

# navi-sanitize

Deterministic Input Sanitization for Untrusted Unicode Text

Nelson Spence

Project Navi LLC • Austin, Texas

<https://github.com/Project-Navi/navi-sanitize>

March 2026 • v1.0 • 0a5b50e

---

## Abstract

Untrusted text entering AI agent pipelines, template engines, and identity systems carries invisible attacks: homoglyph substitution that bypasses keyword filters, zero-width characters that split delimiters, Unicode Tag block characters that encode instructions tokenizers read but humans cannot see, and bidirectional overrides that reorder displayed text. These attacks operate below the layer where existing defenses—HTML escaping, schema validation, probabilistic detection—are designed to function. **navi-sanitize** is a zero-dependency Python library that removes these vectors deterministically at the input boundary. A six-stage pipeline removes null bytes, strips 411 invisible characters, applies NFKC normalization, replaces 54 targeted homoglyphs with Latin equivalents, re-normalizes to guarantee idempotency, and runs a pluggable context-specific escaper—producing identical output for identical input, with zero false positives on legitimate Unicode text. Clean-path latency is 2.8 $\mu$ s per string.

## 1 The Character-Level Trust Gap

Every layer in a modern text-processing stack assumes the characters it receives are what they appear to be. HTML escapers check for `<` and `>`. Schema validators check string length and pattern. Keyword filters check for `system` and `ignore previous instructions`. LLM guardrails pattern-match on known injection phrases.

None of them see Cyrillic `а` (U+0430) where Latin `a` should be. None of them see the zero-width space splitting `{` from `{` into two separate characters that rejoin after filtering. None of them see the invisible ASCII characters encoded in the Unicode Tag block (U+E0000–U+E007F) that a tokenizer will faithfully read as `ignore previous instructions` while a human sees blank space.

This is not a detection problem. Detection requires identifying malicious intent from ambiguous signals, and published evaluations of homoglyph detection consistently report high bypass rates against state-of-the-art classifiers. The character-level attack surface is adversarial by nature: every new detector creates a new evasion target.

The correct analogy is parameterized queries. The industry did not solve SQL injection by building better SQL-in-string detectors. It solved SQL injection by making the attack *structurally impossible*—separating data from code at the boundary. **navi-sanitize** applies the same principle to text: clean the characters at the input boundary, deterministically, before anything downstream touches them.

## 2 Why This Matters for Agent Developers

AI agent architectures route untrusted text through multiple trust boundaries in a single request: user input flows into system prompts, RAG retrieval context, tool call arguments, and structured output parsing. Each boundary is an injection surface, and Unicode-level attacks are particularly effective against LLM pipelines for three reasons.

**Tokenizers are faithful readers.** BPE and SentencePiece tokenizers process Unicode codepoints, not rendered glyphs. A Tag block sequence encoding `ignore previous instructions` produces tokens that the model processes as natural language, even though no human reviewer—and no text-based content filter—can see the content. This is not a theoretical concern; tag smuggling has been demonstrated against production LLM systems.

**Keyword filters operate on surface forms.** Guardrails that block the string `system` will not match `т m` (Cyrillic `т`, `м`). The visual rendering is identical in standard fonts. A zero-width space inserted between characters (`system` with U+200B) defeats exact-match filters while the model reads through the invisible character without interruption. NFKC-normalized fullwidth ASCII (rendering as `system`) bypasses filters tuned to the ASCII range.

**Agent tool calls amplify impact.** When a model generates a function call based on poisoned context—a RAG document containing invisible instructions, a user message with homoglyph-disguised parameters—the downstream system executes the tool call with no awareness that the input was manipulated. Sanitizing text before it enters the prompt pipeline eliminates the vector regardless of what the model does with it.

The defensive posture for agent developers is straightforward: sanitize every string that crosses a trust boundary before it reaches the model. One function call at the edge, and the invisible attack surface disappears.

```
from navi_sanitize import clean

# Before any string enters a prompt, RAG context, or tool call
sanitized = clean(user_input)
```

For structured data—JSON from APIs, nested dicts from RAG retrievals—`walk()` applies the same pipeline to every string in an arbitrary data structure:

```
from navi_sanitize import walk

# Sanitize every string in a RAG retrieval result
safe_context = walk(raw_retrieval_result)
```

## 3 Design Philosophy

Two design choices define navi-sanitize and separate it from every other tool in this space.

### 3.1 Transform, don't detect

`clean()` is a pure function. Same input, same output, every time. No ML models, no confidence scores, no thresholds to tune. The pipeline either modifies a character or it doesn't, and when it does, it logs a structured warning with a count. This makes the library auditable, testable, and predictable in production—properties that matter when the sanitization layer sits between untrusted

input and an autonomous agent.

