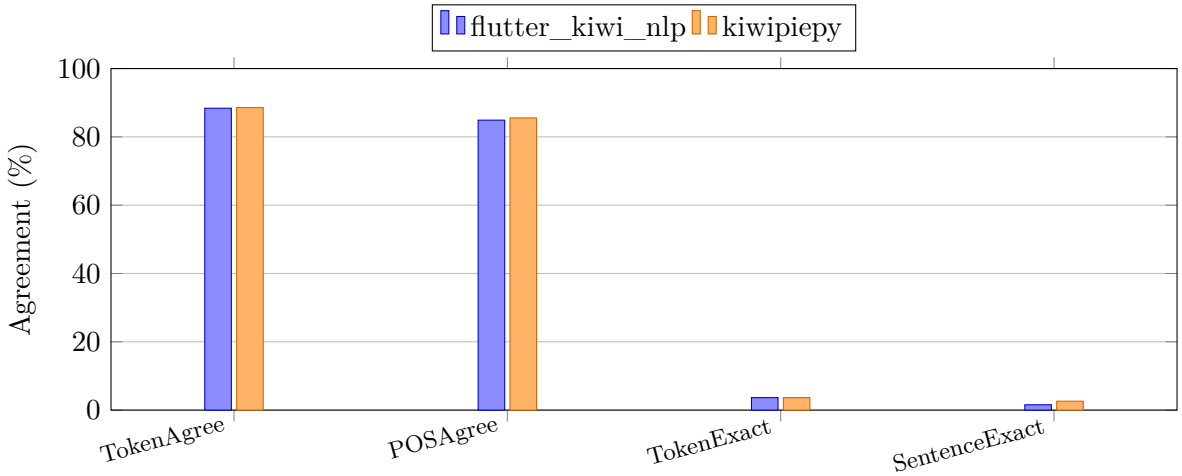

Figure 9: Overall gold-corpus agreement metrics across runtimes.

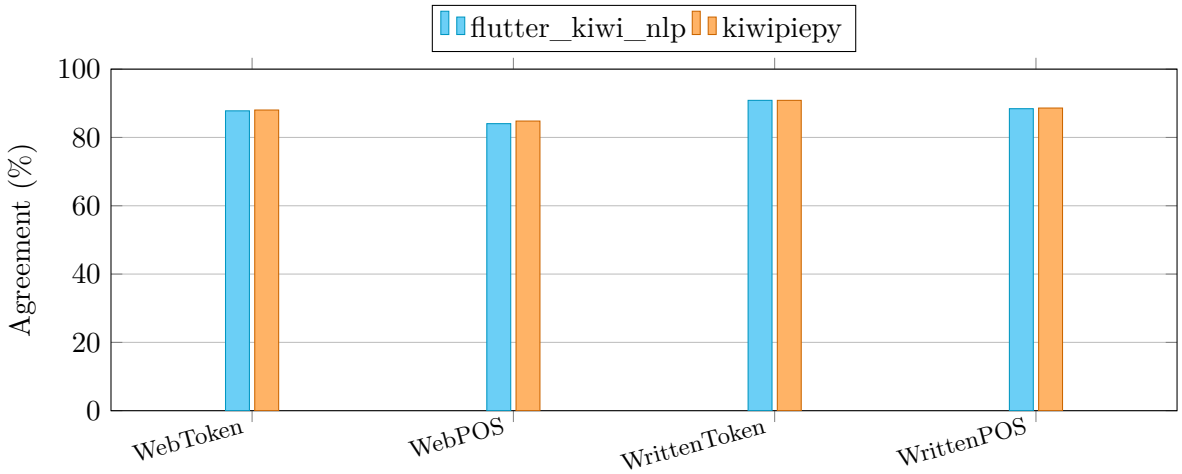

Figure 10: Per-dataset token/POS agreement profile.

Figure 11: Exact-match comparison on gold corpora (strict full-sequence criteria).

