Building Sovereign AI Models

Technical Overview

Table of Contents

Chapter 1: Introduction

Imagine a nation that holds the keys to its own digital future — where its people, data, and institutions work together to create intelligence that understands and reflects its language, culture, history, and needs. This is the vision of Sovereign AI.

Sovereign AI refers to a country's capacity to develop and deploy artificial intelligence using its own infrastructure, local data, skilled talent, and business ecosystems. When a country builds its own AI capabilities, it's not just adopting technology; it's defining how that technology serves its people.

A key aspect of Sovereign AI is the development of Sovereign LLMs — large language models trained on a nation's own data, languages, and cultural context. These models empower countries to embed their unique identity, values, and priorities into the AI systems that drive innovation and shape their future. To advance sovereign AI development, NVIDIA publishes NVIDIA Nemotron, a family of highly open and efficient models that think fast and deliver the highest accuracy for agentic AI. NVIDIA not only releases the models, but also the training datasets, techniques, and learnings, enabling other model builders to adopt various parts of Nemotron to create their own sovereign AI models.

In this guide, we discuss the technical elements of building a Sovereign Large Language Model (LLM) in depth. The development of **Sovereign AI (including Sovereign LLMs)** relies on four key technical pillars: **Data and benchmarks**, **Models**, **Hardware Infrastructure**, and **Software Frameworks**. Each pillar plays a critical role in helping countries develop their sovereign AI capabilities and resources.

Let's have a closer look at each of these four technical pillars, and how NVIDIA technologies fit into the big picture:

- **Data and benchmarks** are of critical importance, and the most practical and urgent place to start. Countries, local enterprises and communities should start with a strategy to curate and secure high-quality local datasets that will help train or customize generations of advanced AI models to come. NVIDIA NeMo™ Curator offers capabilities to process and curate high-quality data from the web or synthesize data in multiple languages.

  In addition to a data strategy, there should be a benchmarking strategy focused not just on measures of intended task accuracy but encompassing key elements of language, history, geography, culture, and the law. NVIDIA NeMo Evaluator is designed to simplify end-to-end evaluation of generative AI applications.

Benchmarks also help us design safety into the model, but when the model is deployed (inference-time) active monitoring of the inputs and outputs of generative AI models is critical. NVIDIA NeMo Guardrails offer a structured way to develop and incorporate the active safety layer into your Sovereign AI deployment.

- **Models** can either be trained from scratch, or finetuned from a high-performing open model. For complete sovereignty, the former is preferred; however, this ultimately depends on the quantity and quality of locally curated data. High-quality commercially permissible open models offer a middle ground. Model builders across the globe are adopting Nemotron models, datasets and techniques to build their sovereign AI models. For example, European countries including France, Italy, Poland, Spain and Sweden are delivering sovereign models with Nemotron.

- **Hardware infrastructure**, either on-prem or in the cloud, should physically reside within the country's borders with robust governance and security protocols. Storing the data in domestic infrastructure can help ensure sensitive data is protected. NVIDIA AI infrastructure based on NVIDIA Hopper and Blackwell GPU architectures and Infiniband networking is the industry standard when it comes to training and inferencing large-scale AI models.

  Building and designing an AI factory is complex, requiring a strategic layout of thousands of servers, cooling systems, and reliable power networks to operate seamlessly and to avoid downtime. The NVIDIA Omniverse Blueprint for AI Factory Digital Twins, powered by NVIDIA Omniverse™ and Universal Scene Description (OpenUSD), provides a unified digital twin where all aspects of an AI factory can be designed, simulated, and optimized together before construction begins.