The only library that previously addressed homoglyph confusion—`confusable_homoglyphs`—takes the opposite approach: a detection API that tells the caller *which* characters are confusable and from *which* scripts. The caller then decides what to do. This is valuable for analysis and auditing. It is not useful at 3 AM when an agent is processing untrusted input in production. `navi-sanitize` takes the opinionated position: confusable characters are replaced, invisible characters are stripped, and the pipeline always produces clean output. The application never handles “this input might be dangerous.” It is already fixed.

### 3.2 Curation over coverage

The Unicode Consortium maintains a `Confusables.txt` dataset with thousands of visually similar character pairs across dozens of scripts. `navi-sanitize` does not use it. Instead, the library maintains a curated map of 54 homoglyph pairs—the Cyrillic, Greek, Armenian, Cherokee, and typographic lookalikes that are actively weaponized in phishing, filter bypass, and injection attacks—plus 411 invisible characters across six categories (zero-width, format/control, variation selectors, Tag block, variation selector supplement, and bidirectional controls).

The curation is the feature. Every entry in the map is auditable by a human reading the source. Every entry has a documented security rationale. The map produces zero false positives on legitimate Unicode text—including CJK, Arabic, Hebrew, and emoji—and covers the attack surface that matters in practice. Exhaustive coverage of all Unicode confusables would flag thousands of benign character pairs and require per-deployment tuning—exactly the operational burden this library exists to eliminate.

## 4 Pipeline Design

Every string passed to `clean()` flows through six stages in strict order (Figure 1). Each stage is a pure function returning a cleaned string and a change indicator. The orchestrator logs warnings; stages have no side effects.

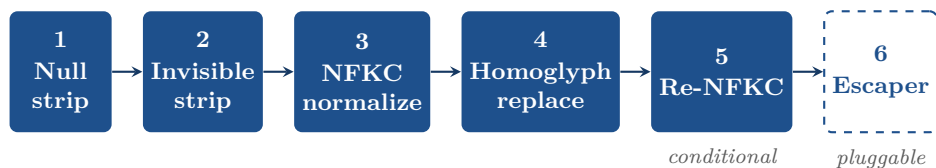


Figure 1: The six-stage sanitization pipeline. Stages 1–4 are universal; Stage 5 is conditional; Stage 6 is a caller-supplied escaper. Stage 5 runs only when Stage 4 replaces homoglyphs.

### 4.1 Why order is load-bearing

The stage ordering is not a convenience—it is a security property. Reordering stages opens attack vectors that the current sequence closes.

**Null bytes before invisibles.** Null bytes can appear between characters that, once the null is removed, form an invisible character sequence. Stripping nulls first ensures Stage 2 sees the actual content.

**Invisibles before NFKC.** A zero-width character inserted between two fullwidth braces would, if NFKC ran first, normalize the braces to ASCII `{` with the zero-width character still between

Stage	Operation	What It Does
1	Null byte strip	Remove <code>\x00</code> (prevents C-extension truncation)
2	Invisible strip	Remove 411 chars across six categories including zero-width, format/control, tag block, bidi, and variation selectors
3	NFKC normalize	Collapse fullwidth ASCII, ligatures, math symbols to standard forms
4	Homoglyph replace	Swap 54 Cyrillic/Greek/Armenian/Cherokee/typographic lookalikes to Latin
5	Re-NFKC	Re-normalize if Stage 4 changed anything (ensures idempotency)
6	Escaper	Caller-supplied <code>str</code> $\rightarrow$ <code>str</code> function for context-specific escaping

Table 1: Pipeline stages in execution order.

them. Stripping invisibles first removes the zero-width character, then NFKC normalizes the braces cleanly. More importantly, some invisible characters are themselves compatibility forms that NFKC would normalize rather than remove; stripping first is the more aggressive and correct choice.

**NFKC before homoglyphs.** NFKC normalization can produce characters that are homoglyph targets. Mathematical italic **a** (U+1D44E) normalizes to standard Latin **a** under NFKC—which is fine—but some mathematical symbols normalize to Greek letters that then require homoglyph replacement. Running NFKC first ensures Stage 4 sees canonical forms.