14.19 Recommended Protocol for Fairer Future Comparisons

1. Force same model assets by passing identical absolute `-model-path`.
2. Keep explicit bitmask parity and additionally validate per-sentence output equivalence for sampled cases.
3. Run repeated trials (for example 10+), report mean/median/stddev.
4. Separate measurements into cold init, warm init, and steady-state analysis throughput.
5. Record exact binary/model versions in output payload metadata.

15 Security and Supply-Chain Considerations

15.1 Archive Integrity

Both native and web fallback flows support optional SHA-256 verification, which is critical when model archives are downloaded at runtime.

15.2 Web CSP and CDN Trust Boundary

By default, web runtime bootstrap uses CDN-hosted `kiwi-nlp` module and WASM URLs (jsDelivr) unless overridden through `FLUTTER_KIWI_NLP_WEB_MODULE_URL` and `FLUTTER_KIWI_NLP_WEB_WASM_URL`. Production deployments therefore need explicit Content Security Policy (CSP) allowances for the selected module and WASM origins (or must self-host these artifacts under an already-allowed origin). This CDN trust boundary is distinct from model-archive integrity checks.

15.3 Dynamic Loading Risk

Environment-based dynamic library overrides are useful for controlled deployment, but should be restricted in hardened environments to avoid untrusted path injection.

15.4 Failure Transparency

The plugin intentionally prefers explicit hard failure with context-rich errors instead of silent fallback. This behavior improves diagnosability and reduces risk of undetected incorrect execution.

16 Maintainability Notes

16.1 Separation of Concerns

- Public API and models are compact and typed.
- Runtime-specific logic is isolated in dedicated files (native, web, stub).
- Shared model-file metadata and validation helpers are centralized in `kiwi_model_assets.dart`.

16.2 Testability Hooks

Native analyzer exposes explicit debug hooks for tests:

- binding factory override,
- archive URL/checksum override,
- HTTP client factory override.

These hooks allow deterministic tests for initialization and fallback branches.

17 Testing and Coverage Quality

17.1 Test Suite Scope

As of February 18, 2026, the test suite is organized in layered form:

- Core package layer (`test/`): 51 unit tests in 14 groups across 9 files.
- Example widget/golden layer (`example/test/`): 5 tests, including 3 screenshot(golden) assertions.
- Example integration/acceptance layer (`example/integration\_test/`): 3 end-to-end tests.

This yields 59 test declarations across repository test directories. Coverage focus in the core package layer remains strongest on API contract behavior, typed model parsing, option flags, error semantics, and native fallback logic. For macOS desktop execution, the three integration tests pass under serial invocation with explicit device pinning (`-d macos`).

Key tested areas include:

- public API export and unsupported-platform behavior,
- native analyzer lifecycle (`create`/`analyze`/ `addUserWord`/`close`),
- model path resolution and asset/archive fallback branches,
- JSON model decoding for `KiwiToken`, `KiwiCandidate`, and `KiwiAnalyzeResult`,
- option constant composition and exception formatting,
- acceptance flow over analyzer demo UI actions (analyze/clear/settings/POS-sheet),
- runtime smoke checks for `create` → `analyze` → `close`,
- native runtime-path probing on macOS FFI candidate resolution,
- screenshot(golden) stability checks for settings and POS dictionary sheets under a fixed mobile viewport.

17.2 Coverage Snapshot

Using `flutter test -coverage test` on February 18, 2026 (core package layer only):

- Raw line coverage: 350/374 = 93.58%.
- Filtered line coverage: 81/81 = 100.00%.

Figure 12: Coverage snapshot used in this paper.

17.3 Why Raw and Filtered Coverage Differ

The project includes a filtered coverage gate (`tool/check_coverage.sh`) that excludes:

- generated binding file:
  `lib/flutter_kiwi_ffi_bindings_generated.dart`,
- native runtime implementation:
  `lib/src/kiwi_analyzer_native.dart`,
- web runtime implementation:
  `lib/src/kiwi_analyzer_web.dart`.

The rationale is that parts of these files require runtime/platform integration contexts that are harder to exercise in hermetic unit tests. This makes filtered coverage useful as a strict gate for stable pure-Dart layers, but it must not be interpreted as full end-to-end runtime coverage.

