# Project Information

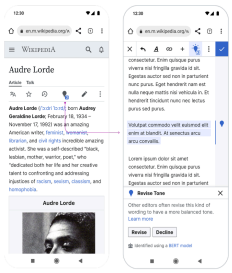
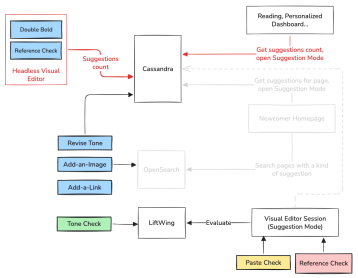
# Prep Pantry: Intake for Server Side Edit Suggestions

---

“Surfacing Edit Suggestions outside of VE” is a priority for FY2026-2027. The project will proceed in multiple phases, of which Phase 1 and Phase 2 is planned.

**Phase 1** aims to add an “edit in VE suggestion mode” button to the MediaWiki mobile web UI when a logged in reader is viewing a page. This button should display the count of available edit suggestions for that page.

**Phase 2** aims to allow showing the edit suggestions relevant to a specific span of content in UIs other than when editing using the visual editor. E.g. while reading a page, or in a list or feed of edit suggestions on a user’s Personalized Dashboard.

| **Product** | |
|---|---|
| What is the business case for this work? Is there a PRD or user stories we can reference? | Requirements/Server-Side Suggestions<br>Other doc: WE1.7 Edit Suggestions KR |
| What does the user experience look like? Are there wireframes we can reference? |    |
| **APP & Project Management** | |
| What KR is associated with this project? | DE 1.1 |
| What is your timeline? Are there earlier and later deliverables? | The user experience pictured above, and described here, needs to [ideally] be delivered by mid-way through Q1, at the latest. This way, we can conclude a controlled experiment before the quarter ends. |
| What is the priority of this work for the listed KR? What | High |

| | |
|---|---|
| happens if some or all fields are not delivered? | |
| **Governance** | |
| What Engineering team(s) are available as (Contingency) Development Owners for business logic? | Editing |
| Who owns the business requirements and priorities for this work (that is, the Product Management Owner)? | Peter Pelberg + Guilherme Gonçalves |
| **Prioritization** | |
| *List the required data fields here in descending priority order:*<br><br>1. *Most important data field*<br>2. *Second most important data field*<br>3. … | |

## Edit Suggestion compute triggers

What can cause a precomputed and stored edit suggestion to no longer be valid? What should cause a recompute?

| Trigger | Associated user actions | What gets invalidated | Phase 1 behavior | Desired behavior | Impact<br>*What happens if suggestions are NOT recomputed?* | Product priority | Difficulty | Notes |
|---|---|---|---|---|---|---|---|---|
| New revision of a page. | Edits, reverts, category changes, article creations etc. | All of the suggestions associated with the page, because they're linked to the previous revision. | Suggestion counts for the new revision are computed within ?(1? 5?)min, for 9X% of revisions. | | High | High | Low | |
| Change to a template used by a page. | Edits to a template included by other pages. | Any suggestions associated with HTML contents rendered by templates. | Nothing happens. Headless VE is not invoked, and even if it was, EditCheck currently ignores HTML rendered by templates. | | Low; VE does not read/depend on the contents of a template | Low | Medium | Not required for Phase 1; if/when we come to learn many false positives we can consider adding support. |

| Trigger | Associated user actions | What gets invalidated | Phase 1 behavior | Desired behavior | Impact<br>*What happens if suggestions are NOT recomputed?* | Product priority | Difficulty | Notes |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | *Potential fix:* we could re-evaluate suggestions whenever the page is *re-rendered* e.g. by a used template changing. |
| Change to Edit Check on-wiki config (e.g. TextMatch). | User edits a edit suggestion config page. | Potentially all suggestions for that wiki. | Suggestions gradually get recomputed with new logic as new revisions are created.<br><br>There is no option to recompute old revisions with new code/config. | | Medium | Medium | TBD | TBD: as part of the initial technical pilot needed to evaluate headless ve resource requirements, we will learn how frequently we can afford to clear the caches which lets us know which approaches we can take to solve this problem and accordingly, the difficulty involved. See T432288 . |
| Code or config changes within Headless VE, including `VisualEditorEditCheckDefaultConfig`. | None, this is done by Editing. | Potentially all suggestions on all wikis. | Suggestions gradually get recomputed with new logic as new revisions are created.<br><br>There is no option to recompute old revisions with new code/config. | | Medium; if unaddressed suggestions could be shown where they are not desired by volunteers. *E.g. if we changed in-house rules for suppressing Tone Check on quoted content* | Low | High | If/when we come to learn we need to invalidate the cache, we can implement the functionality needed to do so manually. |
| Change to external source of suggestions (e.g. image suggestions, link suggestions, revise | Upstream suggestion recompute (batch? User triggered?) | Suggestions associated with the page revision that the external source | Nothing happens. If external suggestions for a page revision are | | Medium; if unaddressed suggestions could be shown where | | Medium | |

| d user ns | What gets invalidated | Phase 1 behavior | Desired behavior | Impact<br>*What happens if suggestions are NOT recomputed?* | Product priority | |
|---|---|---|---|---|---|---|
| | used. | not available at the Headless VE invocation time, they are not stored. | | they are not desired by volunteers. *E.g. if we changed in-house rules for suppressing Tone Check on quoted content* | | |

# Notes

- Principles: undercounting is better than over-counting
- Templates edge case:
  - requiredTemplateParams:
    - A user edits TemplateData to downgrade a param from required to non-required. Now all suggestions of "this required param is missing" are incorrect. This sort of thing is likely quite rare.
  - inCategory filters:
    - Checks can filter for categories that are included via templates - these are rarely changed, but could in theory
  - Fix: We could re-evaluate suggestions whenever the page is *re-rendered* e.g. by a used template changing.
- EditCheck config:

# Suggestion Count

## Edit Suggestion Count

This is the Phase 1 requirement, aimed to be completed in Q1 FY2026-2027.