**Re-NFKC after homoglyphs.** Homoglyph replacement can produce Latin characters adjacent to combining marks that NFKC would compose. Greek **Υ** (U+03A5) followed by a combining tilde: Stage 4 replaces **Υ** with Latin **Y**, leaving **Y** + combining tilde, which NFKC composes to a precomposed form. Without this re-normalization pass, `clean(clean(x))` could differ from `clean(x)`, breaking the idempotency invariant. Stage 5 only runs when Stage 4 actually replaced homoglyphs, adding zero cost for clean text.

**Escaper last, output not re-sanitized.** The escaper’s output is context-specific and must not be altered by earlier stages. A Jinja2 escaper that produces `\{\}` would have its backslashes stripped or re-processed if the pipeline ran again. The escaper is a terminal transformation by design.

## 4.2 Compound attack resolution

The sequential architecture means compound attacks are disarmed layer by layer, without requiring any stage to understand the others. Consider the input `{ [U+200B] { c[U+043E]nfig } [U+200B] }` passed through `clean()` with the Jinja2 escaper:

1. Stage 1: No null bytes. Pass through.
2. Stage 2: Strip zero-width spaces. String becomes `{{ c[U+043E]nfig }}`.
3. Stage 3: No fullwidth forms. Pass through.
4. Stage 4: Replace Cyrillic **o** (U+043E, visually indistinguishable from Latin **o** in most fonts) with Latin **o**. String becomes `{{ config }}`.
5. Stage 5: No combining marks affected. Pass through.
6. Stage 6: Jinja2 escaper backslash-escapes `{{` and `}}`. Final output is safe.

No single stage would have caught this input. The invisible characters masked the Jinja2 delimiters from the escaper. The homoglyph masked the payload from keyword filters. The pipeline’s sequential design resolves both without any stage needing awareness of the other’s threat model.

### 4.3 Pluggable escapers

The first five stages are universal—they apply regardless of where the sanitized text will be used. The sixth stage is context-specific: a plain function matching `Callable[[str], str]`. No classes, no configuration objects, no state.

navi-sanitize ships two built-in escapers: `jinja2_escaper` (backslash-escapes template delimiters including triple-brace edge cases) and `path_escaper` (strips directory traversal sequences with backslash normalization). No LLM prompt escaper is included. Prompt delimiter syntax varies by vendor, changes between model versions, and is not standardizable at the library level. The pluggable design makes it trivial to write one:

```
def my_prompt_escaper(text: str) -> str:
    # Your vendor-specific logic here
    return text.replace("<|system|>", "").replace("<|user|>", "")

clean(user_input, escaper=my_prompt_escaper)
```

## 5 Verification

The pipeline's correctness rests on a single invariant: **idempotency**. For all inputs and all escaper configurations, `clean(clean(x))` must equal `clean(x)`. This is verified across every hostile input in the test suite and enforced as a fuzz invariant via Atheris.

The test suite comprises 300+ cases across six categories:

- **Unit tests** for each stage in isolation and `clean()/walk()` end-to-end.
- **Adversarial tests** ported from the production predecessor (navi-bootstrap), with a **test oracle**: for every shared input, `clean(text, escaper=jinja2_escaper)` must produce identical output to the predecessor's `_sanitize_string(text, escape_jinja=True)`.
- **Bypass attempts** written from the attacker's perspective—a passing test means the attack was blocked. Covers homoglyph map gaps, invisible character gaps, escaper bypasses, and stage-order exploits.
- **Audit remediation** regression tests for 18 bypass vectors found during penetration testing, including supplementary variation selectors (VS17–VS256), the U+E0000 language tag, Greek lowercase homoglyphs, and path escaper backslash normalization.
- **Garbage tests** for size extremes (1 MB strings, 100-level nesting, 10K-key dicts), Unicode edge cases (max codepoint, combining characters, CJK/Arabic/emoji passthrough), and escaper abuse.
- **Fuzz harness** using Atheris, enforcing four invariants on the universal pipeline (without escaper): never raises on valid input, always returns `str`, output contains no null bytes or invisible characters, and idempotent. (When an escaper is active, the harness verifies the first two invariants; the escaper may legitimately re-introduce characters the pipeline would otherwise strip.)

CI runs Semgrep SAST, CodeQL analysis (Python + Actions), pip-audit dependency scanning, and OpenSSF Scorecard on every push. All GitHub Actions are SHA-pinned.

## 6 Comparison with Existing Tools

The relevant comparison is not feature-by-feature—it is threat-model-by-threat-model.

**Unidecode and anyascii** transliterate *all* Unicode to ASCII. They convert Chinese characters to pinyin, Korean to romanization, and Arabic to Latin approximations. This solves the homoglyph problem by destroying all non-Latin content. For applications that process multilingual text—which is most AI agent deployments—this is unacceptable. navi-sanitize replaces 54 targeted lookalikes and leaves legitimate Unicode intact.