- **Software Frameworks** are the components that bind all the above pillars together. Training frameworks enable model creation, while AI inference frameworks put AI systems into everyday use. NVIDIA NeMo is an end-to-end platform for developing custom and sovereign generative AI, anywhere. NeMo enables enterprise-ready models with high-quality data curation, accelerated customization, retrieval-augmented generation (RAG), and model guardrails. NVIDIA NIM is an evolving set of easy-to-use microservices designed for secure, and reliable deployment of high-performance AI model inferencing across clouds, data centers, and workstations. NVIDIA Dynamo is a high-throughput, low-latency inference framework designed for serving generative AI and reasoning models in multi-node distributed environments. NVIDIA Dynamo is designed to be inference engine agnostic (supports SGLang, TensorRT-LLM, vLLM or others) and captures LLM-specific capabilities such as disaggregated prefill and decode inference, dynamic GPU scheduling, LLM-aware request routing, accelerated data transfer and KV cache

offloading.

Since data and models are the basis for establishing sovereign AI, the subsequent sections cover details on data curation, data cleaning and processing, and the steps involved from model sourcing to model deployment.

Chapter 2: Data Curation

The most important aspect of any machine learning system is the underlying data that is used for building that system as well as evaluating its functionality. Consequently, the performance of the entire system hinges on the quantity as well as the quality of the data that was used to train and evaluate it. Broadly speaking, collecting, curating and quality assurance of the data requires the most amount of time and financial resources to ensure that the final product functions within the required specifications.

When starting the sovereign AI journey, the importance of data curation cannot be overstated. Considering that the primary mode of communication in LLM systems is through language, the data must be “high-quality” and comprehensive, which means that it must encapsulate the cultural heritage, knowledge, norms and practices so that the AI system can accurately represent the national and societal values. Fortunately, once high-quality data is collected and curated, it can be re-used many times for many generations of AI systems, making the initial resource commitments a lasting investment.

While the Internet contains vast amounts of language data that modern AI systems use for training and evaluation, the most commonly available language data is in English. As such, an immediate challenge that often arises when building AI systems for local communities is the shortage of target language-specific data. For example, if the intention is to support multiple dialects of the same language, collecting sufficient data that correctly reflects the unique aspects of each dialect may be impractical. We refer to this phenomenon as working with either high-resource or low-resource languages. For instance, English is generally considered a high-resource language given that it is readily available in many forms, whereas Hindi might be considered a lower-resource language compared to English.

In this section, we outline strategies that are often used for acquiring low-resource language data, and the types of data that are relevant for model training. The following topics are covered:

- Types of data to collect
- Approaches to data acquisition
- Data cleaning and processing

Types of Data to Collect

The training data needed for localized LLMs can be broadly categorized into two types: pretraining data and alignment data. Usually, model training starts with pretraining and progresses through alignment. We provide a brief overview of the role of each type of data, then discuss approaches for curating and processing them.

Pretraining Data

This refers to the data that is needed to either train an LLM from scratch (pretraining) or leverage cross-lingual transfer learning to adapt an existing model trained on a high-resource language to a low-resource one (continued pretraining). The primary objective of this effort is to collect exemplary data that encompass diverse vocabulary and language grammar that span across various word and sentence structures, as well as diverse localized or cultural knowledge that are relevant for the model to learn from. In addition to the quality, the quantity of the data is very important at this stage.

Alignment Data

Upon the creation of an LLM that has sufficient understanding of the language structure, vocabulary and general knowledge, it is necessary to instill conversational abilities into the model (such as question answering and general helpfulness), and also align it with human values or culturally accepted norms and traditions (such as preventing the generation of unsafe, incorrect, or harmful content). At this stage, the quality and the consistency of the data is more important than the quantity. This is because the goal is for the model to learn about nuances of human preferences, rather than general knowledge and linguistic capabilities.

Broadly speaking, alignment data consist of two main types: instruction tuning data and preference data.

Instruction tuning data is used to teach the model how to respond to queries and follow prompts, or how to perform tasks in a way that aligns with the user's intent. The focus is on ensuring that the model can produce contextually relevant responses or provide clarifications to user questions. Preference data, on the other hand, aims to provide feedback to the model regarding human preferences, cultural norms and accepted behaviors. The focus is on ensuring that the model adheres to ethical and safety standards.

Approaches to Data Acquisition

We now turn our attention to data curation and processing approaches that are relevant to the broad types of data we just discussed, drawing inspiration from prior work, which leverages the NVIDIA NeMo Curator towards customizing models for non-English languages.

