ChatGPT

DanceResource Wiki Multi-Language Translation System Blueprint

Architecture Overview

- **Server-side Translation Bot:** The system will run as a standalone service (not a MediaWiki plugin) on the same server as the wiki. It interacts with MediaWiki via its HTTP API.
- **Tech Stack:** A simple, maintainable language like **Python** (or similar scripting language) will be used for implementation. Python has robust HTTP libraries and existing localization toolkits, making it suitable for rapid development.
- **Docker Deployment:** The bot is containerized with Docker, ensuring easy deployment and consistency across environments. The container includes the bot application and any necessary dependencies (e.g. translation libraries, database clients).
- **Modular Design:** The application will be organized into modules for different concerns:
- **Watcher/Scheduler:** Monitors wiki pages for updates and manages translation job queues.
- **Translator Engine Module:** Handles calls to external machine translation services or models.
- **Glossary & QA Module:** Manages termbase (glossary) and style guides, and performs quality checks.
- **MediaWiki API Client:** Abstracts MediaWiki API calls (fetching pages, posting translations, etc.).
- **Minimal Footprint:** Since developer maintenance is minimal, the design favors clarity and simplicity over complex optimization. External services (like translation APIs or libraries) are used for heavy lifting, keeping custom code lean.

Translation Workflow & Job Lifecycle

1. **Change Detection:** The bot continuously detects changes on the English source wiki pages. This can be done by polling MediaWiki's recent changes API or via a webhook if available. When a page marked for translation is edited, a **translation job** is created and queued (with a timestamp and priority level).
2. **Job Prioritization:** Each job is assigned a priority based on page importance:
3. Critical pages (front page, key resources) = high priority.
4. Less trafficked or minor pages = low priority.
5. The scheduler ensures high-priority jobs are processed first, while low-priority ones are batched or delayed to reduce load.
6. **Page Fetch & Segmentation:** When executing a job, the bot fetches the page content via the MediaWiki API (e.g. `action=query&prop=revisions&rvprop=content`). It then **segments the content into translatable units**, preserving wiki markup:
7. Non-translatable parts (wiki templates, infoboxes, references, and any content wrapped in `<translate>` tags as non-translatable) are identified and isolated so they remain unchanged.
8. Translatable text is split into smaller units, typically by existing Translate extension markers (`<!--T:1-->`, `<!--T:2-->`, etc.) or by section breaks. This mirrors how the Translate extension breaks text into small pieces for translation [1].