**confusable\_homoglyphs** uses the Unicode Consortium’s full confusables dataset and provides a detection API: it returns which characters are confusable and from which scripts. It does not replace or remove anything. The library is also archived upstream. navi-sanitize’s curated map trades coverage for precision and ships the replacement, not just the signal.

**ftfy** repairs encoding corruption—mojibake, misinterpreted byte sequences, C1 control characters. It is excellent at what it does, and it is complementary, not competing. The critical difference: ftfy explicitly *preserves* bidirectional overrides, zero-width characters, and variation selectors. Its design philosophy is “don’t remove characters that might be intentional.” navi-sanitize’s design philosophy is “remove characters that can be weaponized.” Run ftfy first to fix encoding damage, then navi-sanitize to strip evasion vectors.

**MarkupSafe and nh3** operate at the HTML structure layer—escaping `<`, `>`, `&` and sanitizing tags and attributes. navi-sanitize operates at the character content layer—normalizing the text *inside* those HTML elements. They address orthogonal attack surfaces and compose naturally: `escape(clean(user_input))`.

**No existing tool** combines invisible character stripping, homoglyph replacement, NFKC normalization, and pluggable escaping in a single zero-dependency pipeline. navi-sanitize occupies the gap between encoding repair (ftfy), HTML structure (MarkupSafe), schema validation (pydantic), and full transliteration (Unidecode)—the character-level security layer that none of them provide.

## 7 Performance

Benchmarks measured on Python 3.12, single thread, via `pytest-benchmark`.

Scenario	Mean	Ops/sec
<code>clean()</code> — short, clean text (no-op)	2.8 $\mu$ s	358K
<code>clean()</code> — short, hostile (all stages fire)	67 $\mu$ s	15K
<code>clean()</code> — 13 KB clean text	810 $\mu$ s	1.2K
<code>clean()</code> — 10 KB hostile text	449 $\mu$ s	2.2K
<code>clean()</code> — 100 KB hostile payload	5.7 ms	176
<code>walk()</code> — 100-item nested dict, clean	537 $\mu$ s	1.9K
<code>walk()</code> — 100-item nested dict, hostile	6.9 ms	144

Table 2: Pipeline performance across input profiles.

The number that matters for agent developers: 2.8  $\mu$ s on the clean path. This is negligible relative to any LLM API call, database query, or network round-trip in the pipeline. Even the worst-case short hostile string at 67  $\mu$ s is three orders of magnitude faster than a model inference. The cost of *not* sanitizing—a successful prompt injection, a poisoned tool call, a homoglyph-bypassed guardrail—is incomparably higher.

All stages are  $O(n)$  in string length. The regex for invisible characters is compiled once at import time. The homoglyph map is a dictionary lookup. `walk()` adds `deepcopy` overhead to guarantee the original data structure is never mutated; for hot paths where a copy already exists, calling `clean()` on individual strings avoids this cost.

## 8 Limitations

These are design choices, not gaps.

**Bidi control stripping breaks legitimate RTL rendering.** Stripping U+202A–U+202E and U+2066–U+2069 affects Arabic and Hebrew text that uses explicit directional controls for mixed-direction layout. navi-sanitize prioritizes security over RTL display fidelity. Applications processing known-legitimate RTL text should apply the pipeline selectively.

**The homoglyph map is intentionally not exhaustive.** The Unicode confusables dataset contains thousands of pairs. navi-sanitize’s 54-pair map covers the highest-risk Latin lookalikes. Characters from scripts not in the map (e.g., Georgian, Tibetan) pass through unchanged. This is a deliberate trade-off: zero false positives on legitimate Unicode text at the cost of incomplete coverage of rare confusables.

**NFKC normalization is lossy.** Compatibility decomposition changes some characters irreversibly: superscripts become regular digits, ligatures expand, circled numbers lose their circles. This is correct for security sanitization but may not be appropriate for all text processing contexts. Applications that need to preserve these forms should not use navi-sanitize on that data.

## 9 Conclusion

Every defense in the stack—HTML escaping, schema validation, keyword filtering, ML-based guardrails—assumes the characters are what they appear to be. navi-sanitize makes that assumption true.

One function call at the input boundary. Deterministic. Microseconds. No false positives. What your application receives is what is actually there.

```
pip install navi-sanitize
```

---

navi-sanitize is open source under the MIT license.

<https://github.com/Project-Navi/navi-sanitize>

Python 3.12+ • Zero dependencies • `pip install navi-sanitize`