17.4 Honest Quality Assessment

The present quality state is strong for deterministic unit-level contract tests and now includes initial acceptance/golden coverage in the example app. Remaining risk still concentrates in integration boundaries:

- native dynamic loading and ABI interaction across target platforms,
- web module import/WASM loading under browser/network constraints,
- platform build-hook behavior in real CI/device matrices,
- physical mobile device variability beyond simulator/emulator environments.

Therefore, line coverage should be treated as one quality indicator, not a complete reliability proof. Integration tests on actual target runtimes are the next quality multiplier for this plugin class.

18 Limitations and Future Work

18.1 Current Limitations

- Web runtime depends on module/WASM artifact availability unless self-hosted under application-controlled origins.

- Web deployments must satisfy CSP rules for module/WASM fetch origins; default CDN bootstrap therefore introduces an explicit supply-chain boundary.
- Native archive fallback requires network access and writable cache location when model assets are not bundled locally.
- Throughput can trail Python-native execution in current measurements; a major contributor is bridge-level JSON serialization overhead in the native C layer (dynamic allocation and string copy cost on hot paths).
- Native execution is in-process via FFI; a crash in dependent native code is fatal to the host Flutter process (no process isolation boundary).
- Platforms outside declared plugin matrix are unsupported.
- Current mobile benchmark rows use simulator/emulator environments (iOS debug simulator, Android emulator), not physical iOS/Android devices.
- Web throughput benchmark rows are not yet included in this report.

18.2 Recommended Future Work

1. **Bridge performance redesign:** replace or complement JSON bridge payloads with typed/native memory transfer to reduce serialization overhead and improve warm-path latency/throughput.
2. **Benchmark external validity:** add release/profile measurements on physical iOS and Android devices and include sustained Web throughput rows under explicit network/cache conditions.
3. **Operational hardening:** provide first-class self-hosting recipes for web module/WASM/model assets, with CSP and integrity examples.
4. **Reliability engineering:** evaluate optional crash-containment architectures for native execution (for example, process separation modes for fault-intolerant deployments).
5. **Compatibility governance:** expand CI validation across Kiwi versions/model families and platform-specific binary packaging combinations.
6. **Quality expansion:** add deterministic regression fixtures for hard tokenization/POS edge cases and broaden integration tests on real target runtimes.
7. **API evolution:** assess optional batch-analysis APIs for high-throughput service scenarios without breaking existing contracts.

19 Conclusion

`flutter_kiwi_nlp` contributes a native-first, cross-platform integration architecture that exposes one stable Dart API while spanning two runtime stacks (native FFI and web WASM). The implementation focus is practical deployment quality: explicit fallback order, reproducible build hooks, typed result contracts, and diagnosable failure behavior.

Empirical evidence in this paper indicates three core outcomes. First, runtime semantics are operationally aligned across targets under a shared API model. Second, the package is suitable for on-device and offline-native inference workflows once required model assets are provisioned locally. Third, current performance trade-offs are transparent: Flutter-path throughput can trail the Python baseline in measured settings, with bridge serialization overhead and runtime-context differences as key contributors. The updated benchmark revision further shows that mobile warm-path throughput can improve under tuned run settings while boundary-decomposed profiling still identifies a persistent JSON-path penalty (roughly 11–15% in this report).

This paper intentionally does not claim a new morphology model or SOTA language accuracy contribution. Its contribution is systems engineering: making Kiwi usable in production Flutter applications across Android, iOS, macOS, Linux, Windows, and Web with explicit operational contracts.

For practitioners, the key value is reduced integration risk when one codebase must ship across platforms while preserving deterministic Korean NLP behavior. For researchers and reviewers, the value is a reproducible, implementation-level specification that surfaces both strengths and unresolved constraints.

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," arXiv preprint arXiv:1810.04805, 2018. [Online]. Available: https://arxiv.org/abs/1810.04805. [Accessed: 2026-02-17].

[2] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations," arXiv preprint arXiv:1909.11942, 2019. [Online]. Available: https://arxiv.org/abs/1909.11942. [Accessed: 2026-02-17].