9. Markup like `[[links]]`, templates (`{{Infobox...}}`), and tags (`<ref>...</ref>`) are **preserved** by either leaving them in place or replacing with temporary placeholders during translation to avoid corruption.
10. **Machine Translation Generation:** For each translatable segment, the bot obtains a machine translation:
11. It uses a **cost-effective, high-quality translation engine** that supports 50+ languages. Possible choices include cloud services like *Google Translate API* or *Azure Cognitive Translator* (broad language support, pay-per-use), or an open-source neural model (e.g. **NLLB** or **MarianMT** from HuggingFace) for self-hosting.
12. A fallback chain can be implemented: e.g. try an open-source or cached translation first, and if unavailable or poor quality, fall back to a cloud API. This keeps costs down while ensuring quality.
13. **Glossary Enforcement:** Before sending text to the engine, the bot can enforce glossary terms. For example, protected terms in the source can be wrapped in markers so the engine doesn't translate them (many APIs allow a `<keep>` tag or similar). Alternatively, after translation, the bot will post-process the output to replace certain phrases with the preferred glossary translations.
14. The engine returns a translated segment, which the bot then post-processes to reinsert any templates/markup placeholders and ensure the wikitext syntax is intact.
15. **Draft Insertion into Wiki:** The bot inserts the translated text back into the wiki using the Translate extension's workflow:
16. It uses the MediaWiki API to edit the translation pages or message group units. For example, each segment corresponds to a "message" in the Translate system (often stored as subpages like `Page_Title/SegmentID/LangCode`). The bot calls `action=edit` with an appropriate page title and translation text for each unit, or uses a batch translation API if available.
17. The inserted translations are marked as **draft/unreviewed**. The bot can add a machine translation disclaimer, either in the edit summary or by prefixing the content with the Translate extension's fuzzy flag (`!!FUZZY!!`) to indicate it needs review. This makes it clear these are machine-generated.
18. The Translate extension will treat these as translations provided by a user (the bot), but not yet marked as "proofread." They will be immediately visible to end-users (if desired), but highlighted as not reviewed.
19. **Optional Human Review:** For high-priority or sensitive pages, maintainers or volunteers can review and adjust the translations:
20. The system does **not require** human approval to publish machine translations for low-priority pages (ensuring full coverage of 50+ languages quickly).
21. When human reviewers are available, they can use the standard Translate extension interface to proofread and mark translations as "reviewed" (the bot can also call `action=translationreview` via API to programmatically mark certain translations as reviewed, if we decide to auto-approve some).
22. Review feedback (e.g., corrections) can be fed back into the system – the bot could update its glossary or translation memory based on human edits.
23. **Job Completion:** Once all segments of a page are translated and saved, the job is marked complete. The bot records the revision ID of the source page that was translated. This allows it to detect future edits (if the source page changes, the stored revision ID will differ from the latest, triggering an update job).
24. **Scheduling & Retries:** If a job fails (due to API errors, network issues, etc.), it isn't lost:
25. The job is requeued with an incremented retry count. Exponential backoff can be applied (wait longer for subsequent retries to avoid flapping).

26. After a certain number of failed attempts, the job is marked as **errored** and moved to a separate list for manual inspection. This prevents infinite loops on problematic pages. An alert/log will be generated for developers to check the issue.
27. The scheduler runs continuously (or at set intervals) to pick up new or retried jobs. It can also handle rate limiting to avoid overwhelming either the wiki or the translation API (e.g., process N pages or M segments per minute).

Page Parsing Strategy

- **Template and Markup Preservation:** Before translation, the bot sanitizes the source wikitext:
- It **protects non-translatable fragments** by replacing them with placeholder tokens. These include template invocations (`{{...}}`), HTML comments, references `<ref></ref>`, category links, image links, and any other content that should remain unchanged across languages.
- For example, `{{Infobox dancer|name=...}}` might be replaced with a token like `<TEMPLATE1>` during translation. After obtaining the translation for the surrounding text, the token is swapped back with the original `{{Infobox...}}` code.
- This ensures the machine translation engine doesn't garble wiki syntax. It also means things like file names or template parameters stay exactly as in source (unless they too need translation, which is rare and would be handled case-by-case).
- **Segmentation Logic:** The bot either uses the Translate extension's existing page segmentation or implements a custom segmenter:
- If pages are already marked for translation, the extension will have injected tags (`<!--T:n-->`) denoting segments. The bot can split on those markers, sending each segment separately to the translator. This aligns with how translators would normally work with the content.
- If custom segmentation is needed (for example, if some pages are not yet marked but we still want to translate them), the bot can split text into paragraphs or sentences, but must be careful not to split inside templates or links.
- The segmentation strategy closely mirrors human translation units, which the Translate extension is designed around [2] [3]. Keeping segments small ensures better machine translation quality and easier human review.
- **Non-Translatable Content Handling:** Certain parts of a page should **not be translated** at all:
- Names of specific people, organizations, or works (unless there are official localized names).
- Codes, identifiers, or URLs.
- These can be preserved by wrapping them in `<translate><!--:-->` tags or by the placeholder approach described above. The glossary can also specify terms that should remain in English (or source language) across all languages.
- **Reconciliation:** After translation, the bot merges translated segments back together with the original markup structure:
- It reconstructs the page in the target language as the Translate extension expects. For example, it will assemble all translated units so that the page `Dance_Techniques/fr` (French) consists of the French segments in the correct order, interleaved with any non-translated wiki markup (templates, tables, etc. remain in place).
- The final assembled page is then saved via the API. (If the extension requires a specific method to publish the whole page, the bot will ensure all units are saved which effectively produces the translated page.)
- A final sanity check can run here to ensure the reconstructed page has no obvious markup errors (e.g., unclosed tags or broken template braces).