| Semantics | |
|---|---|
| In simple terms, what does this data represent or measure? | The count of available edit suggestions on the latest revision of an article. |
| In simple terms, how would you describe the data being created? | Edit Suggestions can be created by varying sources. Most are currently tightly coupled with VisualEditor (e.g. TextMatch), others are handled on demand by LiftWing (articlequality), and LiftWing precomputed in a stream (Revise Tone). Others must be precomputed offline batch (LLM based, Image Suggestions, Add-a-link). Some may even be supplied by users via configuration or gadgets.<br><br>The count of edit suggestions will be used to show the user how many suggestions are available when they transition to VE Suggestion Mode. So, as long as VE knows about a potential suggestion, it can count it. |
| **Inputs** | |
| What primary data sources (e.g. MediaWiki tables or instrumented events) do we need to transform? | Mostly Parsoid HTML, but VE based edit suggestions have a lot of other inputs, including user managed on wiki config.<br><br>In the future, we will need more inputs, including the ability to store information about invalidated suggestions. |
| Will we be ingesting, transforming or serving Personal Information? | No. |
| How often does the source data change? | After every edit. Possible other change triggers (config changes, template/templatedata edits) too. |
| **Processing** | |
| What transformations (e.g. filtering, aggregations) do we need to perform? | Transformation: Revision (HTML) + more -> Edit Suggestions.<br><br>Filtering: Some minor filtering on kind of page (wikitext/html, article mainspace, etc).<br><br>Aggregations: This dataset is about the count of available edit suggestions. So we will need to sum them. *(Note: the next dataset* |

| | |
|---|---|
| | *is about storing and using the actual suggestions.)* |
| **Output** | |
| Are we producing structured data (like key-value pairs, JSON even if serialized), or opaque data (like images)? | Structured data. |
| If structured, what is the data model? What types, constraints and relationships exist? | For each page, the count of each available edit suggestion (or issue) type on that page. E.g.<br><br>- Revise_tone: 3<br>- british_english: 2<br>- Reference_needed: 5<br>- Non_npov: 2 |
| How much staleness is acceptable? For example, can the output be a week old relative to the inputs? What happens if this expectation is not met? | At a minimum, it is important that we be able to accurately indicate the *presence* (yes/no) of suggestions within the article (or perhaps, section) a logged-in person is reading, to the latest revision.<br><br>*Note: emphasis on "presence" above to indicate an openness to presenting a binary indication within the UI for this MVP if generating an exact count proves meaningfully complex.*<br><br>Rationale: if we surface an inaccurate count of suggestions to logged-in readers/editors, we risk:<br>1. People becoming confused when they don't see suggestions upon tapping edit, when the same action surfaced suggestions in a previous attempt(s)<br>2. People never discovering suggestions that are available within the article(s) they're reading<br>3. More broadly, people block out the UI we're introducing because they've found it unreliable, therefore not offering meaningful information, and ultimately, something they stop/don't start engaging with. |
| What completeness is acceptable? Can the outputs be computed on samples of the inputs? What happens if this expectation is not met? | The output (read: suggestion count) needs to be computed on an article's latest revisionID.<br><br>If this requirement is not met, we risk:<br>1. *See cell above.* |
| What kind of retention is needed for this data? | For the product feature: none. Only the latest page revision is needed. |
| What amount of historical output data is needed? Do we need to transform the input data from the beginning of time, or only from a certain point in time? | For the product feature: none. Only the latest page revision is needed. |

| Serving | |
|---|---|
| What expectations do we have about the interfaces or APIs for accessing this data? Who is the client? | Mobile Web logged-in readers on for now. Scope may expand to include temp accounts and logged-out users in the future.<br><br>This data will be requested on every page load for logged-in users on mobile web, which is roughly 4k requests/second. |
| What kinds of read patterns are expected? Are they sequential or random lookups? Do we need search semantics? | Not for Phase 1, which only needs this count. In the future, we will need to discover pages with edit suggestions, and serve the actual suggestion details. |
| What kinds of writing patterns are expected? Are writes frequent or rare? Individual or in bulk? | For each edit, each possible edit suggestion type will be checked for counts. |
| Are reads or writes geographically concentrated? | |
| **Operations** | |
| Roughly, how much data are we producing? How do we expect it to grow over time? | For Phase 1 proof of concept, we expect there to be ≤ 100 edit suggestion types available per wiki, and in turn, computed per article. |
| Do we expect a steady stream of input, or bursts? What could cause a burst? | Steady: new revisions (edits) are the input. |
| What amount of downtime is acceptable (for transforming data *and* for serving it)? What happens if this expectation is not met? | If data is not available, the service should gracefully downgrade and not show the feature. |
| What kind of read latency is acceptable? For example, can a lookup via an API take 200ms? What happens if this expectation is not met? | The suggestion count will be shown in the UI on every logged-in mobile web page load. It should be shown to the user as soon as possible so they don't scroll past it in the edit UI.<br><br>This could be either fetched server side during page load, or client side. Either way, the client's fetch will timeout after <TODO> ms.<br><br>Since this will be fetched on page load, the latency must be very low. Ideally, 10-50ms. |
| How should operational downtimes be handled? | |

# Design - Edit Suggestions Count per Page - v0.1

*Assisted by: Claude Sonnet 4.6*

Status:

- 2026-06-09: initial drafting
- 2026-06-17: initial draft ready for internal review
- 2026-06-22: initial recommendation: LAC + MW JobQueue
- 2026-06-24: draft v0.1 ready for prelim stakeholder review and comments.

# Background

Edit Suggestions can be produced by a variety of systems. Some are tightly coupled with VisualEditor (VE) (e.g. TextMatch), some are precomputed via a stream by LiftWing (e.g. Revise Tone), and others (may?) require offline batch processing (e.g. LLM-based suggestions, Image Suggestions, Add-a-link). Parsoid HTML is the primary input for most, though VE based suggestions have additional inputs including user-managed on-wiki config.

This design introduces shared infrastructure for precomputing and serving edit suggestions. This specific data product design is for Phase 1: storing per-article edit suggestion counts keyed to the latest page revision. Phase 2 intends to store and serve full suggestion details (e.g. title, description, actions, content span the suggestion is relevant to, etc.) and support finding pages with suggestions types.  A future phase aims to incorporate user feedback about suggestions.

Additionally, this design is operating within the broader FY2026-2027 ST6.2 KR: building common derived data platform support. Other use cases are also inputs to design decisions.

# Requirements

- Precompute and store the count of each edit suggestion type available in VE Suggestion Mode on the latest revision of every mainspace article.
- Counts should reflect the latest revision at time of read; stale counts risk user-facing inconsistencies that erode trust in the feature.
- The data must be available quickly on every page read for all logged-in mobile web users:
    - target read latency of ~10ms (Otto to verify this)
    - at roughly 4,000 requests/second
- Writes are triggered on every edit: < ~20 updates per second.
- On data unavailability, the system must gracefully degrade, possibly by suppressing the UI feature.

# Assumptions

- Only the latest revision needs to be stored and served.
- Invalidation of suggestions due to things other than new revisions is out of scope for Phase 1. See Phase 2 Prep Pantry document for more details.
    - E.g. TextMatch config changes or template rerenders.

    - E.g. users reject suggestions.
    - TODO <link here> Appendix has a table of all potential things that could cause computed edit suggestions for a page to be invalid.
- No PII is involved in any input, transformation, or output.
- Only suggestions that can be shown in VE Suggestion Mode (whether computed in VE or fetched by it from existing APIs) are in scope for Phase 1.
    - If VE Suggestion mode uses any offline batch computed suggestions (e.g. add-an-image, add-a-link) and the batch computed suggestion is not available for a revision at the time the full suggestion count is computed, it will not be included in the count.
- Edit suggestions for a page can be computed within several seconds in the worst case.
- The suggestions for a new page revision will not be immediately available after the edit is saved.

