ChatGPT

XliffBatchTranslate – Specification Document

Purpose

The **XliffBatchTranslate** project is a .NET 8 console application designed to translate large numbers of XLIFF files from English into a chosen target language. It wraps an LLM-based translation service exposed through an HTTP API (the project defaults to LM Studio but can be pointed at any service that implements the Chat Completion API). The tool understands XLIFF structure, preserves inline tags and placeholders, provides caching to avoid redundant translations, and outputs translated files while reporting progress to the console.

High-Level Workflow

1. **Command-line invocation** – Users run the tool with parameters specifying an input directory, an output directory and a target language. Optional parameters allow overriding the translation endpoint and model. The program demonstrates the required syntax (`dotnet run -- <inputFolder> <outputFolder> <targetLanguage> [endpoint] [model]`) and prints an example when insufficient arguments are supplied [1].
2. **Normalization of language names and codes** – The program accepts either language codes (e.g., `fr` or `es-ES`) or human-readable names (e.g., "french"). Two private methods map these inputs to a canonical language name and a BCP-47 language code [2] [3]. The mapping covers several common languages and falls back to capitalizing the user-supplied string.
3. **File enumeration** – The application scans the input directory for files ending in `.xlf` or `.xliff` [4]. It optionally pre-counts the total number of `<trans-unit>` elements across all files to provide an overall progress estimate [5].
4. **Translation loop** – For each XLIFF file, the program creates a new output file in the corresponding relative path. It instantiates an `XliffTranslator` with a configured `LmStudioClient` and `XliffTranslateOptions`, and calls `TranslateFileAsync` to translate the file. Progress for each file and overall statistics are displayed on the console [6].
5. **Reporting** – After processing all files, the program prints a summary including the number of files, total translation units, number translated, skipped, cached and failed [7].

Command-line Options

The executable's `Main` method expects at least three arguments: the input folder, the output folder, and the target language. Two optional arguments allow specifying a custom endpoint and model. The defaults are:

Argument	Description	Default
`inputFolder`	Folder containing `.xlf` / `.xliff` files to translate	–

Argument	Description	Default
`outputFolder`	Destination folder for translated files	–
`targetLanguage`	ISO code or name of the desired language	–
`endpoint`	URL of ChatCompletion-compatible endpoint	`http://127.0.0.1:1234/v1/chat/completions` [8]
`model`	Name of the model to use	`Unbabel/TowerInstruct-7B-v0.2` [9]

Core Components

Data Models (`Models.cs`)

- `ChatCompletionRequest` – Represents a request to the translation service. It contains the model name, an ordered list of chat messages (each with a `role` and `content`), a temperature and optional maximum token limit [10].
- `ChatMessage` – Represents a single message in the ChatCompletion request with `role` (e.g., `user`, `system`) and `content` [11].
- `ChatCompletionResponse` **and** `Choice` – Represent the structure of the JSON response. The response contains an array of `choices`, and each choice can include a `message` (preferred) or a fallback `text` field [12].
- `XliffTranslateOptions` – Holds configuration for a translation run: target language name and code; whether to translate when the current target matches the source (`TranslateIfTargetMissingOrSameAsSource`); translation parameters like temperature and maximum tokens; and whether to cache results [13].
- `XliffTranslateStats` – Records totals for processed translation units, translated count, skipped count, cache hits and failures [14].
- `FileProgress` – Records progress within a file; stores the number of processed units and total units [15].

LM Studio Client (`LMStudioClient.cs`)

`LmStudioClient` wraps an HTTP client and provides a single asynchronous method `TranslateStrictAsync` that sends a prompt to the translation service.

- **Configuration** – The client is initialized with an `HttpClient`, the endpoint URL, the model name and an optional system message [16].
- **Prompt Construction** – The method constructs a user prompt like “Translate the following text from English into <targetLanguage>” followed by the source text and a language label [17]. The system message (if provided) is prepended as a separate chat message [18]. A `ChatCompletionRequest` is assembled with temperature set to 0 and a configurable maximum token count [19].
- **Request/Response Handling** – The client sends a POST request with the JSON body to the endpoint and reads the response. It deserializes the JSON into a `ChatCompletionResponse` and returns the

trimmed `message.content` if available or falls back to `text` [20]. On error or parse failure, it returns `null` rather than throwing.

XLIFF Tokenization (`XliffTokenization.cs`)

Mixed-content `<source>` elements in XLIFF can contain nested tags (e.g., `<g>` or `<x>`). Translating such content directly often causes the model to rearrange or remove tags. `XliffTokenization` addresses this by replacing each non-text node with a stable placeholder token and later reinserting the original nodes.