Internet Archives and Existing Corpora

Many open-source initiatives, research institutions, or other public efforts have compiled and maintained publicly available multilingual datasets that can be accessed freely. Examples include Common Crawl and Wikipedia. Given their size, these datasets can be a great first step towards creating the corpora needed for starting the model pretraining process. NeMo Curator provides example utilities for downloading and extracting Common Crawl, ArXiv, and Wikipedia data. In addition, it provides a flexible interface to extend the utility to other datasets.

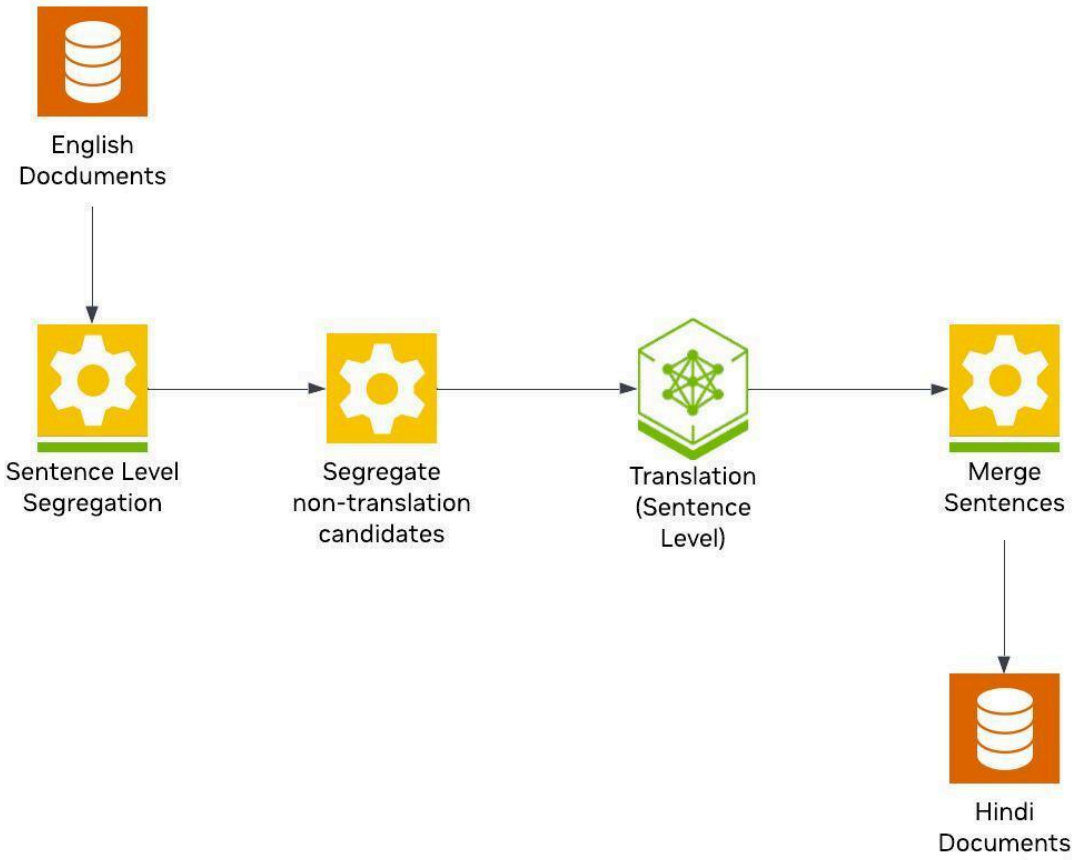
NVIDIA recently published a blog on curating high-quality Vietnamese language data using NeMo Curator, where multiple data sources were used. Specifically, Wikipedia's Vietnamese articles, the Vietnamese subsets of the C4 and OSCAR datasets, as well as a Vietnamese news corpus containing local news articles were combined together to form the initial corpora, which was subsequently processed and deduplicated for quality assurance.