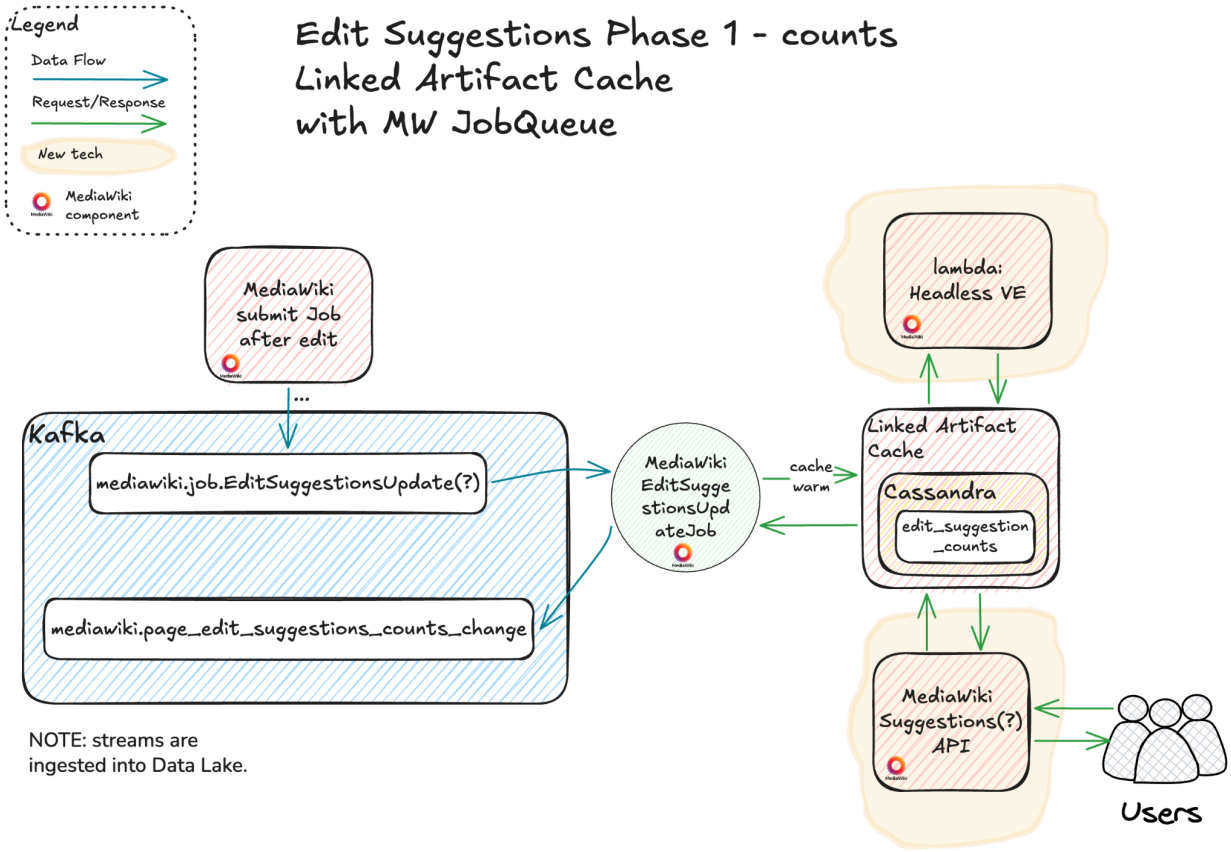
# Design

*Diagram Drafting here.*

## Recommendation: Linked Artifact Cache with MediaWiki JobQueue

This architecture has the most encouragement from the SRE Data Persistence team, as well as having the fewest new technologies to deploy and maintain. It also may have some future advantages for paving the path for future derived data use cases that can be computed on-demand by a lambda function.

## Diagram




## Compute: Headless VisualEditor

Edit suggestion computation will be handled by a headless instance of VisualEditor (VE). This will be the '[lambda function]' behind LAC.

VE is tightly coupled with [Suggestion Mode] and already contains all the logic and in-browser data model needed to compute edit suggestions, including support for [user-configured suggestions on individual wikis]. Extracting this logic into a standalone library would be prohibitively complex given VE's existing context dependencies.

Instead, we will deploy a headless browser that loads MediaWiki's VisualEditor JavaScript, wrapped in an gRPC API (if behind Linked Artifact Cache, see below).

See also Headless VisualEditor for EditCheck.

### Alternatives to Headless VE considered

**Calculate suggestions in the client**
For surfacing suggestions in read mode, we could download VE code and run it in the background, similar to how VisualDiffs are calculated.
Cons:

- Data not available anywhere else, e.g. a suggestion feed on Special:Homepage, so this only works for this initial product. See T416531 for other places we are considering surfacing suggestions.
- High bandwidth cost added to all readers (fetching code and Parsoid HTML)
- Potentially slow to compute (several seconds to fetch and evaluate)

**Run VE in Node**
While we believe it is possible to run at least the core visual editor in Node with JSDOM, for VE and EditChecks to run we also require:

- Additional code from extensions that register VE handlers (Cite, Math, etc.)
- MediaWiki config values
- MediaWiki API calls (for link metadata, TemplateData etc.)
- Potentially other ResourceLoader modules from across MediaWiki

Getting this working would be a multi-quarter undertaking with large maintenance costs.

**Duplicate EditCheck logic to run against Parsoid HTML**
*See T431602 for details.*
As with the above this would have a very large development and maintenance cost. The slight benefit is that it could be implemented selective/progressively on a per check bases (although ~80% of the cost will be setting up shared logic for all checks).

## Serving: MediaWiki API

Regardless of which architecture is chosen, a MediaWiki API layer will sit between the backend storage and the product feature. This layer abstracts the storage implementation from consumers, meaning the backend can evolve independently without requiring changes to the product.

It is not yet clear where the right abstraction level for this API layer is. Is this a ‘linked artifacts’ MediaWiki extension? Or is this specifically for Edit Suggestions? Or both? TBD.

## Storage: Linked Artifact Cache

Linked Artifact Cache (LAC) is a service for caching derived data artifacts linked to a MediaWiki entity (e.g. a page revision). On a cache miss, LAC invokes a compute function (AKA a 'lambda function') on demand (in this case Headless VE) to compute and store the result.

LAC stores artifacts in Cassandra as opaque blobs keyed to a MediaWiki entity, with schemas defined externally per artifact type. For edit suggestion counts, the artifact would be a JSON mapping of suggestion type name to count, e.g.

json

```json
{
  "revise_tone": 3,
  "british_english": 2,
  "reference_needed": 5
}
```

Note: some structured task suggestion types (e.g. Add Link, Add Image) may not be compatible with LAC's on-demand cache semantics. Batch computed suggestions cannot be computed on-demand. The time at which these kinds of suggestions are available is out of sync with the time at which on-demand suggestions are available. Batch computed suggestions are available at most within hours; on-demand suggestions are expected to be available within seconds.