[3] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators," arXiv preprint arXiv:2003.10555, 2020. [Online]. Available: https://arxiv.org/abs/2003.10555. [Accessed: 2026-02-17].

[4] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," arXiv preprint arXiv:1910.01108, 2019. [Online]. Available: https://arxiv.org/abs/1910.01108. [Accessed: 2026-02-17].

[5] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "TinyBERT: Distilling BERT for Natural Language Understanding," arXiv preprint arXiv:1909.10351, 2019. [Online]. Available: https://arxiv.org/abs/1909.10351. [Accessed: 2026-02-17].

[6] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers," arXiv preprint arXiv:2002.10957, 2020. [Online]. Available: https://arxiv.org/abs/2002.10957. [Accessed: 2026-02-17].

[7] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices," arXiv preprint arXiv:2004.02984, 2020. [Online]. Available: https://arxiv.org/abs/2004.02984. [Accessed: 2026-02-17].

[8] F. N. Iandola, A. E. Shaw, R. Krishna, and K. W. Keutzer, "SqueezeBERT: What can computer vision teach NLP about efficient neural networks?" arXiv preprint arXiv:2006.11316, 2020. [Online]. Available: https://arxiv.org/abs/2006.11316. [Accessed: 2026-02-17].

[9] T. Tambe et al., "EdgeBERT: Sentence-Level Energy Optimizations for Latency-Aware Multi-Task NLP Inference," arXiv preprint arXiv:2011.14203, 2020. [Online]. Available: https://arxiv.org/abs/2011.14203. [Accessed: 2026-02-17].

[10] J. Xin, R. Tang, J. Lee, Y. Yu, and J. Lin, "DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference," arXiv preprint arXiv:2004.12993, 2020. [Online]. Available: https://arxiv.org/abs/2004.12993. [Accessed: 2026-02-17].

[11] F. Fusco, D. Pascual, P. Staar, and D. Antognini, "pNLP-Mixer: an Efficient all-MLP Architecture for Language," arXiv preprint arXiv:2202.04350, 2022. [Online]. Available: https://arxiv.org/abs/2202.04350. [Accessed: 2026-02-17].

[12] N. Goyal, "A comprehensive study of on-device NLP applications – VQA, automated Form filling, Smart Replies for Linguistic Codeswitching," arXiv preprint arXiv:2409.19010, 2024. [Online]. Available: https://arxiv.org/abs/2409.19010. [Accessed: 2026-02-17].

[13] S. Wang, A. Shenoy, P. Chuang, and J. Nguyen, "Now It Sounds Like You: Learning Personalized Vocabulary On Device," arXiv preprint arXiv:2305.03584, 2023. [Online]. Available: https://arxiv.org/abs/2305.03584. [Accessed: 2026-02-17].

[14] S. Lee, H. Jang, Y. Baik, S. Park, and H. Shin, "KR-BERT: A Small-Scale Korean-Specific Language Model," arXiv preprint arXiv:2008.03979, 2020. [Online]. Available: https://arxiv.org/abs/2008.03979. [Accessed: 2026-02-17].

[15] S. Park et al., "KLUE: Korean Language Understanding Evaluation," arXiv preprint arXiv:2105.09680, 2021. [Online]. Available: https://arxiv.org/abs/2105.09680. [Accessed: 2026-02-17].

[16] K. Yang, Y. Jang, T. Lee, J. Seong, H. Lee, H. Jang, and H. Lim, "KoBigBird-large: Transformation of Transformer for Korean Language Understanding," arXiv preprint arXiv:2309.10339, 2023. [Online]. Available: https://arxiv.org/abs/2309.10339. [Accessed: 2026-02-17].

[17] A. Matteson, C. Lee, Y.-B. Kim, and H. Lim, "Rich Character-Level Information for Korean Morphological Analysis and Part-of-Speech Tagging," arXiv preprint arXiv:1806.10771, 2018. [Online]. Available: https://arxiv.org/abs/1806.10771. [Accessed: 2026-02-17].