Synthetic Data Generation: Translation and Transliteration

While using publicly available datasets is a great starting point, the amount of data can still be quite limited. As such, approaches such as translation and transliteration from high resource languages have gained popularity. The core idea of these approaches is to take data from high-resource languages such as English, and adapt them to low-resource languages.

The figure below demonstrates a translation pipeline made using NeMo Curator, which leverages a sentence translator for translating documents from English to Hindi.

Figure 1: A translation pipeline using NeMo Curator



Other commonly used approaches for synthetic data generation include back-translation and transliteration. Back-translation involves translating the text from the low-resource language to a high-resource language and back, which can alter sentence structure without affecting the semantics. Transliteration involves converting the text from low-resource languages' writing system to another one (such as writing Hindi words in the Roman alphabet instead of the Devanagari script). This approach can leverage the token representation of an existing LLM that was previously trained on a high-resource language towards understanding the semantic meaning of the low-resource language when used with a different writing system.

NVIDIA recently published a paper describing the details of creating Nemotron-4-Mini-Hindi-4B, a small language model for Hindi, which is India's most prevalent language. In addition to Hindi data scraped from public Internet sources, this work leveraged the translation of English documents into Hindi to further increase the pretraining dataset size, by as much as 60 billion tokens. The authors demonstrated the efficacy of this approach by evaluating the model on a variety of Hindi benchmarks. Follow-up work systematically studied the translation strategies for the alignment data and showed that LLM-based selective translation is more effective than traditional

methods because it better preserves critical non-translatable elements like code and structured data formats.

Furthermore, Nemo Curator offers 10+ scalable prebuilt synthetic data generation pipelines using OpenAI-compatible APIs for Supervised-Fine-Tuning (SFT), preference data and pretraining data.

One major drawback of creating synthetic data is the possibility of introducing noisy samples into the dataset. An example of noise generated during the translation process includes repetitive text and translation of undesirable elements or unknown vocabulary (e.g. math and coding blocks, or technical words). This necessitates filtering out noisy samples before starting training. Common approaches include using n-gram statistical filtering, or back-translation methods to determine if the synthetic data meets the quality criteria.

To get diverse data from both low- and high-quality data, we developed a solution based on synthetic data generation while building the Nemotron-CC dataset. We use SDG pipelines to generate training data from both low- and high-quality documents. For low-quality documents, we repurpose useful information by prompting a large language model to rewrite the text in a clear, Wikipedia-style format. For high-quality data, the goal is to produce more unique and informative pretraining tokens by rephrasing or condensing essential knowledge.

To achieve this, we ran four different LLMs with tailored prompts to generate diverse question-answer pairs in various formats.

Some of these formats included:

- Yes/no, multiple-choice, and open-ended questions
- Distilled versions of the text that are more concise and clear
- Rewritten passages that focus solely on meaningful content while filtering out noise
- Structured lists that capture the key knowledge from the source text

Crowdsourcing and Other Community-Based Efforts

The approaches outlined thus far are mainly geared towards zero- or low-cost methods of acquiring existing low-resource language data and are primarily suitable for continued pretraining.

For alignment data, on the other hand, it is often necessary to allocate resources towards creating and labeling high-quality data that can ensure the safety and the

reliability of the system. Popular approaches include crowdsourcing or partnerships with local organizations and linguists. This way, the adherence of the data to local norms and regulations can be more easily controlled and certified.

Data Cleaning and Processing

Once the data is acquired, they require processing before usage for training purposes. This is because the acquired documents, especially those when scraped from the web, might demonstrate the following characteristics that might make them unfit for training without further processing:

- Duplicate text where certain documents or text are repeated over and over
- Irrelevant data, for example, documents in a non-target language, or data from topics unrelated to the operation domain of the AI system
- Non-uniform character representations
- Personally identifiable information, such as names, email addresses, or other personal information tied to real people

Considering that the acquired data may come from various sources, it is also important to blend the data properly to ensure the final dataset contains a reasonable mix of all representative documents that are intended to be used for model training.