For the purposes of this design we assume either that they are not relevant for Suggestion Mode, or that Headless VE's compute logic already accounts for them.

LAC does not (yet!) have a built-in precompute or cache warming mechanism; it only stores results when a value is requested. Since computing edit suggestions for a revision can take several seconds, we need a separate mechanism to warm the cache (precompute) after each edit, before a user requests the data. The recommended option is to use MediaWiki JobQueue.

After warming the cache, we can emit additional reusable side outputs, e.g. a `mediawiki.page_suggestion_counts_change` stream that will automatically be ingested to the Data Lake for historical analysis.

### Precompute: MediaWiki JobQueue

After each edit, a job is submitted to the MediaWiki JobQueue. This job will request edit suggestions from LAC for the new revision, warming the cache.

**Pros**

- Low barrier to entry and maintenance for MW teams.
- No new tech to deploy; only new Jobs to implement.
- Possibility to implement MW abstractions to pave the path for doing this in the future.

**Cons**

- A MW Job is a specific action tied to a specific event (an edit). This results in an RPC-style update rather than a Pub/Sub pattern, which is less flexible for future use cases. Every action taken results in a new job submitted.

If we use MW JobQueue, there may be opportunities to pave the path for more MediaWiki based derived data update features and use cases. E.g. a generalized Linked Artifact Cache update job that knows what caches need warmed? Integration with DomainEvents so others can respond to Linked Artifact updated events? TBD.

## Ownership

*Ownership Roles as defined in Service Catalog/Ownership Roles and Responsibilities - MediaWiki for **Active WMF Development**.*

| | **Compute**<br><br>Headless VE / Lambda | **Serving**<br><br>MW access to LAC service[1] | **Storage**<br><br>Linked Artifact Cache | **Precompute**<br><br>MediaWiki Job |
|---|---|---|---|---|
| **Development Owner** | MediaWiki Editing | MediaWiki Editing | SRE Data Persistence | Data Engineering |
| **Configuration & Deployment Owner** | MediaWiki Editing[2] | Release Engineering | SRE Data Persistence | Release Engineering |

| | | | | |
|---|---|---|---|---|
| **Incident Response Owner** | MediaWiki Editing | MediaWiki Editing | SRE Data Persistence | Data Engineering |
| **Product Management Owner** | MediaWiki Editing | MediaWiki Editing | MediaWiki Editing | Data Engineering |

[1] The MediaWiki access layer API is not yet well defined. In the near term, it will likely consist of a very simple LAC HTTP client wrapper. At this stage it may be owned by the Editing Team. As the Data Engineering Team onboards more linked artifact derived data use cases, they may develop and own this layer.

[2] Editing Team will own the deployment and ongoing operations. Service Ops can assist with operations, including initial service deployment in Kubernetes.

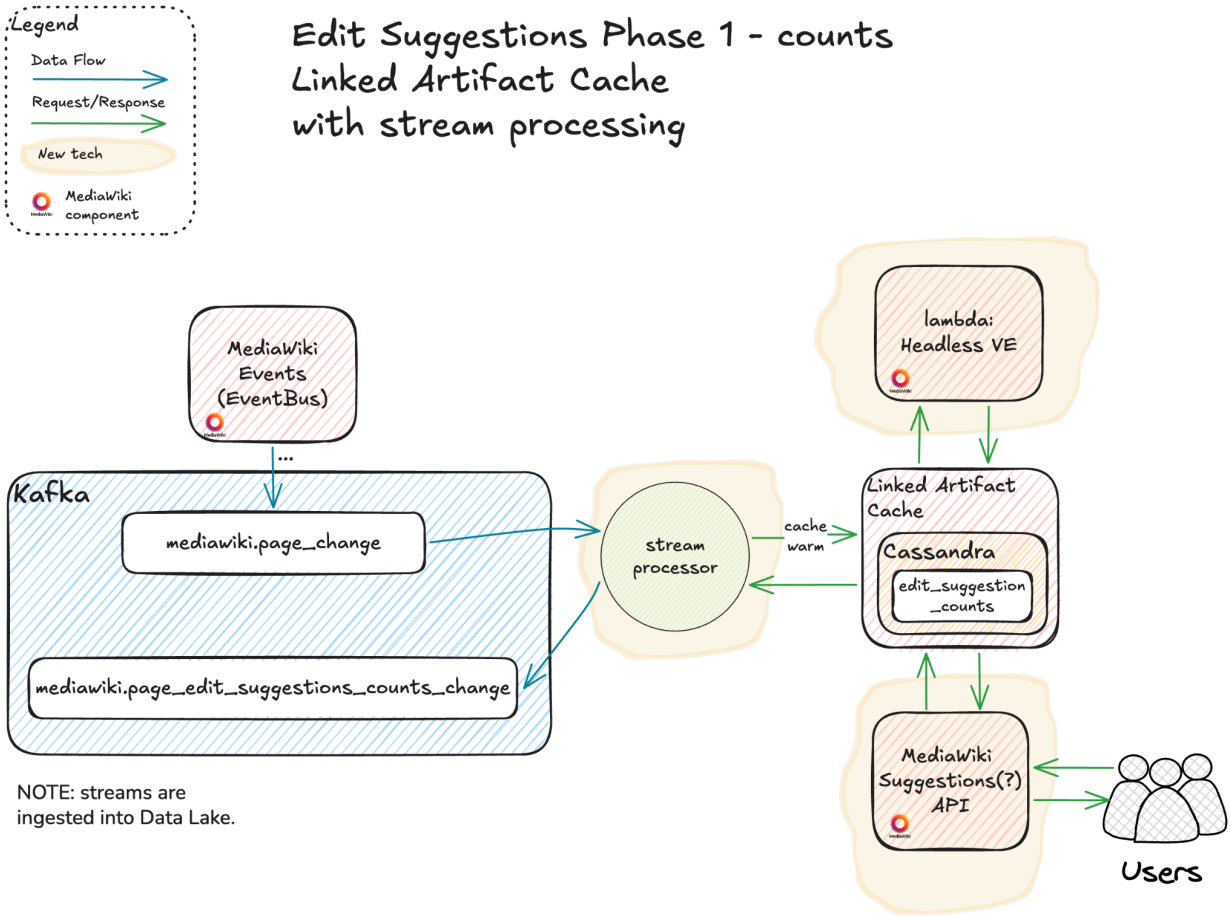
---

## Alternatives

While there are several architectural alternatives, the common features are using Headless VE for compute and a MediaWiki based API middle layer for serving. These commonalities are documented above in the recommended architecture and will not be repeated here.

### LAC Precompute Option: Stream Processing

Status: under consideration




Instead of using a MediaWiki job to warm suggestions in LAC, we could use a standalone stream processor.

