Author	Comment
Chandima Kulathilake	18th September 2025 – Created using Researcher
Patrick Rousseau	Project ownership and data curation for training

Building a Small Language Model for **Toaripi**

A small language model for Toaripi (East Elema; ISO 639-3: tqo) that generates educational content for primary learners — online and offline.

Toaripi is a low-resource Papuan language (~23,000 speakers) with little digital text available. We aim to train a *small language model (SLM)* using a parallel **English–Toaripi Bible** corpus. Below, we identify comparable models, then outline a technology guide covering model selection, training steps with bilingual data, and offline deployment strategies.

Native Speakers
≈23,000
Toaripi (East Elema) population

Parallel Corpus Size
~31k
Aligned verse pairs (~750k words per language)

Model Size Target
1–7 B
Parameters (small Transformer for efficiency)

Offline Hardware
Raspberry Pi
7B model runs at 4-bit quantization on 8 GB RAM

Typologically Similar Models & Data Sources

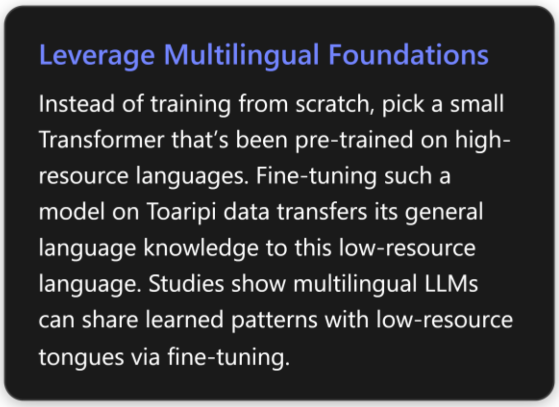
Typological context: Toaripi is a Trans–New Guinea language with likely agglutinative morphology and an oral-centric tradition. No dedicated pre-trained model exists for Toaripi, so we look to analogous low-resource languages and multilingual models:

- **Multilingual LLMs:** Meta's *No Language Left Behind (NLLB)* project trained translation models for 200+ languages (including many with only Bible-based data). For example, NLLB's 1.3B-parameter model learned to translate sparse languages using religious texts. This demonstrates a *typologically diverse model* that can handle languages with complex morphology and limited data – a scenario similar to Toaripi. Another example is the JW300 corpus (Jehovah's Witness text in 300+ languages) used to build broad low-resource NLP models. While these are primarily translation systems, they show that **aligned scripture data can successfully bootstrap models for very low-resource tongues**.
- **Related languages:** There are few closely related languages with NLP resources. However, Papua New Guinea's *Tok Pisin* (an English-based creole) has more data and even open translation models; but typologically it's quite different (simpler

morphology). A more similar structural comparison might be drawn to other morphologically-rich indigenous languages that saw AI efforts – e.g. the *Inuktitut* language (Eskimo–Aleut family) where a text-to-speech AI was built by partnering with the community. Like Toaripi, Inuktitut has complex polysynthetic words and scarce text, and the project leveraged a bilingual corpus with human feedback. The success of such projects indicates that **small models can learn grammatically complex languages if guided by aligned data and community expertise**.

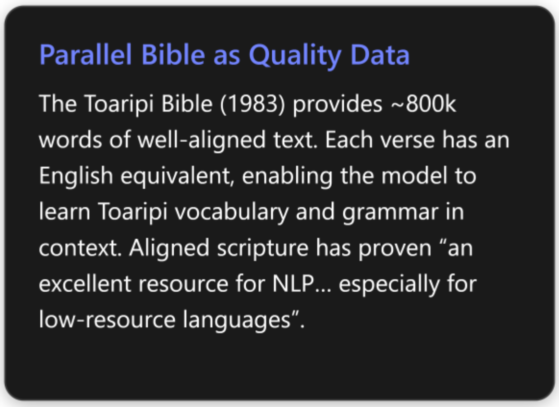
- **Pre-trained multilingual encoders:** Models like **XLM-RoBERTa** or **mBERT** were trained on 100+ languages and cover diverse grammar systems. They likely don't include Toaripi (no Wikipedia), but they *do* capture general cross-lingual patterns that could help if fine-tuned. Note these are **encoder-only** (not text generators). For generation tasks, multilingual sequence-to-sequence models (e.g. mBART or mT5) include many low-resource languages; if Toaripi text were available, such models would be typologically capable of handling its syntax. In practice, since **Toaripi lacks any large corpus**, we focus on choosing a small **general-purpose generative model** as a baseline and fine-tuning it with our parallel Bible data (see next section).