- `ExtractTextWithTokens` – Iterates through the nodes of a given `<source>` element, concatenating text nodes and replacing any non-text nodes with tokens like `__XLF_TAG_0__`. It returns the concatenated text and a list of cloned original nodes [21].
- `RehydrateNodesFromTokens` – Reconstructs a sequence of `XNode` objects from a translated string by finding token positions and inserting the original nodes back in place. If any token is missing from the translation, it treats the entire result as plain text to avoid dropping markup [22].
- `CloneNode` – Uses simple round-trip XML parsing to clone `XNode` instances safely, supporting common node types (`XElement`, `XCData`, `XText`, `XComment`, `XProcessingInstruction` and `XDocumentType`) [23].

Placeholder Protection (`PlaceholderProtection.cs`)

User interface strings often contain placeholders such as numbered indices (`{0}`), named placeholders (`{User}`), hash-bracket tokens (`#[Shared.Filters]`), percent formats (`%s` or `%1$s`), and shell-style variables (`${VAR}`). The model must not translate or alter these. `PlaceholderProtection` detects such patterns using compiled regular expressions [24] and replaces them with tokens like `__XLF_PH_0__` before translation [25].

- `Protect` – Scans the input string for all supported placeholder patterns, replaces each match with a token, and stores the original placeholder in a list. The method returns the modified string and the list of originals [25].
- `Restore` – After translation, replaces each placeholder token with its original value in order [26].
- `AllTokensPresent` – Verifies that every placeholder token appears in the translated string; this is one of the validation checks used to detect bad translations [27].

XLIFF Translator (`XliffTranslator.cs`)

`XliffTranslator` orchestrates per-file translation. It uses `XliffTokenization` and `PlaceholderProtection` to maintain markup and placeholders, leverages an in-memory cache to avoid repeated translations, and applies heuristics to detect and retry bad translations.

- **Initialization** – Constructed with an `LmStudioClient` and `XliffTranslateOptions`. It also maintains a thread-safe cache (`ConcurrentDictionary`) to map input strings (after tokenization and placeholder protection) to previously translated strings [28].
- **Counting Units** – `CountTransUnits` loads an XLIFF file and counts the number of `<trans-unit>` elements [29].

- **Per-file Translation** – `TranslateFileAsync` loads an input XLIFF file and ensures the `<file>` element's `target-language` attribute matches the desired language [30]. It iterates through each `<trans-unit>`, fetching `<source>` and `<target>` elements. A unit is translated if the target is missing or, optionally, if it is identical to the source; otherwise the unit is skipped [31]. Empty sources are also skipped.

- **Translation Steps**:

- **Tokenize inline tags** – Calls `XliffTokenization.ExtractTextWithTokens` on the `<source>` element to replace child tags with stable tokens [32].
- **Protect placeholders** – Calls `PlaceholderProtection.Protect` on the resulting text to replace placeholders with separate tokens [33].
- **Cache lookup & translation** – If caching is enabled and the tokenized string exists in the cache, the cached translation is used and the cache hit count is incremented. Otherwise the translator calls `TranslateStrictWithValidationAsync`, which wraps the LM Studio translation and validation logic [34].
- **Restore placeholders** – The translator calls `PlaceholderProtection.Restore` to replace placeholder tokens with the originals [35].
- **Rehydrate inline tags** – The `<target>` element is cleared and repopulated by calling `XliffTokenization.RehydrateNodesFromTokens`, which reinserts the original XML nodes in place of the tag tokens [36].

Throughout this process, `XliffTranslateStats` counts translated units, skipped units and cache hits. Progress callbacks report the number of processed and total units to the console [37].

- **Validation and Retries** – `TranslateStrictWithValidationAsync` calls `SafeLmCallAsync` to translate the tokenized text. If the candidate translation appears valid (according to `LooksBad`), it is returned. Otherwise it retries with a lower maximum token count (64 tokens) [38]. If both attempts fail validation, `null` is returned and the original tokenized text is used.
- **Heuristics (`LooksBad`)** – Implements several heuristics to detect poor translations [39]:
- Rejects empty or whitespace-only results [40].
- Ensures all placeholder tokens and tag tokens are present; missing tokens cause rejection [41].
- Searches for prompt leakage or instruction text (e.g., "Translate ONLY", "Preserve tokens", or foreign phrases) and rejects candidates containing such markers [42].