A Bento stream processor consumes `mediawiki.page_change` events and requests edit suggestions from LAC for each new revision, warming the cache.

Why Bento vs other stream processor options?

- Change-prop is excluded: it has no maintainer and WMF intends to replace it.
- Flink is excluded for now: we would like streaming cache warming to be easily configurable and deployable. Deploying and operating many small Flink jobs has proven cumbersome.

**Pros**

- `mediawiki.page_change` is a reusable data input. Subscribing to it (or other input streams) is a Pub/Sub pattern that generalizes to future use cases.

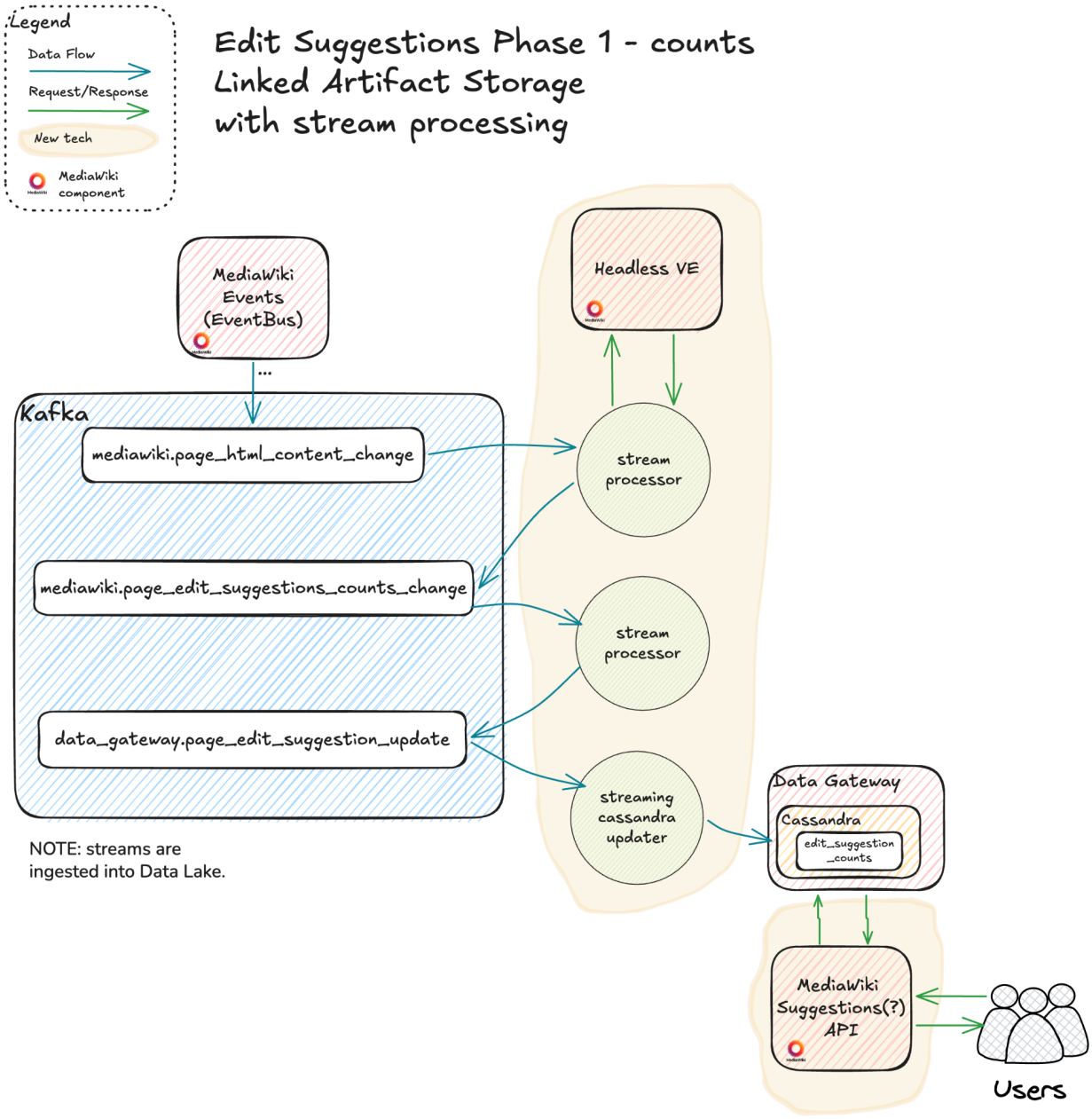
- Benthos (Bento predecessor) is already in use at WMF, and SRE likes it and intends to use it more.

**Cons**

- More moving parts than the MW Job approach.
- New tech to deploy and operate.

### Linked Artifact Storage - Event Driven with Data Gateway & Stream Processors

Status: under consideration




This option takes a “[materialized view](#)” approach. Edit suggestions are fully precomputed and stored directly in Cassandra from an update event stream. This approach is modeled on the [Search Update Pipeline's `weighted_tags`](#) update mechanism.

This materialized view can also be thought of as Linked Artifacts. However, unlike LAC, this is not a cache. There is no recompute on a ‘cache miss’. We have informally been referring to this storage pattern as “Linked Artifact Storage” (LAS).

This option supports suggestion types that cannot be computed on demand, since the pipeline can incorporate inputs from multiple sources — not just Headless VE.

This approach is being taken for the Contributor Count per Page dataset being developed in 2026 for the Attribution API.

### Storage

Any storage could be used to store “linked artifacts”.  We could reuse the same Linked Artifact Cache Cassandra data model, we could use Data Gateway and create non opaque key-value tables for edit suggestions, or we could store this data in some other data store.

For the purposes of this alternative, we will assume Data Gateway and Cassandra.

Edit suggestion counts are stored in a Cassandra table via Data Gateway, e.g.

`edit_suggestion_counts`

| Column | Type |
|---|---|
| `wiki_id` | `string` |
| `page_id` | `int` |
| `rev_id` | `int` |
| `edit_suggestion_counts` | `map<string, int>` |
| `record_update_dt` | `timestamp` |

For Phase 2, storing actual suggestion details might require a table for each suggestion type.

### Precompute

Headless VE will be deployed either as a standalone service, or embedded as a side car container along with a stream processing application.  Embedding Headless VE

would allow the compute service to scale with the stream processor. If we need to backfill, we just scale up the processor.

The stream processing approach also allows us to optimize the compute function (Headless VE) by providing precomputed inputs (HTML).

A stream processor consumes `mediawiki.page_html_content_change.v1` events, passes HTML to Headless VE to fetch the suggestion count, and emits a new `mediawiki.page_edit_suggestion_counts_change` stream. This stream is automatically ingested into the Data Lake and is available for other consumers.

A second stream processor consumes `mediawiki.page_edit_suggestion_counts_change` and emits a `data_gateway.edit_suggestion_counts_update` stream. A Kafka consumer ingests this stream and writes records to Cassandra via Data Gateway.