Key takeaway: There is *no off-the-shelf Toaripi model*, but the approach to similar languages has been to leverage **multilingual Transformer models and parallel religious texts**. We will follow that path: start with an open small LM that has learned diverse languages (or at least English) and adapt it to Toaripi.



Leverage Multilingual Foundations

Instead of training from scratch, pick a small Transformer that's been pre-trained on high-resource languages. Fine-tuning such a model on Toaripi data transfers its general language knowledge to this low-resource language. Studies show multilingual LLMs can share learned patterns with low-resource tongues via fine-tuning.



Parallel Bible as Quality Data

The Toaripi Bible (1983) provides ~800k words of well-aligned text. Each verse has an English equivalent, enabling the model to learn Toaripi vocabulary and grammar in context. Aligned scripture has proven "an excellent resource for NLP... especially for low-resource languages".

Candidate Small Models (Open-Source & Offline-Friendly)

We need an **open-source** model that is **small enough** to train and run offline, yet expressive enough for Toaripi. Recent innovations in *small language models (SLMs)* (50M–7B parameters) make this feasible. The table below highlights some suitable model families and their relevance:

Model Family	Params	Architecture	Pre-training	License	Relevance to Toaripi
Meta LLaMA 2	7B	Transformer decoder (GPT-style)	Multilingual text (20+ languages)	*Permissive for research* (some usage limits)	Strong base model with high fluency; 7B version is a top performer in SLM class. Good candidate if usage stays non-commercial.
Mistral 7B	7B	Transformer decoder (improved)	English & mixed data (inherited from LLaMA)	Apache 2.0 (open)	**Recommended.** Latest high-quality 7B model; outperforms older models and is fully open. Ideal if we have the GPU memory to fine-tune it.
StableLM	3B / 7B	Transformer decoder	English web and code (StabilityAI)	CC BY-SA (open)	Smaller variants (3B) can train on modest hardware. Provides a decent base for English, which fine-tuning can extend to Toaripi. May need more tuning for quality vs. larger models.
Vicuna-7B (LLaMA-1)	7B	Transformer (chat-tuned)	Fine-tuned on user chats (multi-turn)	Non-commercial (derivative of LLaMA)	Already conversational. Could be adapted, but original LLaMA-1 license is restrictive. If licensing is a concern, use a fully open equivalent chat model.
XLM-R / mBERT	0.3B (550M)	Transformer encoder (bi-directional)	Multilingual (100+ langs)	Apache 2.0	Not generative (good for classification/embedding). Could use as a feature extractor or to initialize embeddings. But for our goal (text generation) a decoder model is needed.
NLLB-200 (distilled)	1.3B	Seq-to-seq Transformer	Trained on 200 languages (many low-resource)	MIT (open)	A multilingual **translation** model by Meta covering Pacific languages. Useful for translation tasks (English↔Toaripi). We could use it to generate Toaripi drafts by translating English prompts. Not optimized for free-form generation.
OpenLLM (Phi-2)	2.7B	Transformer decoder	English (code and text) by Microsoft	MIT (open)	Very efficient model designed for **edge devices**. Demonstrated to run on Raspberry Pi with 2–3 bit quantization. Could be fine-tuned to Toaripi if high quality on general text is acceptable.

Why these? All the above are small enough to run **locally** with quantization and have **open weights** or allowable licenses. For example, **Mistral-7B** stands out for its state-of-the-art performance among small models and fully open license, making it a prime choice. If 7B is too large for our hardware, a mid-size model like StableLM-3B or the 2.7B Phi-2 (OpenLM) can be alternatives that still capture a lot of language patterns at lower compute cost. We avoid purely monolingual small models that are trained only on English text (they may lack diversity in structure); however, even an English-trained GPT-style model will learn Toaripi well through fine-tuning because the **shared vocabulary and syntax learning in any language model transfers to new languages** to some extent.