[18] T. Kudo, "MeCab: Yet Another Part-of-Speech and Morphological Analyzer," [Online]. Available: https://taku910.github.io/mecab/. [Accessed: 2026-02-17].

[19] Eunjeon Project, "Eunjeon Korean NLP project page (MeCab-ko lineage)," Blog. [Online]. Available: http://eunjeon.blogspot.com/. [Accessed: 2026-02-17].

[20] Kakao Corp., "Khaiii repository," GitHub. [Online]. Available: https://github.com/kakao/khaiii. [Accessed: 2026-02-17].

[21] KoNLPy Contributors, "KoNLPy documentation," [Online]. Available: https://konlpy.org/en/latest/. [Accessed: 2026-02-17].

[22] GitHub Docs, "GitHub REST API documentation." [Online]. Available: https://docs.github.com/en/rest. [Accessed: 2026-02-17].

[23] bab2min, "Kiwi repository," GitHub. [Online]. Available: https://github.com/bab2min/Kiwi. [Accessed: 2026-02-17].

[24] Open Korean Text Contributors, "Open Korean Text repository," GitHub. [Online]. Available: https://github.com/open-korean-text/open-korean-text. [Accessed: 2026-02-17].

[25] SHINEWARE, "KOMORAN repository," GitHub. [Online]. Available: https://github.com/shineware/KOMORAN. [Accessed: 2026-02-17].

[26] KoNLPy Contributors, "KoNLPy repository," GitHub. [Online]. Available: https://github.com/konlpy/konlpy. [Accessed: 2026-02-17].

[27] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding," arXiv preprint arXiv:1804.07461, 2018. [Online]. Available: https://arxiv.org/abs/1804.07461. [Accessed: 2026-02-17].

[28] A. Wang et al., "SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems," arXiv preprint arXiv:1905.00537, 2019. [Online]. Available: https://arxiv.org/abs/1905.00537. [Accessed: 2026-02-17].

[29] Y. Tay et al., "Long Range Arena: A Benchmark for Efficient Transformers," arXiv preprint arXiv:2011.04006, 2020. [Online]. Available: https://arxiv.org/abs/2011.04006. [Accessed: 2026-02-17].

[30] C. Coleman et al., "Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark," arXiv preprint arXiv:1806.01427, 2018. [Online]. Available: https://arxiv.org/abs/1806.01427. [Accessed: 2026-02-17].

[31] V. J. Reddi et al., "MLPerf Inference Benchmark," arXiv preprint arXiv:1911.02549, 2019. [Online]. Available: https://arxiv.org/abs/1911.02549. [Accessed: 2026-02-17].

[32] S. S. Chawathe et al., "Tiny Machine Learning: Progress and Futures," arXiv preprint arXiv:2403.19076, 2024. [Online]. Available: https://arxiv.org/abs/2403.19076. [Accessed: 2026-02-17].

[33] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," arXiv preprint arXiv:1602.05629, 2016. [Online]. Available: https://arxiv.org/abs/1602.05629. [Accessed: 2026-02-17].

[34] E. Park and S. Park, "Kiwi: A Study on Developing a Korean Morphological Analyzer," *Korean Journal of Digital Humanities*, vol. 1, no. 1, pp. 109–135, 2018. [Online]. Available: https://accesson.kr/kjdh/v.1/1/109/43508. [Accessed: 2026-02-17].

[35] bab2min, "Kiwi model/type definitions (`include/kiwi/Types.h`)," GitHub. [Online]. Available: https://github.com/bab2min/Kiwi/blob/main/include/kiwi/Types.h. [Accessed: 2026-02-17].

[36] bab2min, "Kiwi C++ API and model includes (`include/kiwi/Kiwi.h`)," GitHub. [Online]. Available: https://github.com/bab2min/Kiwi/blob/main/include/kiwi/Kiwi.h. [Accessed: 2026-02-17].

[37] bab2min, "Kiwi path-scoring implementation (`src/PathEvaluator.hpp`)," GitHub. [Online]. Available: https://github.com/bab2min/Kiwi/blob/main/src/PathEvaluator.hpp. [Accessed: 2026-02-17].