Translation Engine & Fallback Logic

- **Primary Engine:** Choose a primary machine translation service that balances quality and cost. **Google Translate** or **Microsoft Translator** are reliable choices with broad language support (100+ languages). For example, Google's Neural Machine Translation offers high quality for many language pairs and scales well.
- **Fallback Engines:** Implement a fallback mechanism for resilience and coverage:
- If the primary API fails (network downtime or rate-limit) or doesn't support a particular language pair, the bot will automatically switch to an alternate service. Possible backups include **LibreTranslate** (self-hosted, supports ~18 languages) or **Amazon Translate** (high-quality, ~70 languages).
- Another fallback could be an on-premise model like Meta's **NLLB-200** (which covers 200 languages) or the OPUS-MT models [4] hosted via HuggingFace. These can run in Docker containers with GPU support for faster inference. Initially, due to resource constraints, on-prem models might be used for only a few languages or for experimentation.
- **Quality Tuning:** The system can experiment with different engines per language to see which yields the best results:
- For example, DeepL (if it supported more languages) might be preferred for European languages due to higher fluency, while Google might be better for others. The architecture allows plugging in new translators or updating the engine without changing the core workflow.
- The bot keeps track of which engine was used for each translation. This data can later inform if certain engines consistently produce translations requiring fewer corrections.
- **Cost Management:** Since this project is open-source and likely budget-conscious, the bot could incorporate logic to minimize cost:
- Use free or open-source translation for languages where quality is acceptable, and use paid APIs for languages where open models are insufficient.
- Cache translations of identical text (or use translation memory) so that if the same sentence appears in multiple pages, it isn't translated via API repeatedly. The Translate extension's translation memory [4] can assist in suggesting previous translations of the same text, which the bot can leverage to avoid redundant API calls.
- **Machine Translation Disclaimers:** Every translation unit inserted will include a note (in comments or edit summary) indicating it was auto-generated. This transparency aligns with community expectations and invites improvement:
- e.g., Edit summary: "Machine translation by bot (draft, needs review)."
- If the wiki interface supports it, the bot can also add a small banner at the top of translated pages: "This page is automatically translated. You can help by reviewing and improving the translation."

Terminology Management & Glossary Agent

- **Termbase Generation:** An **AI-driven glossary agent** periodically analyzes the wiki's content to build a terminology database:
- It scans English pages for domain-specific terms (e.g. dance styles, technical terms, names of organizations or events). This can be done via keyword extraction techniques or by looking at words that are frequently used and likely not common in general language.
- For each term, the agent gathers candidate translations in target languages. It can use Wikidata or Wikipedia interlanguage links, existing glossaries in the dance domain, or even query an AI model