> **Decision:** We recommend fine-tuning a **multilingual Transformer in the ~1B–7B range**, leaning towards the upper end if resources allow. Larger SLMs (6–7B) tend to produce more fluent and coherent text, which is important for generating natural-sounding Toaripi sentences. For instance, using **Mistral-7B** as the base model would likely yield the best Toaripi output (it has strong baseline performance and was released explicitly for open use). If GPU memory is a bottleneck, a ~3B parameter model can be used with some quality trade-off. The key is that the model be **pre-trained on enough general language data** – which all of the above are – so it doesn't start from zero.

Preparing & Training on the English–Toaripi Corpus

Parallel data: The Toaripi Bible provides *verse-aligned parallel text*. Each verse in Toaripi has a corresponding English verse with the same meaning and verse ID. This structure is extremely helpful:

- It offers thousands of **sentence pairs** that are translations of each other. The model can learn that "word X in Toaripi = word Y in English" and overall semantic alignment. Such bilingual signal will impart Toaripi vocabulary while grounding it in English semantics.
- The Bible's text is well-edited and consistent in style, giving a solid grammatical example of Toaripi (albeit a formal style). We may need to mitigate archaic or overly formal tones (see below), but as a training backbone, scripture text is high-quality (no slang, very few errors). In low-resource NLP, *religious texts have often been a key resource* – e.g. Christodoulopoulos & Steedman (2014) aligned 100 Bibles and successfully projected linguistic structures to many languages.

Data preprocessing: We will extract the verses and create a tabular dataset, for example:

```
VerseID | English verse (e.g. KJV)         | Toaripi verse (1983
translation)
John 3:16 | "For God so loved the world..." | "Ambo Atua ... ara
hari evetai..."
```

Steps to prepare data:

1. **Acquire text**: Ideally, obtain digital text from a licensed source. *ScriptureEarth* or the Bible Society might provide the Toaripi text. If not, we can scrape via an API like YouVersion (which lists Toaripi *TQO1983*). For English, use a public-domain version (e.g. **World English Bible** or **KJV**) to avoid copyright issues.
2. **Align verses**: Both Toaripi and English Bibles use standard book/chapter/verse indexing. We ensure verses line up by those indices. (Occasionally, translations merge or split verses; our script will detect missing numbers and adjust accordingly, e.g., skip a verse if one side doesn't have it). Overall, alignment is straightforward thanks to the numbering system – "every verse is uniquely identified, simplifying alignment".
3. **Cleaning**: Remove any markup, footnotes, or verse headers. The text should be plain sentences. Also normalize punctuation (e.g., ensure both versions use the same basic punctuation set so that alignment isn't thrown off by fancy quotes or m-dashes). The Toaripi text uses Latin script (some diacritics), which is fine for modern tokenizers but we'll ensure they're preserved correctly (e.g., "á" in **"Áre"**).
4. **Split data**: We might reserve a small portion of verses as a **validation set** to monitor training (e.g., keep aside 5% of verses to check model performance on unseen pairs). This can help detect overfitting.

Training strategy: We have a few options to fine-tune the model on this data:

- **Translation Fine-Tuning:** Formulate it as a translation task. E.g., feed the model an English sentence and train it to output the Toaripi translation (and possibly vice-versa). This *cross-lingual* training teaches the model to map meanings between English and Toaripi. Even a few epochs of this will strongly imbue the model with Toaripi vocabulary and grammar aligned with English. After this, we could prompt the model in English and have it generate Toaripi, effectively using it as a translator when needed. **Why translation?** It leverages the parallel nature of data; and large LMs have shown they can learn translation with surprisingly little data because the task is well-defined. For example, researchers found that even a handful of example pairs can induce translation ability in GPT models via fine-tuning or prompting. Here we have thousands of pairs – plenty to fine-tune a bilingual model.
- **Direct Language Modeling:** Alternatively, convert the corpus into a Toaripi monolingual set and do standard LM fine-tuning (predict next word in Toaripi sequences). We could simply concatenate Toaripi verses as training text. However, *danger:* the model might overfit to Bible style and just memorize verses. To mitigate

that, one approach is *shuffled verses* or adding variety: for instance, intermix English and Toaripi verses in training with language tags (so it's a multilingual LM). But since our aim is *generation of new content*, not just mirroring scripture, pure LM on Bible text may limit stylistic range.