We will now discuss the importance of each of these processing steps and highlight the facilities that NeMo Curator provides for each stage. For a more comprehensive overview of these operations, please refer to our blog post that delves deeper into this topic.

Unicode Unification

Standardizing text data and eliminating encoding errors, common in web-crawled datasets, is an important first step in any data processing pipeline. NeMo Curator uses open source libraries such as ftfy: fixes text for you to detect and resolve character representation issues in the datasets in a scalable manner.

Language Identification

To ensure the acquired data only contains the desired languages, it is important to identify the languages represented in the acquired documents. NeMo Curator provides a predefined set of heuristics to identify the language of a given set of documents.

Parallel Data Curation

Parallel data curation is one of the most effective strategies for training multilingual models, as it helps them understand direct cross-lingual relationships and improves performance on translation, speech, and understanding tasks. Models learn best when trained on well-aligned examples that convey the same meaning across languages.

NeMo Curator streamlines this process with a GPU accelerated pipeline that filters out noise and misaligned examples using semantic similarity (LaBSE), length ratio, character histograms, and quality estimation. This approach efficiently removes low-quality or hallucinated data, even from synthetic sources and scales to massive datasets, filtering the MOSEL dataset across 8 nodes with 8 A100 GPUs in just 47 minutes. The result is cleaner data for more accurate and robust multilingual or speech-to-speech models.

Deduplication

Another important processing stage is the removal of identical, or near-identical documents to reduce data redundancy. This stage can significantly increase the quality of the data, prompting the training of more accurate models, and reduces the data size, which lowers the storage and computation costs of working with large-scale datasets.

NeMo Curator provides exact deduplication, fuzzy deduplication and semantic deduplication modules to aid with this stage. Exact deduplication involves removing duplicates that are exactly identical, for instance when the same document was scraped multiple times from the web. Fuzzy deduplication involves finding documents that contain similar subsets of text. Semantic deduplication takes this processing one step further by analyzing the documents at a semantic level and marking documents that contain semantically similar information. These functionalities are accelerated using RAPIDS cuDF, cuML, and crossfit.

Filtering

NeMo Curator delivers scalable and customizable data filtering capabilities to enhance dataset quality by removing irrelevant or low-quality documents based on diverse criteria such as duplicates, language, domain, and content heuristics. It supports both heuristic-based filtering—using statistics like length, punctuation, and repetitiveness—and classifier-based filtering aligned with high-quality reference data. With over 30 built-in filters for English, non-English, and code datasets, NeMo Curator also enables users to gather detailed statistics and metadata, offering a flexible interface to create and combine custom filters for optimal downstream model training.

Model-based Classification

NeMo Curator enables efficient, distributed classification of large text datasets using pretrained models across multi-node, multi-GPU clusters.

Classifier models are specialized in categorizing data into predefined groups or classes, playing a crucial role in optimizing data processing pipelines for fine-tuning and pretraining generative AI models. Their value lies in enhancing data quality by filtering out low-quality or toxic data, ensuring only clean and relevant information feeds downstream processes.

Beyond filtering, classifier models add value through data enrichment, annotating data with metadata like domain, type, or content specifics and creative quality-specific blends.

The NeMo Curator team has released the following classifier models:

- Domain Classifier: A text classification model to classify documents into one of 26 domain classes.
- Quality Classifier DeBERTa: A text classification model that classifies documents into one of three classes (High, Medium, or Low) based on the quality of the document.
- Prompt Task and Complexity Classifier: A multiheaded model that classifies English text prompts across 11 task types such as Open QA, Chatbot, and Text Generation, as well as six complexity dimensions, including Creativity, Domain Knowledge, and Reasoning. Developers can use this model for tasks such as prompt routing and understanding user prompts.
- Instruction Data Guard: A deep learning classification model that helps identify LLM poisoning attacks in datasets, generates a score, and predicts whether the input data is benign or poisonous.
- Multilingual Domain Classifier: A multilingual text classification model that categorizes content in 52 languages, including English, Chinese, Arabic, Spanish, and Hindi, across 26 domains such as Arts, Business, Science, and Technology.
- Content Type Classifier DeBERTa: A text classification model designed to categorize documents into one of 11 distinct speech types based on their content, such as Blogs, News, and Reviews.
- FineWeb Mixtral Edu Classifier & FineWeb Nemotron-4 Edu Classifier: Text classification models designed to determine the educational value of a piece of text (score 0-5 from low to high).