This mechanism would allow us to build automated ingestion of timely data into Cassandra for future use cases.

Note: there are variations on this pipeline design. The invariant is that an input stream (`mediawiki.page_html_content_change`) is consumed, suggestion counts are fetched from Headless VE, and Cassandra is updated with the result.

If this option is chosen, we will likely use Bento as the stream processor.

### CirrusSearch counts via Search Weighted Tags

Status: rejected

CirrusSearch weighted tags can store label -> value (0-1000) per page. It is easy to produce this data, and it is easy for MW to get it. Preliminary discussions with the Search team indicate that we could temporarily use this feature to store and serve counts of edit suggestions for this phase 1 product. When we implement phase 2 (storing the actual suggestions), we would switch to using the stored suggestions, instead of using CirrusSearch for counts.

The Search team informed us that the latency of a CirrusSearch document query could be close to 100ms.

Erik Bernhardson wrote:

> It's approximately equivalent to a cirrusdoc query (https://en.wikipedia.org/wiki/Special:ApiSandbox#action=query&format=json&prop=cirrusdoc&titles=MediaWiki&formatversion=2), i'm getting ~350ms round trip which accounts for all the mediawiki parts. The backend request itself is probably sub 100ms, but it's still a query that has to run.

If this count were loaded async by JavaScript after the page loads, this 100ms might be acceptable. However, the product requirements are such that the count should be visible to the user when the page loads, so that it is maximally visible to them. Users are likely to scroll past the 'edit' control UIs while reading.

This count will be shown on every logged in mobile web page load, and it is conceivable that it may be shown to all logged web users.

Adding ~100ms to page load times is unacceptable.

# Appendix

## LAC vs LAS - tradeoffs

LAC and LAS share the same compute layer (Headless VE) and similar storage (Cassandra), but differ in how data gets there and what happens when something goes wrong.

### Consistency

LAC self heals. A missed cache warm is recovered on the next user request, which triggers a cache miss and recompute. LAS has no such fallback. A missed event means suggestion counts for that revision are never computed and stored without manual intervention.

Because LAC is self-healing, we can tune TTLs to expire data that is rarely or never accessed. The cache will repopulate on demand if needed. LAS, by contrast, expects to be fully materialized; since there is no recompute fallback, data cannot be safely

expired. This self-healing feature may not be strictly required for Edit Suggestions, but it might be for other future use cases.

For revision artifacts, correctness of LAS is dependent on timing —both that event processing has completed, and that the retention policy hasn't resulted in artifact removal, prior to the client request; Client requests that do not arrive within the period occurring after event processing is complete, but before retention kicks in, will fail.

In Phase 2, we will rely on CirrusSearch weighted_tags to index which pages have suggestions. Weighted_tags is an update event, and may suffer from the same consistency issues. This means that even though LAC can self-heal, the index of LAC may not.

### Operational Complexity

LAC with MW Job Queue has the lowest footprint: no new infrastructure, only new MW Jobs.

LAC with stream processing adds new tech (bento) deploy and maintain.

LAS adds the most moving parts: more stream processors and a new Cassandra update mechanism.

### Generalizability & Optimization

LAC is limited to suggestion types computable by Headless VE on demand.

LAS supports suggestion types that cannot be computed on demand and can incorporate inputs from multiple sources. It also opens up pipeline optimizations not available to LAC.  Parsoid HTML can be consumed from a precomputed stream rather than re-fetched on each computation.

LAS's `data_gateway.edit_suggestion_counts_update` feature introduces a paved path for writing timely data into Cassandra. We may still need this for another future use case if we don't choose LAS here.

### Data Reusability

Both options can emit a reusable event stream (e.g. `page_edit_suggestion_counts_change`) for Data Lake ingestion and other

consumers. In the LAS option, this stream is central to the production pipeline. In the LAC option, it is a side output of cache warming.

### Edit Suggestion compute triggers

Moved to Project Information tab.

---

# ROUGH NOTES

VE currently knows about most, but not all possible edit suggestions. Revise Tone is stored in Cassandra + CirrusSearch. CirrusSearch is used by Growth homepage to look up and display pages with tone issues, and then when a user clicks on “edit”, Growth server side code looks up the actual tone issue from Cassandra and provides it to VE client via JS data.

For VE based edit suggestions, there are many inputs.  TextMatch suggestions are fully created and configured by users via on wiki config.  It is not tractable to compute these kinds of suggestions without MW, let alone without VE.
We will likely need an API wrapper around a “Headless VE” that we can use as a transformer.

Existent edit suggestion sources:

- VE + MediaWiki
- Image Suggestions:
    - data lake -> Cassandra+CirrusSearch weighted tags)
- Link Suggestions:
    - MediaWiki Job (or maintenance script) fetch from Link Recommendation Service -> MySQL table + CirrusSearch weighted tags
- Revise Tone Suggestions
    - events -> LiftWing -> Cassandra+CirrusSearch weighted tags

What is the fastest way to get a count of suggestions (per type?) in Q1?

CirrusSearch weighted tags can store label -> value (0-1000). It is easy to produce this data, and it is easy for MW to get it. Preliminary discussions with the Search team indicate that we can (temporarily) use this feature to store and serve counts of edit suggestions.

### Computing

For non VE suggestions:

- Instead of producing a simple recommendation “exists” value, produce the count of suggestions.
- Image Suggestion is probably always 1 (?) . If not, these are in Cassandra, so we should be able to look them up at page load time and count them.
- Add-A-Link - make MediaWiki produce weighted_tags with link suggestion count values.
- Revise Tone: The value for this tag is currently the confidence of the revise tone suggestion. We can’t reuse the value. But, since this one already has actual suggestions in Cassandra, we should be able to count them. We just have to look them up on page load.

For VE suggestions:

- Editing team should build a headless VE with an API wrapper.
- For each edit: call this API, produce appropriate weighted_tags events.
    - TBD how do to this. Options include
        - Batch: Data Lake + airflow batch
        - Stream processing
        - MW Job Queue

Future: for LLM based suggestions (batch only), we will have to do batch airflow. TBD confirm this?

It would be nice if we had one interface to transfer suggestion change data that wasn’t just weighted_tags events.

For Phase 1, we’ll could go with weighted_tags update events, since we will only need CirrusSearch for counts.

In the future (Phase 2?), we should design a common data model to represent suggestions (or issues?) and produce event (streams) with suggestion data. These events should be ingested into the serving storage (Cassandra+CirrusSearch).

NOTE: we considered the potential for providing HTML to Headless VE instead of fetching it.  This would make it harder to use LAC. However, after discussion, it seems this is not an optimization worth our time, and ParserCache is probably sufficient.
https://wikimedia.slack.com/archives/C024Z8K9CAU/p1781104835586409

### Serving

- Custom MW API that exposes your edit suggestion counts
- Search Team plugs an implementation in that will query it out of CirrusSearch
    - **With the understanding that implementation gets replaced with cassandra down the line**

Later: the same MW API can be used to use cassandra to get this data instead of fetching in from CirrusSearch.