for suggestions. For example, if “ball change” is a dance term, the agent finds what it’s called in Italian, Serbian, etc.
- The result is a **multilingual termbase**: a mapping of English terms to their approved translations (or guidelines to keep them in English if appropriate).
- **Style Guide Creation:** The agent also creates a **per-language style guide**:
- Using large language models (or rules), it drafts guidelines on tone, formality, and conventions. E.g., for Serbian it might note “Use Cyrillic script (if applicable) and formal second person plural for public content; avoid anglicisms when a common Serbian term exists.” For Italian: “Use formal tone for wiki articles; avoid literal translation of idioms; maintain consistent tense,” etc.
- These guides are refined by examining high-quality translations (if any exist) and general localization best practices for each language.
- The style guides and glossary can be stored in a structured format (e.g. YAML or JSON) in a repository or database. This allows easy updates and usage by the bot.
- **Integration into Translation Process:** The glossary and style rules are enforced at multiple points:
- When sending text for machine translation, the bot can prepend critical glossary entries as hints (some translation APIs allow a glossary file to be uploaded, ensuring the engine uses those translations for specific terms).
- After translation, the QA step (described below) checks the output: it verifies that all occurrences of glossary terms are correctly translated. If a forbidden term or an inconsistent translation is found, the bot can automatically fix it (using the correct term) or flag the segment for human review.
- Style rules (which are harder to enforce automatically) are used in QA to flag potential issues. For instance, if the style guide says “avoid passive voice” or “use formal address,” the bot might not reliably rewrite the text, but it could highlight those segments to be looked at by a human reviewer or future AI refinement.
- **Translation Memory:** In addition to the glossary, the system maintains a **translation memory (TM)** – essentially a database of all sentences/segments that have been translated before:
- Each time a new translation is confirmed (by the bot or a human), it’s stored in the TM with the source and target segment.
- When a similar or identical source segment needs translation later, the bot can retrieve the previous translation suggestion (with a similarity score). This improves consistency across pages and can dramatically speed up the process as content scales.
- The MediaWiki Translate extension has built-in support for translation memory and suggestions [5], which the bot can query via the API (e.g., using `action=query&list=translationmemory` if available, or by maintaining its own TM store).
- **Glossary/Style Guide Updates:** External contributors or language experts can update the termbase and style guides:
- The data might be stored in a git repository (since the project is open-source). Pull requests can be made to update a translation of a term or adjust a guideline for a language.
- The bot could validate these contributions (e.g., ensuring no duplicate terms, proper format) and automatically deploy updated glossaries. A CI pipeline might run tests to ensure the changes won’t break existing translations (for example, if a term translation is changed, the pipeline could re-check all translations for consistency).

Quality Assurance & Error Handling

- **Pre-Publication Checks:** Before committing translated content to the wiki, the bot runs a QA pipeline on each segment and page:

- **Markup Integrity Check:** Ensure that all wiki markup is balanced and uncorrupted. The bot can parse the translated wikitext in a sandbox mode (using MediaWiki's `action=parse` or a wikitext parser library) to see if it throws errors or produces the expected structure. Unclosed tags, broken link syntax, etc., will be caught here.
- **Template & Link Preservation:** Verify that placeholders were replaced correctly and templates/links remain intact. For example, count that the number of `[[` and `]]` brackets in the translation matches the source, and that template tokens have been fully restored.
- **Glossary Compliance:** As mentioned, scan the translated text for each glossary term:
    - If a term should be translated a certain way, check that it appears. If not, either replace it automatically or mark the translation as fuzzy/unreviewed with a note.
    - If a term should remain in English (source), ensure it wasn't erroneously translated.
- **Length & Formatting Checks:** Compare length of source vs translation. Large discrepancies (e.g., a translation is less than 50% or more than 200% the length of source) might indicate dropped content or repetition – flag these for review. Also check for duplicate punctuation or awkward spacing introduced by the MT engine.
- **Language-specific Localizations:** Some languages have special requirements (for instance, different quotation mark styles, decimal comma vs point, etc.). The style guide can provide some regex or rules to validate these. The QA can apply those rules to catch obvious issues (e.g., English quote marks in French text).
- **Untranslated or Fuzzy Segments:** Ensure no segment is left untranslated. If the engine fails to produce a translation (or returns the source text unchanged for some reason), the bot can detect that (source and target identical) and either retry with another engine or mark the segment as needing manual translation.
- **Post-Publication Monitoring:** Even after publishing translations, the system monitors for errors:
- It can periodically fetch a random sample of translated pages and run the parse check to ensure they still render correctly (especially useful if templates were updated or if a translation was manually edited incorrectly).
- If a problem is found, it flags the page and can automatically revert the problematic translation unit to an empty state (so that it doesn't break the page for readers) while notifying maintainers.
- **Logging & Reporting:** All translation actions and QA results are logged:
- A dashboard (could be as simple as a wiki page or a small web UI) shows recent translation jobs, their status, any errors, and stats like number of segments translated, time taken, etc.
- Errors are categorized (markup error, glossary mismatch, API failure, etc.) to help developers or volunteers quickly understand and address them.
- The system could also send email or chat notifications on critical failures (for example, if the translation API key expired or the wiki API is unreachable).
- **Retry & Fallback for Errors:** As described in the job lifecycle, the system will retry failed jobs. Specifically:
- **Translation API errors:** If the primary engine fails for a segment, the bot will log the failure and automatically call the fallback engine for that segment. This could happen within the same job run.
- **Network issues or Wiki API errors:** The job is halted and rescheduled. If the wiki is down for maintenance, the bot will simply hold off and retry later.
- **Content errors:** If QA finds issues that it cannot fix automatically, the job may complete with warnings. Those warnings could mark the translation as fuzzy (so it doesn't count as fully done) and list the issues for a human to fix at convenience. However, the translated content (even if imperfect) is still saved to make it available to readers immediately unless it's completely broken.