Customization for New Use-cases

In addition to the host of processings that are supported out of the box and described here, every module in NeMo Curator can easily be customized for domain- or business-specific use-cases. This customization can be achieved by leveraging the Python API of NeMo Curator. The usage of external tooling, such as custom domain or quality classifiers is also supported.

Chapter 3: Model Sourcing

There are two main starting points for building your own AI model:

- Pretraining a model from scratch.
- Starting from a previously trained LLM, and further training it on your "Sovereign" and trusted data sources. This can take the form of continuing the pretraining, and/or aligning the model to closely reflect human intentions and values using one or more of model alignment techniques. Starting from an Open Source Foundation is briefly discussed.

When using a previously trained LLM, which can offer significant benefits in terms of providing known capabilities and allow faster fine tuning with less compute, there are trade-offs. An AI model is a reflection of the data it is trained on. Data sources and blends used for training frontier LLMs are often considered trade secrets, even for models that are open-sourced for wide or restricted community use. As a result, while model cards and research papers may provide some insight into the datasets used, the level of detail is often insufficient to establish clear lineage and accountability for the model's outputs. This lack of transparency can be particularly problematic when deploying AI in mission critical or sensitive use cases. Consequently, the choice between the two options above hinges on the need for fine-grained control over the data sources that inform the model.

Model sizing is also an important consideration. The following related topics are briefly explained later:

- Task complexity
- Deployment cost considerations

Pretraining from Scratch

This allows a high degree of control over data sources and architecture. While there are approaches to remove or edit information (such as "Model Unlearning") from a pretrained starting point, building from scratch ensures that LLMs are trained on data that is relevant, and aligned with their interests and values from the beginning.

UAE's Falcon series are notable examples of models that were built from scratch. At the time of launch, Falcon 40B was one of the most capable models in the community. Other notable initiatives include Japan's Fugaku-LLM, Netherlands GPT-NL, and India's Krutrim.

It is worth noting that the cost of building frontier LLMs from scratch can be substantial in terms of energy and infrastructure.

Starting from an Open Source Foundation

Building on existing open source foundational models presents a practical approach to finding the starting point to create your own representative. Overall, this approach is usually less resource intensive in terms of both data and compute requirements as compared to pretraining from scratch.

A good example of this is the Trustworthy AI Dialogue Engine, or TAIDE, built by Sovereign AI initiative in Taiwan by further training of Meta's Llama-2 7B model with 41.44 billion tokens [source] of traditional Chinese text from Taiwanese domestic data sources. TAIDE outperformed several popular state-of-the-art models, such as its foundation Llama-2 7B model (trained predominantly on English data) in Taiwanese language and culture contexts.

TAIDE is not an outlier in building upon open source models like Llama-2. In fact, Singapore's SEA-LIONv3 family of models have also leveraged pretrained weights from Llama-3.1 and Gemma-2 through continued training. Notably, this approach is a departure from their earlier work on SEA-LIONv1, which was pretrained from scratch.

Model Sizing

The number of parameters in a model affects its ability on language understanding tasks - with larger models typically demonstrating better contextual understanding of complex tasks such as reasoning. The model size also has a direct impact on deployment costs, as larger models require more computational resources to run. They require more significant computational resources, particularly memory and processing power. The frontier LLMs today can range from tens to hundreds of billions of parameters, with some state-of-the-art models based on architectures like Mixture of Experts (MoE), reaching trillions of parameters.