It is not bad to fetch this data from CirrusSearch. The issue is that this particular feature will be shown on every page load.
Dashboards will query CirrusSearch for pages with suggestions, and they should be able to use those results to show the count of suggestions in the future.

# Headless VE Pilot

# Headless VE Pilot

*Tracking ticket:* T432286

On Jul 9, 2026, members of the DPE, Editing, and SRE Teams met to discuss the viability of using headless VE to compute suggestion counts.

In that meeting, SRE identified three key concerns that, if left unaddressed, make the headless VE approach infeasible:

1. **Maintainability:** Headless VE depends on browsers which are frequently updated. Keeping pace with these updates creates a non-trivial maintenance burden.
2. **Security:** Browsers are highly complex and constantly impacted by vulnerabilities. While timely updates are a mitigation (subject to the maintainability risk above), their attack surface is larger than most other production systems.
3. **Resource requirements:** Browsers are generally heavy applications in their CPU and RAM requirements, and depending on the footprint of Headless VE, it could cause significant strain in our already limited resources.

This is a proposal for a bounded pilot that we think could be effective at A) mitigating the above concerns and B) enabling us to learn how (if at all) valuable the product intervention is and what further investment(s) we might need to make should the impact warrant wider deployment.

## Decisions

We are interested in the following immediate questions:

**A.** What impact does surfacing the count of suggestions to logged-in readers on mobile web have on the main DE1.1 metric, that is, constructive edits by distinct registered users with <100 edits?
**B.** Approximately, what resources does Headless VE require under realistic traffic? | T432286

Assuming a positive answer for **A**, which motivates further investment in this product direction, we move on to:

C. With forecasted hardware resources for FY26-27, what's a realistic product roadmap assuming we keep Headless VE (i.e. how might we scale across users, wikis, % traffic)?
D. How might we automate browser updates?
E. What other maintainability, security and resource optimization mitigations would be required by SRE to take on the roles of Incident Response Owner and Configuration and Deployment Owner for Headless VE?

This enables a cost-benefit analysis of Headless VE itself as a solution.

## Proposed Pilot

Question **A** should be resolved via a Test Kitchen experiment, where a fraction of logged-in users are enrolled into the product intervention, as already planned under T430712.

To enable the experiment, and answer **B** without committing to a full Headless VE rollout, we'll proceed with the design outlined under Suggestion Count, with two additions:

- [DPE] The `EditSuggestionsUpdateJob` must pre-emptively filter out revisions done to:
  - Wikidata.
  - Wikis not enrolled in the experiment.
- [Editing] The Headless VE lambda must implement logic to sample a configurable percentage of incoming page IDs, rather than invoking a headless browser on every revision or page ID.

In deploying this, Editing and SRE will agree on a pilot schedule that allows evaluation of **B**. For example:

- Day 1: Sample 1% of pages in 1 wiki, enrolling 0% of logged-in users in the A/B test.
- Day 2: Sample 5% of pages in 1 wiki, enrolling 0% of logged-in users in the A/B test.
- …
- Day 10: Sample 90% of pages in 1 wiki, enrolling 1% of logged-in users in the A/B test.
- Day 11: Sample 90% of pages in 1 wiki, enrolling 10% of logged-in users in the A/B test.

- …

Having concluded the schedule, the following teams are responsible for answering the questions:

| Question | Responsible | Consulted |
|---|---|---|
| **A** | Editing | Experiment Platform |
| **B** | SRE Service Operations | None. |
| **C** | Editing | SRE Service Operations |
| **D** | SRE Service Operations | Data Engineering |
| **E** | SRE Service Operations | SRE Infrastructure Foundations |

# TODO - Stored Edit Suggestions

*Please copy this tab for each distinct unit of data (e.g. a metric or field) and order the data fields by priority in the Project Information tab.*

| **Semantics** | |
|---|---|
| In simple terms, what does this data represent or measure? | |
| In simple terms, how would you describe the data being created? | |
| **Inputs** | |
| What primary data sources (e.g. MediaWiki tables or instrumented events) do we need to transform? | |
| Will we be ingesting, transforming or serving Personal Information? | |
| How often does the source data change? | |
| **Processing** | |
| What transformations (e.g. filtering, aggregations) do we need to perform? | |
| **Output** | |
| Are we producing structured data (like key-value pairs, JSON even if serialized), or opaque data (like images)? | |
| If structured, what is the data model? What types, constraints and relationships exist? | |
| How much staleness is acceptable? For example, can the output be a week old relative to the inputs? What happens if this expectation is not met? | |
| What completeness is acceptable? Can the outputs be computed on samples of the inputs? What happens if this expectation is not met? | |
| What kind of retention is needed for this data? | |
| What amount of historical output data is needed? Do we need to transform the input data from the beginning of time, or only from a certain point in time? | |

| Serving | |
|---|---|
| Who is the client accessing this data? | |
| How do we expect to serve this data to the client? Does it require an API or some other kind of interface? | |
| What kinds of read patterns are expected? Are they sequential or random lookups? Do we need search semantics? | |
| What kinds of writing patterns are expected? Are writes frequent or rare? Individual or in bulk? | |
| Are reads or writes geographically concentrated? | |
| **Operations** | |
| What frequency of reads is expected (e.g. 100 queries/s)? Is there an uneven split between kinds of reads (e.g. search vs retrieval)? | |
| What kind of read latency is acceptable? For example, can a lookup via an API take 200ms? What happens if this expectation is not met? | |
| Roughly, how much data are we producing? How do we expect it to grow over time? | |
| Do we expect a steady stream of input, or bursts? What could cause a burst? | |
| What amount of downtime is acceptable (for transforming data *and* for serving it)? What happens if this expectation is not met? | |

*Data Engineering to fill below this line*

---

# <Data Field> Design

## Background

*(2-3 paragraphs) What background is needed to understand whether this design solves the stated requirements? What systems should the reader be aware of? Are there other projects or high-level vision that influence our choices.*

## Assumptions

*(A few bullet points) What are things that we didn't hear as requirements or decide yet, but assume to be true in this design? Things like edge behaviors for systems or metric definitions, or even downstream usage of the data.*

# Requirements

*(A few bullet points) What problem are we solving? How do we quantify it? How fast should things be? How much data are we transforming?*

*If you're re-using this template outside of Prep Pantry, please fill this out! For Prep Pantry, no need to repeat the table above. If you need to expand on it with additional or more detailed requirements that surfaced along the way, this is the place.*

*The chosen design should be relatively "obvious" and follow from these requirements.*

## Design

*What does the data pipeline look like? How are we transforming and serving the data?*

