Research Findings: Detecting Superseded Pending Changes

Executive Summary

This document presents research findings on methods to detect when text additions from pending revisions have been removed, modified, or superseded in the current article version. Three approaches were investigated:

1. **Token/Word-Level Diff Tracking** - WikiTrust-inspired text persistence analysis
2. **Semantic Similarity Text Comparison** - Using fuzzy matching and diff libraries
3. **LLM-Assisted Detection** - Using open-source language models for semantic change detection

Based on the research, **Approach B (Semantic Similarity)** is recommended for immediate implementation, with **Approach A (Token-Level Tracking)** as a future enhancement for high-accuracy scenarios.

Current System Analysis

Data Model

The PendingChangesBot-ng system currently stores:

- **PendingRevision**: Contains wikitext, parentid, revid, timestamp, etc.
- **PendingPage**: Contains stable_revid, title, categories
- Revisions are fetched via Superset queries and cached locally

Existing Autoreview Logic

Located in app/reviews/autoreview.py, the system already implements:

- Bot user auto-approval
- Auto-approved group checks
- Redirect conversion detection
- Blocking category checks
- New render error detection
- Reference-only edit detection

Key Observation

The reference-only edit logic (_is_reference_only_edit()) demonstrates the system can:

1. Compare old vs. new wikitext.
2. Parse wikitext to selectively ignore certain changes (e.g., reference tags).

This provides a solid foundation for implementing the new superseded change detection.

Proposed Approaches

Approach A: Token/Word-Level Diff Tracking

This method is inspired by WikiTrust and involves tracking the origin and persistence of each word or token across revisions.

How It Works:

1. **Initial Annotation**: When a pending revision is created, its added text is tokenized (split into words). Each token is annotated with the revid it originated from.
2. **Diff Tracking**: To check if the text still exists in the current stable revision, we perform a word-level diff between the pending revision's wikitext and the current wikitext.
3. **Persistence Check**: We then check how many of the originally added tokens from the pending revision are still present in the current version. If the percentage of surviving tokens falls below a threshold (e.g., 10%), the change is considered superseded.

Python Libraries:

- difflib (standard library)
- diff_match_patch (by Google)

Pros:

- ✅ High accuracy, especially for literal text matches.
- ✅ Robust against minor reformatting.
- ✅ Conceptually proven by WikiTrust.

Cons:

- ❌ Does not understand semantics (rephrasing is seen as a removal).
- ❌ Requires storing token-level metadata, which could increase database complexity if we decide to track persistence across *all* revisions (not required for this specific task).
- ❌ Can be computationally intensive for very large articles.

Approach B: Semantic Similarity (Recommended for MVP)

This approach uses string similarity algorithms to determine if the *essence* of an addition is still present, without needing to match the exact wording.

How It Works:

1. **Extract Additions**: Use a standard diff to get all text blocks that were added in the pending revision.
2. **Compare with Current Text**: For each added block, calculate its similarity score against the entire text of the current stable revision. A simple and effective way is to find the best matching substring in the current text.
3. **Threshold Check**: If the similarity score for *all* added blocks is very low (e.g., below a 0.2 ratio using SequenceMatcher), it implies the added content has been effectively removed or rewritten beyond recognition.

Python Libraries:

- difflib.SequenceMatcher (standard library, excellent for this task)
- RapidFuzz (a faster, more advanced alternative)

Pros:

- ✅ Very simple to implement with standard libraries.
- ✅ Extremely fast and low on resources.
- ✅ Tolerant to rephrasing and minor edits within the added text.
- ✅ No need for extra data storage.

Cons:

- ❌ Can be fooled by common phrases or words, leading to potential false positives (mitigated by checking against the parent revision text as a baseline).
- ❌ Less precise than token-level tracking for exact wording.

Approach C: LLM-Assisted Detection

This method leverages a local, open-source Large Language Model (LLM) to perform a semantic analysis of the changes.

How It Works:

1. **Prepare Input**: Provide the LLM with three pieces of text: the parent revision wikitext, the pending revision wikitext, and the current stable revision wikitext.
2. **Formulate Prompt**: Ask the LLM a direct question, such as: *"Analyze the changes between the parent and pending revision. Then, determine if the substance of those additions still exists in the current revision. Respond with only 'YES' or 'NO'."*
3. **Evaluate Response**: The LLM's response directly determines if the change is superseded.

Python Libraries & Tools:

- ollama-python: A client for running local LLMs like Llama 3, Mistral, or Phi-3.
- An Ollama server instance running a small, quantized model (e.g., phi3:mini, qwen:4b).

Pros:

- ✅ Highest potential accuracy, as it understands context, rephrasing, and semantics.
- ✅ Can correctly identify cases where text is moved or integrated into a new paragraph.
- ✅ Simple API call (once the infrastructure is set up).

Cons:

- ❌ Requires dedicated hardware (a GPU with 8GB+ VRAM is recommended for good performance).
- ❌ Slower than other methods (0.5-2 seconds per check, depending on hardware).
- ❌ Introduces a significant new dependency (Ollama server).
- ❌ Prompt engineering may be required to get consistently reliable results.