MediaWiki API Integration

- **Page Content Retrieval:** Uses `action=query` with appropriate parameters to get source page content and revision info. For example:

  ```
  api.php?action=query&titles=Dance_Techniques&prop=revisions&rvprop=content|ids
  ```

  This returns the wikitext of "Dance_Techniques" (English source) and its revision ID, which we use to track changes.
- **Listing Translatable Pages:** The bot can retrieve a list of pages marked for translation via `action=query&meta=messagegroups` or by category membership. Translatable pages often belong to a category like "Translations" or have a specific prefix. The **Translate extension** organizes page translations as message groups (e.g., group id “page-Dance_Techniques”) [6] . The bot can query these to get all relevant pages.
- **Fetching Translation Units:** To get the text of each translation unit (segment), the API module `action=query&prop=messagecollection` can be used, specifying the page’s message group and source language. This returns all segments with their content and translation state. Alternatively, the bot can parse the raw wikitext for `<!--T:n-->` markers to split content, but using the extension’s API is cleaner.
- **Editing Translations:** The primary way to save a translation is via the standard edit API:
- Construct the title for the translation unit or subpage. For example, if a page “Dance_Techniques” is marked for translation, unit 5 in French might be stored at something like “Dance_Techniques/5/fr”. (The exact pattern can be confirmed via the message collection data or how the extension names subpages.)
- Call `action=edit` with the title, language text, and an edit token. The bot will have a bot account with permissions to edit translations. The edit includes `summary` indicating machine translation.
- If possible, use MediaWiki’s ability to mark edits as bot edits to avoid flooding recent changes.
- **Translation Review API:** For pages where we want to automatically mark translations as reviewed (perhaps if we trust the engine or after a human lightly post-edits them), use `action=translationreview` with the revision ID of the translated message [7] . This will mark that particular translation as “reviewed/approved” in the Translate system. By default, the bot will **not** auto-review its own translations (to keep them distinguishable from human-verified content), except possibly on user request for certain languages.
- **Tokens and Authentication:** The bot uses a bot account’s credentials via API. It must obtain a CSRF token (`action=query&meta=tokens`) [8] for edit and review actions. All write actions are done with POST requests. The bot will handle login (maintaining a session cookie) or use an API token if available for that account.
- **Error Handling:** The MediaWiki API will return error codes if something fails (e.g., edit conflicts, bad tokens). The bot checks responses for errors and decides to retry or log accordingly. Some actions like marking a page for translation (if needed) might not have direct API support (the StackOverflow discussion noted no API for “mark for translation” in older versions). In such cases, an admin may have to mark pages manually using the web interface, or we consider running a maintenance script via CLI for initial setup. Once pages are marked, the rest (fetching and translating) is achievable through the API.