## Ownership

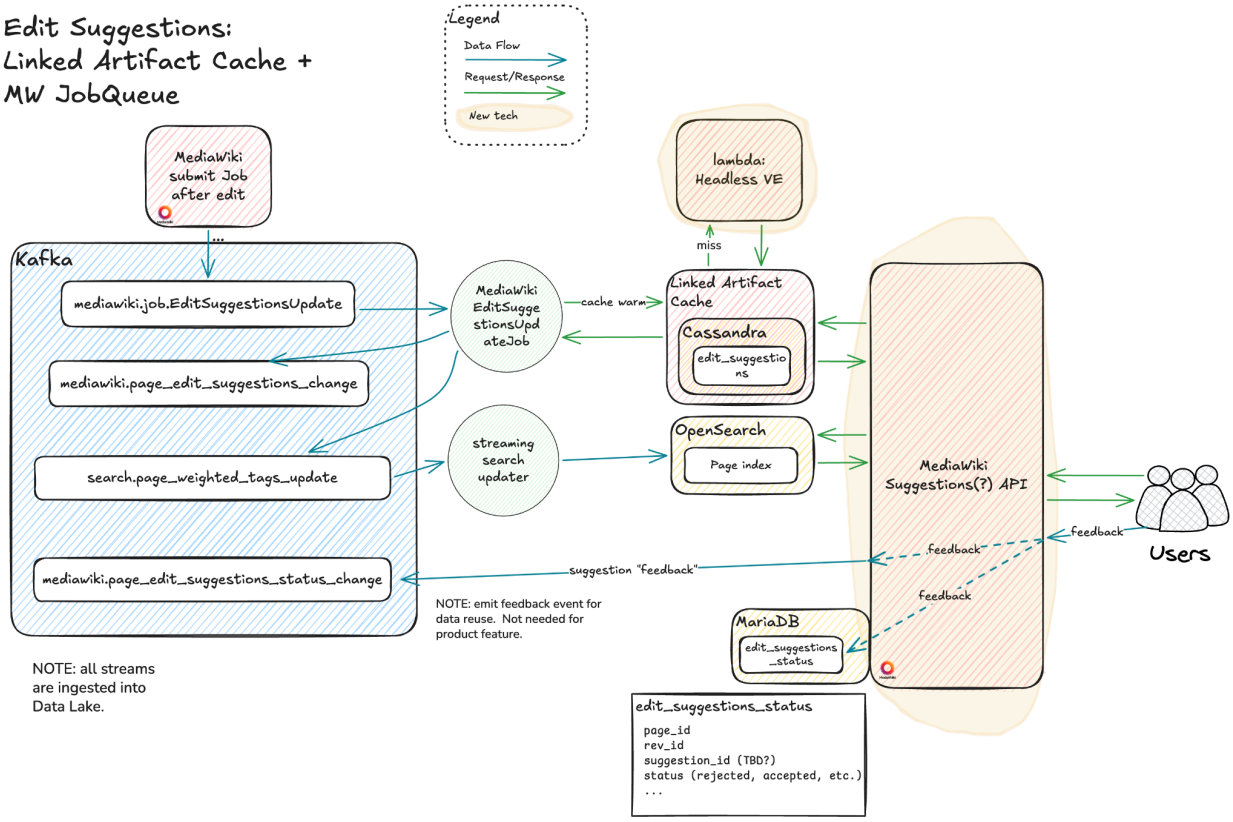
*What are the owned components of this design, and who owns them? Bonus points: try using the framework in Ownership Roles and Responsibilities.*

## Alternatives

*(Subsections, 1-2 paragraphs each) Other options that satisfy some-but-not-all requirements, or violate assumptions, and why we rejected them.*

# ROUGH NOTES

# Diagraming thinking



# 📜 Scroll

# Scroll

SCROLL/EditCheckHeadless
⚓ T432715 New Scroll Request for: EditCheckHeadless
SCROLL/EditCheckHeadless

| | | | | | |
|---|---|---|---|---|---|
| Prerequisites | 1.1 | Stub wikitech page created (EditCheckHeadless) | Done | T434129 | dchan created at least a stub |
| Prerequisites | 1.2 | Service catalog entry | N/A | | Catalog doesn't appear to exist yet; likely nothing to do |
| Prerequisites | 1.3 | Contact info updated | Done | T434134 | dchan created ticket and updated contact info |
| Prerequisites | 1.4 | SRE review of the design | In Progress | | This technical pilot is the SRE design review |
| Prerequisites | 1.5 | Security review | In Progress | T432708 | Run the requested Claude skills on the repo; review results; nudge security team |
| Prerequisites | 1.6 | Data persistence review | Done | T414140 | Using LAC as suggested |
| Operating Procedures | 2.1 | Flesh out wikitech page re: user-facing feature the server powers | To Do | | See README |
| Operating Procedures | 2.2 | Baremetal service setup | N/A | | Not a baremetal service |
| Operating Procedures | 2.3 | Kubernetes setup | To Do | T434109 | Still needed |
| Operating Procedures | 2.4 | Service URL | Needs Decision | | Work out whether one is actually needed |
| Operating Procedures | 2.5 | Staging environment | To Do | | No staging environment currently |
| Operating Procedures | 2.6 | Traffic estimate | Done | T433024 | Estimate exists (sampling) |
| Operating Procedures | 2.7 | Accept traffic from CDN | N/A | | Service doesn't accept traffic from the CDN |
| Operating Procedures | 2.8 | Datacenter switchover handling | Needs Decision | | Determine if service is "in discovery"; likely doesn't need to care about switchovers even if so |
| Operating Procedures | 2.9 | Document systems we communicate with on wikitech page | To Do | | See README |
| Release Confidence | 3.1 | Link to repo and recent image | Partial | T433514 | Repo is linked; recent image not yet created |
| Release Confidence | 3.3 | Rollout plan documented with failure points | Partial | T433024 | Ticket counts as a plan; write up on wikitech and describe failure points |
| Release Confidence | 3.4 | Configuration and deployment owners | To Do | | Identify owners (editing people); document and learn how to deploy these servers |
| Observability | 4.1 | Logs and metrics | In Progress | | Sending logs to logstash; prometheus metrics explicitly off; determine if that must change |
| Observability | 4.2 | Business metrics | N/A | | Likely N/A for pilot; no user-facing flows; |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | success is "does this work?" |
| Observability | 4.5 | Dashboards | To Do | | None exist; investigate creating from 4.1 metrics; crib from existing LAC/jobqueue health metrics |
| Observability | 4.7 | Alerts | To Do | | None specified; decide what warrants alerts; learn Alertmanager |
| Observability | 4.8 | Incident response manager and runbooks | To Do | | None known to exist |
| Reliability and Performance | 5.3 | Health/liveness endpoint | To Do | | Do not currently have one |
| Reliability and Performance | 5.4 | Single points of failure analysis | Partial | | README likely covers it; pull out onto wikitech |