Comparative Analysis

Feature	Approach A (Token-Level)	Approach B (Similarity)	Approach C (LLM-Assisted)
Accuracy	High (for literal text)	Medium-High	Very High (semantic)
Complexity	Medium	**Low**	High (infrastructure)
Performance	Fast	**Very Fast**	Slow
Dependencies	None / diff-match-patch	None	Ollama, GPU
Handles Rephrasing	No	Partially	**Yes**
Recommendation	Future Enhancement	**MVP Implementation**	For specific edge cases

Test Scenarios

Here is an analysis of how each approach would handle different real-world editing scenarios.

- **Pending Change**: User A adds the sentence: *"The city was founded in 1886 during the gold rush."*

Scenario 1: Clean Removal

- **Action**: User B later removes that exact sentence.
- **Approach A**: ✅ **Detects**: The original tokens are gone.
- **Approach B**: ✅ **Detects**: Similarity score drops to near-zero.
- **Approach C**: ✅ **Detects**: LLM understands the removal.

Scenario 2: Addition Incorporated with Rephrasing

- **Action**: User B rewrites the sentence to: *"Founded in 1886, the city's origins trace back to the gold rush."*
- **Approach A**: ❌ **Fails**: Sees the original tokens as removed and new ones added.
- **Approach B**: ✅ **Detects**: Similarity score remains high (>0.8), so it correctly identifies the text as still present.
- **Approach C**: ✅ **Detects**: LLM understands the sentences are semantically equivalent.

Scenario 3: Addition Moved to a Different Section

- **Action**: User B moves the sentence from the "History" section to the "Economy" section.
- **Approach A**: ✅ **Detects**: The original tokens are still present in the document.
- **Approach B**: ✅ **Detects**: The sentence is found elsewhere, resulting in a high similarity score.
- **Approach C**: ✅ **Detects**: LLM confirms the content still exists.

Scenario 4: Partial Removal / Vandalism

- **Action**: User B vandalizes the sentence to: *"The city was founded in 1886 during the mold rush."*
- **Approach A**: ❌ **Fails**: Most tokens are still present, so it considers the addition intact.
- **Approach B**: ✅ **Detects**: Similarity score is very high, correctly identifying most of the text is still present.
- **Approach C**: ✅ **Detects**: LLM can understand the semantic shift and might correctly identify it as a modification.

Implementation Proposal

A hybrid approach is recommended, starting with the simplest effective method.

Phase 1: MVP with Semantic Similarity (Approach B)

1. **Create a new detector function** in autoreview.py called _is_addition_superseded().
2. **Logic**:
   - Get the diff between the pending revision's parent and the pending revision itself to identify all added text blocks.
   - If no text was added (only removals/moves), return False.
   - For each added block, clean it (e.g., remove templates, convert to plain text).
   - Use difflib.SequenceMatcher to find the best match for this block within the current stable revision's plain text.
   - If the top similarity score for *all* added blocks is below a configurable threshold (e.g., 0.2), return True.
3. **Integrate**: Call this new function from the main autoreview logic. If it returns True, the pending change is skipped with a log message: "Skipping review: All additions have been superseded or removed."

Phase 2: Enhancement with Token-Level Tracking (Approach A)

- If the MVP proves insufficient for certain edge cases, implement Approach A as a more precise secondary check.
- The _is_addition_superseded() function could first run the fast similarity check. If the result is ambiguous, it could then run the more computationally intensive token-level check for a final decision.

Future Consideration: LLM-Assisted Review

- If the project acquires GPU resources, Approach C could be implemented for a "high confidence" review tier or to handle complex cases that the other two methods fail to resolve correctly.

Risks and Mitigations

1. **Risk**: False positives (incorrectly skipping a change that should be reviewed).
   - **Mitigation**: Start with a very conservative threshold for similarity. Log all automated skips for manual audit during the initial deployment phase to fine-tune the threshold.
2. **Risk**: Performance impact on the review queue.
   - **Mitigation**: Approach B is extremely lightweight. Benchmarking should be done, but the impact is expected to be negligible compared to existing checks.
3. **Risk**: Inability to handle complex wikitext (e.g., nested templates).
   - **Mitigation**: Utilize the existing wikitext parsing libraries to convert wikitext to plain text before comparison. This is a proven method already used in the reference-only edit detector.

Conclusion & Recommendation

The investigation confirms that detecting superseded pending changes is technically feasible with modern Python libraries.

- **Recommendation**: Implement **Approach B (Semantic Similarity)** as the Minimum Viable Product (MVP). It provides the best balance of simplicity, performance, and accuracy for the most common use cases.
- **Roadmap**:
  1. **Develop and deploy the MVP (Phase 1).**
  2. **Monitor and analyze** its performance and accuracy over a one-month period.
  3. Based on the analysis, decide if **Phase 2 (Token-Level Tracking)** is necessary to handle edge cases.
  4. Re-evaluate the need for **LLM-assisted detection** once hardware and infrastructure plans are more concrete.