A Appendix A: End-to-End Usage Example

```
import 'package:flutter_kiwi_nlp/flutter_kiwi_nlp.dart';

Future<void> runSample() async {
  final KiwiAnalyzer analyzer = await KiwiAnalyzer.create(
    numThreads: -1,
    buildOptions: KiwiBuildOption.defaultOption,
    matchOptions: KiwiMatchOption.allWithNormalizing,
  );

  final KiwiAnalyzeResult result = await analyzer.analyze(
    'sample sentence for analysis',
    options: const KiwiAnalyzeOptions(topN: 1),
  );

  for (final KiwiCandidate candidate in result.candidates) {
    for (final KiwiToken token in candidate.tokens) {
      // Use token.form, token.tag, token.start, token.length, ...
    }
  }

  await analyzer.addUserWord('newword', tag: 'NNP', score: 1.0);
  await analyzer.close();
}
```

B Appendix B: Native and Web Runtime Equivalence Notes

- Method signatures are intentionally identical across native and web implementations.
- Result model shape is normalized to `KiwiAnalyzeResult -> KiwiCandidate -> KiwiToken`.
- Lifecycle rules are consistent: use-after-close triggers `KiwiException`.
- Web create-time handling of `numThreads`/`matchOptions` differs (accepted for parity, not applied at creation).

C Appendix C: Reproducibility Statement

This report describes repository state and behavior as observed on February 18, 2026. Build scripts, dependency versions, defaults, and benchmark numbers may change in future releases. Formalized benchmark/testing pseudocode is provided in Appendix D.

Minimal command sequence used for benchmark reproduction: The measured mobile rows in this revision are simulator/emulator runs; physical device command templates are included below for direct extension.

```
# Desktop baseline (macOS)
uv run python tool/benchmark/run_compare.py \
  --device macos --mode release --trials 5 \
  --warmup-runs 3 --measure-runs 15
```

```
# iOS simulator row (debug in this setup)
uv run python tool/benchmark/run_compare.py \
  --device "iPhone 17" --mode debug --trials 5 \
  --warmup-runs 3 --measure-runs 15

# Android emulator row (release)
uv run python tool/benchmark/run_compare.py \
  --device <android_emulator_id> --mode release --trials 5 \
  --warmup-runs 3 --measure-runs 15

# iOS physical device row (release; requires attached iPhone)
uv run python tool/benchmark/run_compare.py \
  --device <ios_physical_device_id> --mode release --trials 5 \
  --warmup-runs 3 --measure-runs 15

# Android physical device row (release; requires attached phone)
uv run python tool/benchmark/run_compare.py \
  --device <android_physical_device_id> --mode release --trials 5 \
  --warmup-runs 3 --measure-runs 15

# Core package unit tests (plugin root)
flutter test test

# Example screenshot(golden) tests (example app)
cd example
flutter test test/kiwi_ui_golden_test.dart

# Example integration/acceptance tests (run serially to avoid build.db lock)
flutter test integration_test/kiwi_runtime_smoke_test.dart -d macos
flutter test integration_test/kiwi_acceptance_flow_test.dart -d macos
flutter test integration_test/kiwi_native_runtime_path_test.dart -d macos
cd ..

# Gold-corpus agreement
uv run python tool/benchmark/gold_corpus_compare.py \
  --device macos --mode release

# Wrapper activity quantification snapshot
uv run python tool/benchmark/collect_wrapper_activity.py \
  --as-of-date 2026-02-17

# Environment capture used for Table "iOS/Android benchmark environment details"
flutter devices
sw_vers
sysctl -n machdep.cpu.brand_string
sysctl -n hw.ncpu
sysctl -n hw.memsize
xcodebuild -version
xcrun simctl list devices --json
xcrun simctl list runtimes --json
xcrun simctl list devicetypes --json
xcrun simctl spawn <ios_simulator_udid> /usr/sbin/sysctl -n hw.ncpu
xcrun simctl spawn <ios_simulator_udid> /usr/sbin/sysctl -n hw.memsize
adb -s <android_emulator_id> emu avd name
```

```
adb -s <android_emulator_id> shell getprop ro.build.version.sdk
adb -s <android_emulator_id> shell cat /proc/meminfo
adb -s <android_emulator_id> shell cat /proc/cpuinfo
cat ~/.android/avd/Pixel_9.avd/config.ini
```