- **Combined approach:** A promising sequence noted in research is: **(1)** fine-tune for **English↔Toaripi translation**, then **(2)** use the model to **generate new Toaripi data** (e.g. translate some simple English children's stories or educational prompts into Toaripi), and **(3)** fine-tune again on this *augmented Toaripi dataset* to make the model a Toaripi prose generator. Essentially, we *bootstrap* additional training data via the model itself (a form of **back-translation** augmentation). For example, we can take a set of English sentences about daily life (which the Bible doesn't cover) and have our intermediate model translate them to Toaripi – producing synthetic Toaripi sentences that aren't from the Bible. Then include those in a second round of training. This helps the model adapt to more casual or modern topics (e.g., "school", "football", etc.) that the Bible corpus lacks.

In practice, we will likely do an **initial fine-tune that mixes translation and generation objectives**. Concretely, we can set up training examples where sometimes the model is given English and asked for Toaripi, and other times given Toaripi and asked to continue in Toaripi. Recent low-resource NLP techniques suggest that *multitask training* (both translation and language modeling) can be beneficial, as it teaches both alignment and fluency.

Avoiding biblical tone: The Bible has a formal tone and antiquated vocabulary. Our model should be able to "code-switch" to a simpler, child-friendly style. Strategies to achieve this:

- Emphasize narrative verses over genealogies or very archaic constructions during training (the Bible has stories, dialogues, and also lists/rules – the former are closer to everyday language). We might put more weight on narrative parts.
- Use **prompting techniques** at generation time (discussed later) to instruct style, e.g. "Use simple modern language" etc.. Because our final model will likely respond to instructions, we can guide its tone through the prompt.
- If possible, fine-tune on any *non-Bible Toaripi text* we can find. For example, if there are folk tales or word lists collected by linguists, adding those would broaden the style. (Even a small amount of such data, if available, could help diversify the model's output vocabulary beyond strictly biblical terms.)

Tokenization considerations: Our base model will have its own tokenizer (e.g. SentencePiece for LLaMA/Mistral). It might not "know" certain Toaripi words, leading to awkward subword splits. We can extend the tokenizer: for instance, add a custom vocab of frequent Toaripi character sequences. Toaripi uses standard Latin letters plus perhaps diacritic marks, so encoding isn't a big issue (no new script). But a word like *"hari evetai"* might be broken into small pieces if not in the original vocab. We can use the parallel corpus to identify the most common Toaripi words and ensure the tokenizer merges them. Many modern models allow adding tokens or using byte-fallback; we can also rely on bytes (UTF-8) if needed. In summary, we will **audit the tokenizer output** on some sample Toaripi text: if we see overly fragmented outputs, we'll train a custom tokenizer or use byte-level modeling. Proper tokenization can improve fluency and reduce training difficulty.

Regularization: Given only ~30k sentence pairs, we will train with caution to prevent overfitting. Techniques: use a modest learning rate, apply *early stopping* if validation loss stops improving, and possibly use dropout on the model. We can also slightly perturb input sentences (e.g. random replace a word with itself 50% of time) to make the model work harder – known as noise injection. But with a pre-trained base, overfitting is less of a risk than if we were training from scratch, because the model has millions of general patterns "baked in." Still, we'll monitor performance on held-out verses.

Human evaluation: Because automatic metrics (like BLEU for translation or perplexity for LM) won't fully capture quality of Toaripi output, we plan to involve fluent speakers for evaluation. For example, after training, have a Toaripi speaker review a set of model-generated sentences or translations for grammaticality and naturalness. This feedback can guide further fine-tuning. Community input is *critical* – as seen in Microsoft's Inuktitut project, **engaging native speakers to review and correct the AI's output greatly improved the end quality**. We anticipate doing a small round of such "human in the loop" refinement: e.g., if certain frequently used terms sound too literal or formal, we adjust and perhaps fine-tune on those corrections (a lightweight form of RLHF).