- **Performance Considerations:** When dealing with 50+ languages, the number of API calls can grow quickly (each segment per language is an edit). To manage this:
- Use bulk queries where possible (e.g., fetch multiple pages or segments in one query by listing titles or using generator parameters).
- Rate-limit edits to stay within MediaWiki API limits. The bot can also yield between edits to avoid server strain.
- Monitor API continue parameters for listing queries (if a page has many segments, the messagecollection query might require pagination).

Deployment & CI/CD Pipeline

- **Dockerized Deployment:** The entire bot application is packaged in a Docker image. Configuration (like API credentials, chosen translation engines, etc.) is provided via environment variables or a mounted config file. This allows the same image to be used for development, staging, or production simply by changing configs.
- **Continuous Integration (CI):** A CI workflow (e.g., GitHub Actions) can be set up to run tests and build the Docker image on each commit:
- **Unit/Integration Tests:** Using SpecKit and spec-driven development, tests will be written for each module. For instance, a test might simulate a wiki page input and verify that segmentation produces expected tokens, or that the glossary replacement works correctly. Another test might mock the translation API to ensure the bot handles the responses as expected.
- **Spec Validation:** Because the project follows spec-driven (SDD) principles, the spec (this blueprint and subsequent refined specs) can include an explicit checklist. The CI can include a step to ensure the implemented code meets the spec (possibly via SpecKit's tools). For example, if the spec says "should support at least 10 languages initially," the test can instantiate 10 language jobs and see that none fails.
- **Continuous Deployment (CD):** Optionally, set up CD to automatically deploy the new Docker image to the server:
- This could be triggered when changes are merged to main. Deployment might simply pull the new container and restart the bot service.
- Before full deployment, there might be a staging wiki or a dry-run mode: the bot could run against a test wiki (or a subset of pages) to ensure nothing has broken. This is especially useful when updating the translation logic to avoid mass publishing errors.
- **Stage-wise Rollout:** The implementation will be delivered in stages (milestones), each fully tested:
- **Stage 1 – MVP (Minimum Viable Product):** Basic change detection and translation for 2 languages (Serbian and Italian). No glossary or advanced QA yet – just fetch page, machine translate segments, and save to wiki. Ensure the process works end-to-end manually.
- **Stage 2 – Glossary & QA:** Introduce the terminology database and style guide for Serbian and Italian. Implement the QA checks (markup preservation, glossary enforcement, etc.). At this stage, also integrate a second translation engine as fallback and test the fallback logic.
- **Stage 3 – Scale to 10–20 Languages:** Add more languages (Spanish, French, German, Chinese, etc.). This will involve expanding the glossary/style guide generation (the AI agent might be run once for each new language to bootstrap the guidelines). Optimize performance of batch processing (maybe translate multiple pages in parallel, if API limits allow).
- **Stage 4 – Full 50+ Language Support:** Hardening and optimization to handle many languages. Possibly integrate with a caching layer or database to handle the volume of segments. Also

implement any needed **fallback logic** at scale (e.g., if a certain engine has usage quotas, distribute calls among alternatives).

- **Stage 5 – Human-in-the-loop Enhancements:** Provide convenient ways for volunteers to review and fix translations. Perhaps generate periodic reports of "Most viewed machine translations that are unreviewed" to guide human editors. Also, incorporate human feedback (if a user edits a translated page directly, sync that back to the translation memory).
- **Monitoring & Maintenance:** The deployment includes monitoring hooks:
- Container health checks ensure the bot process is running. If it crashes, Docker can auto-restart it.
- Logging from the bot can be aggregated (using something like the ELK stack or simply log files on disk) to track activity.
- A scheduled job (via cron or within the app) can periodically do maintenance tasks like cleaning up old logs, refreshing the glossary via the AI agent, or re-validating a random set of translations.
- Developer documentation is hosted with the code, making it easy for new contributors to set up a dev environment (e.g., how to spin up a local MediaWiki with the Translate extension for testing).