Primary generated artifacts:

- `benchmark/results/macos_release_t5_token_count_s10_v2/comparison.md`
- `benchmark/results/ios_debug_t5_token_count_s10_v2/comparison.md`
- `benchmark/results/android_release_t5_token_count_s10_v2/comparison.md`
- `benchmark/results/macos_release_t5_token_count_s10_v2/flutter_kiwi_benchmark_trials.json`
- `benchmark/results/macos_release_t5_token_count_s10_v2/kiwipiepy_benchmark_trials.json`
- `benchmark/results/gold_eval/comparison.md`
- `benchmark/results/gold_eval/flutter_overall.json`
- `benchmark/results/gold_eval/kiwipiepy_overall.json`
- `benchmark/results/wrapper_activity/wrapper_activity.json`
- `benchmark/results/wrapper_activity/wrapper_activity.md`
- per-trial JSON files for each runtime in `benchmark/results/`.
- screenshot(golden) baselines in `example/test/goldens/`.

D Appendix D: Benchmark and Test Execution Pseudocode

Algorithm 1 Cross-runtime benchmark orchestration (`tool/benchmark/run_compare.py`)

Input: device d, mode m, corpus p, trials n, warmup w, measure r
Output: comparison report and per-trial JSON artifacts for Flutter and Python

Resolve output directory key from (d, m, n, w, r) and options.
for $i \leftarrow 1$ **to** n **do**
    Launch Flutter benchmark target `example/lib/benchmark_main.dart` on (d, m).
    Wait for marker `KIWI_BENCHMARK_JSON=` in stdout.
    if marker is missing on Android **then**
        Scan `adb logcat` and reconstruct payload from base64 chunks.
    end if
    Persist Flutter trial payload JSON.
    Run Python benchmark (`tool/benchmark/kiwipiepy_benchmark.py`) with the same corpus and option bitmasks.
    Persist Python trial payload JSON.
end for
Aggregate trial statistics (mean/stddev, ratio, per-trial snapshots).
Generate `comparison.md` via `tool/benchmark/compare_results.py`.

Algorithm 2 Gold-corpus agreement evaluation (`tool/benchmark/gold_corpus_compare.py`)

Input: asset list G, device d, mode m, top-N, build/match options
Output: token/POS agreement metrics and dataset/overall JSON outputs

for all $g \in G$ **do**
    Load tab-separated $(sentence, gold\_tokens)$ entries from g.
    Run Flutter evaluator target `example/lib/gold_eval_main.dart`.
    Parse `KIWI_GOLD_EVAL_JSON=` payload.
    Run Python evaluator (`kiwipiepy`) with aligned options.
    Align outputs per sentence (top-1 candidate) and compute: token agreement, POS agreement, token-pair exact, sentence exact.
    Persist per-dataset JSON summaries.
end for
Aggregate micro/macro overall metrics across datasets.
Emit `benchmark/results/gold_eval/comparison.md`, `flutter_overall.json`, and `kiwipiepy_overall.json`.

Algorithm 3 Unit-test and coverage gate pipeline (`tool/check_coverage.sh`)

Input: repository root R, coverage threshold τ (default 100%)
Output: pass/fail decision with filtered coverage report

Execute `flutter test -coverage test`.
if `coverage/lcov.info` does not exist **then**
    fail with missing-report error.
end if
Filter LCOV records, excluding generated/native/web runtime files.
Compute filtered coverage: `coverage` $= 100 \times$ `LH`/`LF`.
Write filtered report to `coverage/lcov.filtered.info`.
if `coverage` $< \tau$ **then**
    fail coverage gate.
else
    pass and report filtered percentage.
end if