By the end of training, we expect the model to **understand Toaripi** and produce it reasonably well. The bilingual training will ensure it doesn't hallucinate completely incorrect grammar – it will base its Toaripi on real examples. The next step is making sure this model can be *used in practice*, especially in under-resourced settings.

Deployment for Offline Use

A core requirement is that the model runs **offline** or in low-connectivity environments. Many Toaripi-speaking villages have limited internet and only basic hardware. We therefore plan for an **edge deployment**:

- Use **model compression** to shrink the final model for inference.
- Support running on a typical *affordable laptop* or even a *Raspberry Pi*-class device.

Model quantization: After fine-tuning, the model (e.g. 7B parameters) can be quantized from 16-bit to 8-bit or 4-bit integers. This drastically reduces memory footprint (a 7B model ~14 GB in float32 can drop to ~4 GB in int4). The slight loss in precision usually has minimal impact on generation quality, especially for high-level content. We will use proven tools like **llama.cpp** which implements 4-bit and even 3-bit inference for LLaMA-type models. In fact, community tests show 4-bit quantized LMs retain *most* of their accuracy. We'll verify that the Toaripi outputs remain good after quantization (they should).

Optimized runtime: Instead of Python, we'll run the quantized model with a lightweight C++ engine (e.g. llama.cpp or similar). This leverages CPU vector instructions (AVX on x86, NEON on ARM) to speed up inference. For example, *llama.cpp* on a desktop CPU can generate a few tokens per second with a 7B model at 4-bit, which is acceptable for short paragraphs. On a Raspberry Pi 4 or 5, llama.cpp can run a 3B model with ~4 seconds per word at 4-bit, according to one project that got a 1.1B model responding on a Pi 4. New libraries like **picoLLM** by Picovoice further optimize this, showing even a 2.7B model running *faster than realtime* on a Pi 4 by using 2–3 bit quantization and efficient threading. We will explore these options (the end goal: **maximize tokens/second on minimal hardware**).

Hardware expectations:

- *Raspberry Pi 4/5 (4–8 GB RAM):* These can handle models up to ~3B parameters in 4-bit. A Pi 5 with 8GB was able to load a 4-bit 3B model (~3GB) and generate short sentences with llama.cpp. A 7B model might be *just* beyond a Pi's capacity unless we further reduce precision or accept heavy swap to SD (which would be slow). If we insist on 7B on Pi, the newest Pi 5 with 8GB plus a fast SSD for swap could possibly do it, but performance would suffer. For practical use in villages, a 2–3B model might be the sweet spot for RPi deployment (faster and fits comfortably).
- *Mobile Android device:* Though not our first target, many communities have Android phones. A 1B model can run on a modern phone with int8 quantization, albeit slowly (~1 token/sec). There are frameworks (TensorFlow Lite, NNAPI) to deploy models on phones. If needed, we could distribute a simplified model (maybe 500M-1B parameters) as an .apk for an offline app. But due to speed and complexity constraints, we prioritize Pi/PC for now.
- *Laptop/desktop:* An ordinary laptop with 8GB+ RAM can run a 4-bit 7B model. For example, an Intel i5 with 4 cores can generate a short paragraph in ~30 seconds with a quantized 7B. That is acceptable for a teacher preparing materials (not instantaneous, but usable). If the laptop has a GPU, even better – but we assume worst-case (CPU only). We will ensure the deployment can use multi-threading to utilize all cores for a speed boost.

The table below summarizes deployment considerations:

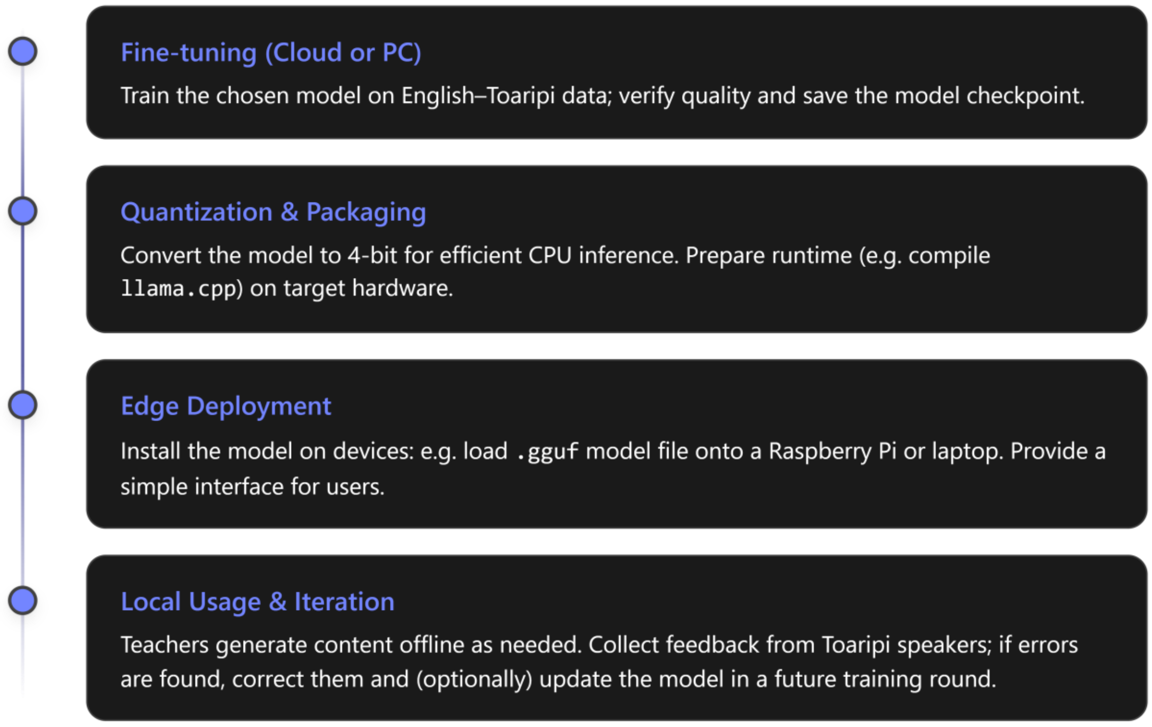
Deployment Option	Description & Tools	Pros	Cons / Trade-offs
Local offline (on-device) ***e.g. Pi or laptop***	Run model on local hardware. Use 4-bit quantized weights with **llama.cpp** or similar. No Internet needed.	- Full autonomy: works **anywhere, anytime**- Privacy: data stays local- Empowers community to use AI freely without connectivity limits.	- Speed limited by device CPU (long outputs are slow)- Must distribute model files to devices (several GB) - Requires some tech setup (we can pre-load SD cards with everything).
Central offline hub ***e.g. one PC at school***	A "hub" device (more powerful PC at a central location) runs the model and produces content (possibly in batch). That output is then **shared** to others via print or USB.	- Only one system to maintain/upgrade (easier support)- Can generate lots of content under expert supervision, ensuring correctness before distribution.	- Not interactive for each teacher in real time- Requires a distribution channel (someone needs to regularly deliver the generated materials to village teachers).
Cloud service (online)	Host the model (or a larger one) on a server (Azure VM or web API). Users connect via a web interface.	- Could use a bigger model (for even better quality) since running on cloud GPU.- No local compute needed; any phone with internet can use it.- Easy to update centrally (improve model or app anytime).	- Internet **required**, which many target schools lack.- Ongoing cloud costs (not sustainable without funding).- Less control for community if funding or connectivity fails.
Mobile app (offline)	Package a smaller model inside a mobile app using ONNX/TFLite. Everything runs on the phone.	- Highly portable: each teacher can have the app in hand.- No network needed after downloading app.- Leverages existing smartphones (if available).	- Complex to implement and optimize (mobile inference is tricky).- Limited by phone hardware: likely need a very small model (which might degrade output quality).- Could drain battery during long usage.