- Rejects outputs that are disproportionately long relative to short inputs (inputs $\leq$ 20 characters producing outputs > 80 characters) [43].

- **Safe Call** – `SafeLmCallAsync` wraps the LM Studio call in a try/catch and returns `null` on exception [44].

- **Whitespace Normalization** – `NormalizeWs` collapses all runs of whitespace to single spaces, used when comparing the source and target strings [45].

Program (`Program.cs`)

The `Program` class contains the entry point and orchestrates the overall process.

- **Language Mapping Functions** – `NormalizeName` converts input language codes or names into friendly names expected by the prompt (e.g., `es` → `Spanish`, `de-DE` → `German`). `NormalizeCode` maps inputs to BCP-47 codes used for the `target-language` attribute [2] [3].
- **Main Method** – Parses command-line arguments and prints usage if insufficient arguments are provided [1]. It resolves absolute paths for input and output directories and ensures the input directory exists [46]. The method sets up a `LmStudioClient` with user-provided or default endpoint and model [47] and constructs `XliffTranslateOptions` (target language name and code, translation rules, temperature and caching) [48]. It enumerates all `.xlf` / `.xliff` files [4], optionally counts the total translation units across files [5], and loops through each file calling `TranslateFileAsync` while reporting per-file progress [6]. A summary of overall processing times and statistics is printed at the end [7].

Project File and Build

The project uses the .NET SDK with the following settings:

- Output type: Console application
- Target framework: .NET 8.0
- Nullable reference types and implicit `using` directives enabled [49]

A Visual Studio solution file is included to allow the project to be opened in IDEs [50].

Extensibility and Configuration Notes

- **Adding new languages** – To support additional language codes or names, extend the `NormalizeName` and `NormalizeCode` switch statements with the appropriate mappings [2] [3].
- **Adjusting translation prompt** – The prompt construction in `LmStudioClient.TranslateStrictAsync` can be modified to include additional instructions or context. Commented lines in the source show an alternative prompt that describes UI labels and placeholder preservation [51].
- **System message** – `Program` currently uses an empty system message. A custom system message can be passed to `LmStudioClient` to influence translation style or instruct the model.
- **Caching and performance** – The in-memory cache prevents duplicate translations of identical segments, which improves speed when many files share the same source strings [34]. For very large translation batches or long-running processes, consider persisting the cache between runs.
- **Validation heuristics** – The `LooksBad` method centralizes translation quality checks. Developers can adjust the heuristics by adding or removing markers or changing length thresholds [52].
- **Integration with other models** – While the tool defaults to LM Studio, any service compatible with the ChatCompletion API can be used by providing its endpoint and model name on the command line. Ensure that the service respects the token placeholders and handles maximum token limits appropriately.

Summary

XliffBatchTranslate provides a pragmatic pipeline for translating XLIFF files in bulk using an LLM. It addresses common issues such as mixed content, placeholder preservation and translation quality through dedicated helper classes and validation heuristics. The CLI exposes a simple interface with sensible defaults while still allowing configuration of model and endpoint. Its modular design (client, tokenization, placeholder protection, translator and CLI) makes it straightforward to extend or integrate into existing localization workflows.

---

1 2 3 4 5 6 7 8 9 46 47 48 Program.cs

https://github.com/djglxxii/XliffBatchTranslate/blob/HEAD/XliffBatchTranslate/Program.cs

10 11 12 13 Models.cs

https://github.com/djglxxii/XliffBatchTranslate/blob/HEAD/XliffBatchTranslate/Models.cs

14 15 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 52 XliffTranslator.cs

https://github.com/djglxxii/XliffBatchTranslate/blob/HEAD/XliffBatchTranslate/XliffTranslator.cs

16 17 18 19 20 51 LMStudioClient.cs

https://github.com/djglxxii/XliffBatchTranslate/blob/HEAD/XliffBatchTranslate/LMStudioClient.cs

21 22 23 XliffTokenization.cs

https://github.com/djglxxii/XliffBatchTranslate/blob/HEAD/XliffBatchTranslate/XliffTokenization.cs

24 25 26 27 PlaceholderProtection.cs

https://github.com/djglxxii/XliffBatchTranslate/blob/HEAD/XliffBatchTranslate/PlaceholderProtection.cs

49 XliffBatchTranslate.csproj

https://github.com/djglxxii/XliffBatchTranslate/blob/HEAD/XliffBatchTranslate/XliffBatchTranslate.csproj

50 XliffBatchTranslate.sln

https://github.com/djglxxii/XliffBatchTranslate/blob/HEAD/XliffBatchTranslate.sln