Open-Source Collaboration and Governance

- **Public Repository:** The project's code will reside on GitHub under the danceresource.org organization. It will include a README, setup instructions, and contribution guidelines.
- **Spec-Driven Development:** Using **GitHub SpecKit**, the project maintains up-to-date specifications and plans:
- The spec (like this document) is version-controlled. Changes to desired behavior are made in the spec first, then implemented in code. This ensures clarity of purpose and helps external contributors understand the system's design.
- Alongside the spec, automated tests and "definition of done" checklists are provided. Contributors adding a feature or fixing a bug can run `specify` to see the expected behavior and run tests to verify.
- **Contributor Workflows:** External contributors can fork and propose changes. To facilitate improvements:
- **Language Packs:** Contributors could improve the glossary or style guide for their language via a structured file. A GitHub Action might validate these (to avoid format errors) and even deploy them automatically to the running bot after approval.
- **Translation Accuracy:** If contributors want to improve how the translation engine handles certain patterns (for example, maybe a regex post-processing to fix common errors), they can propose changes. The spec-driven tests would ensure such changes don't break existing functionality.
- **Documentation:** All key aspects (architecture, how to add a new language, how to deploy, etc.) are documented in the repo wiki or docs folder. New contributors are encouraged to read the spec and run the system in a dry mode to see how it works.
- **Community Input:** The system welcomes feedback from users of the wiki:
- A feedback mechanism (like a special wiki page or form) can be provided where readers can report bad translations. The bot can periodically scrape this and either schedule those pages for retranslation (maybe with a different engine or updated glossary) or flag them for human review.
- If a particular translation in one language is consistently problematic, community members for that language could be granted a way to adjust the machine translation engine or parameters for that language (perhaps via config).
- **Licensing:** All code will be open-source (likely under an MIT or GPL license, given the context of MediaWiki's GPL license). This ensures transparency and that improvements can be shared back.

Similarly, any AI-generated artifacts (glossaries, style rules) will be open data so that others can reuse them for related projects.

- **Spec Evolution:** As the project grows (more languages, more features), the specification will evolve. SpecKit allows the spec to be the single source of truth – when new requirements come (e.g., support a new translation API, or integrate with a new version of MediaWiki), maintainers will update the spec and then let the AI agent/coding process adjust the implementation accordingly. This living spec approach makes maintenance possible even with minimal developer involvement, because future AI coding assistants can rely on the spec to make correct changes.

Conclusion

This blueprint outlines a **spec-driven, phased approach** to implementing an agentic translation system for DanceResource Wiki. By starting with a solid foundation (server-side bot, MediaWiki API integration, basic MT) and gradually layering on glossary intelligence, QA checks, and support for dozens of languages, the project can deliver value quickly and improve over time. Each stage will be validated with tests and guided by the specification to ensure reliability. The end result will be an open-source, automated translation workflow that can scale to 50+ languages, with community-driven refinements, all while maintaining the integrity of the wiki's content and the spirit of open collaboration.

Sources:

- MediaWiki Translate extension splits pages into small units for translation [2], and it provides translation memory and other aids to support consistency [3]. These features guide our design for segmenting content and leveraging past translations.

---

[1] [2] localization - How to write translated MediaWiki articles through the api using Translate extension? - Stack Overflow
https://stackoverflow.com/questions/13725344/how-to-write-translated-mediawiki-articles-through-the-api-using-translate-exten

[3] [4] [5] [6] [7] [8] Help:Extension:Translate/API - MediaWiki
https://www.mediawiki.org/wiki/Help:Extension:Translate/API