Based on these considerations, we favor a **hybrid offline approach**: have the model runnable on local devices (so no reliance on internet), but also use some centralized preparation for efficiency. For example, a tech-savvy volunteer at the district level could run the model on a decent PC to generate a "stockpile" of Toaripi exercises and stories, which are then printed or saved to memory cards for each school (*centralized offline* use). Meanwhile, a Raspberry Pi at a school could allow teachers to do on-the-fly generation for their specific needs (*local offline* use). This way, teachers aren't completely dependent on their own device's slow inference for every task – they have some pre-made content – but they also have the capability to create new material locally if needed.

Hardware requirements for fine-tuning: It's worth noting that training (fine-tuning) the model is separate from deployment. Fine-tuning a 1B model can be done on a single modern GPU (e.g. 16 GB VRAM) or even CPU (slowly). Fine-tuning a 7B model typically needs at least one high-end GPU (e.g., an NVIDIA A100 or RTX 3090) or else multi-GPU with smaller cards. This process is a one-time effort; we can perform it on Azure cloud VMs or a research cluster and then focus on deployment. Training time is not critical (if it takes a day or two, that's fine). The end users will never need to retrain the model themselves – they'll use the pre-trained checkpoint.

Deployment example: We will provide a simple CLI and perhaps a minimal web UI to interact with the model. For instance, a teacher could open a laptop (or connect to a Pi's local Wi-Fi hotspot) and access a small web page where they input a prompt ("List five animals in Toaripi with their English meanings") and the model outputs the result. This web UI would be served by the device itself (no internet). Under the hood, it calls the quantized model. We'll ensure the interface is lightweight (possibly a text-based UI or basic HTML) so it runs even on a Pi.

Finally, we will document how to set all this up in a **Deployment Guide** for non-technical users, and likely prepare pre-configured SD card images for Raspberry Pi that have everything installed. This way, an educator can simply plug in the Pi and turn it on to use the model offline.


Fine-tuning (Cloud or PC)
Train the chosen model on English–Toaripi data; verify quality and save the model checkpoint.

Quantization & Packaging
Convert the model to 4-bit for efficient CPU inference. Prepare runtime (e.g. compile `llama.cpp`) on target hardware.

Edge Deployment
Install the model on devices: e.g. load `.gguf` model file onto a Raspberry Pi or laptop. Provide a simple interface for users.

Local Usage & Iteration
Teachers generate content offline as needed. Collect feedback from Toaripi speakers; if errors are found, correct them and (optionally) update the model in a future training round.

Result: By following this plan – choosing the right small model, fine-tuning with parallel Bible text, and deploying with quantization – we will create a practical Toaripi language model. It will be capable of producing new sentences and educational materials in Toaripi, **powered by transfer learning from English and delivered in an offline-friendly manner**. This approach balances cutting-edge NLP techniques with the reality of rural tech infrastructure, and it aligns with our goals of language preservation and educational impact.

Appendix

References:

- Christodoulopoulos, C. & Steedman, M. (2014). *A massively parallel corpus: the Bible in 100 languages.* LREC.[5][5]
- Agić, Ž. & Vulić, I. (2019). *JW300: A wide-coverage parallel corpus for low-resource languages.* ACL.
- **Microsoft Community** – Indigenous Languages: *Inuktitut Text-to-Speech collaboration* (2024)[12][12]; *Te Reo Māori AI Project* (2025)[11][11].
- Dextralabs (2025). *Top 15 Small Language Models for 2025*[2][2].
- Picovoice (2024). *Running LLM on Raspberry Pi with picoLLM*[9][9].
- UGA Vision Paper (2024). *Foundation Models for Low-Resource Language Education*[4][4].

(Additional inline citations are included in the text above to specific sources and tool documentation for factual statements.)

References

[1] SLlama: A Small Language Model for Extremely Low Resource Domains

[2] 15 Best Small Language Models [SLMs] in 2025 | Dextralabs

[3] Mistral 7B | Mistral AI

[4] Foundation Models for Low-Resource Language Education (Vision Paper)

[5] A massively parallel corpus: the Bible in 100 languages

[6] Government of Nunavut introduces Inuktitut text-to-speech functionality ...

[7] The Bible in the Toaripi Language Gulf Province Papua New Guinea

[8] Easy way to run speedy Small Language Models on a Raspberry Pi

[9] Local LLM on Raspberry Pi using picoLLM| Picovoice