While there is no strict industry definition, for this discussion, language models can be broadly categorized by their parameter count as follows:

- Small Language Models (or SLMs)
  - Range: O(1) billion parameters
  - Examples: Gemma-2 2B, Phi-2 2.7B, Llama-3 8B, Llama-Nemotron-Nano 4B/8B

- Medium-scale Language Models:
  - Range: O(10) billion parameters
  - Examples: Llama-3 70B, mixtral 8x7B, Llama-Nemotron-Super 49B
- Large Language Models
  - Range: O(100) billion parameters
  - Examples: Llama-3 405B, Nemotron-4 340B, Llama-Nemotron-Ultra 253B

Task-complexity

The size of the model you choose to build should correspond to your intended use case. While a very large generalist model can do well on a wide variety of use cases, it may be overkill and lead to increased customization and inference costs.

- Simpler domain-specific tasks like summarization, classification, sentiment-analysis, simple question-answering, and text editing can typically be handled by smaller language models. Additionally, for specialized applications, smaller domain-specific models can often outperform larger general-purpose models. Fine-tuning smaller models for specific domains to satisfactory accuracy can be more cost-effective as well.
- Tasks like multi-step reasoning, problem-solving, and open-ended generation are considered to be more challenging, and larger models tend to be more performant. They tend to grasp the complex relationships and nuances between words better.

In general, it is recommended to collect evaluation data specific to your use cases and run benchmarks on a range of model sizes. This is followed by customizing the smaller models and observing if their performance is satisfactory. Move up to larger model sizes to improve accuracy if needed.

Sometimes it may also be possible to perform “Task Decomposition” [source] to break down a more complex task into smaller sub-tasks that can be effectively handled by one or more smaller specialized models. There are several open-source agentic frameworks that can enable this. However, this comes at an expense – an increase in system complexity.

Deployment Cost Considerations

Training and deploying large models requires substantial amounts of memory, which can be a significant bottleneck. These models need sufficient GPU memory to store their weights, which can range from tens to hundreds of gigabytes.

Training necessitates storing intermediate activations and optimizer states, making it even more memory-intensive than inference. Although inference requires less memory than training, it can still be demanding, particularly when working with larger batch sizes.

To alleviate these memory requirements, researchers and practitioners often employ distributed training and inference across multiple GPUs and nodes using model parallelism techniques. For instance, the Llama 3 models were trained on a massive dataset of over 15 trillion tokens with training distributed over 16,000 NVIDIA H100 GPUs [source].

When it comes to inference, memory sizing is crucial to ensure efficient deployment. While training costs are a one-time expense, inference costs can be a recurring burden. Therefore, model size determination should carefully consider the memory requirements at inference time. Fortunately, many popular model families offer a range of sizes that can fit on a specific number of devices, making deployment more feasible. To illustrate this, consider the Llama-3 family of models [source], where Llama 3.1 8B was designed to fit on consumer GPUs, Llama 3.1 70B for large-scale AI applications, and Llama 3.1 405B for synthetic data generation.

To understand this better, the two main contributors to the GPU LLM memory requirement are model weights and the KV cache.

- Model weights: Memory is occupied by the model parameters. As an example, a model with 7 billion parameters (such as Llama 3.1 8B), loaded in 16-bit precision (FP16 or BF16) would take roughly 8B * sizeof(FP16) ~= 16 GB of memory.
- KV caching: Memory is occupied by the caching of self-attention tensors to avoid redundant computation.

Size of KV cache per token in bytes = 2 * (number-of-layers) * (number-of-attention-heads * dimension-of-head) * precision-in-bytes

For example, with a Llama 3.1 8B model in 16-bit precision, a batch size of 1, and using a 16K context -

Size of KV cache in Llama 3.1 8B (16K context, batch size 1) = (16384) * (2 * 32 * 8 * (4096 / 32) * 2) ~= 1.95 GB

NOTE: The model dimension of head is calculated as 4096/32 due to grouped query attention.

Total GPU memory can be calculated as -

Total Memory = Model-Weights + KV-Cache × Concurrent-Requests

Total Memory for Llama 3.1 8B (16-bit, 16K context, batch size N) = 16 GB + 1.95*N GB

As you can see, the KV cache grows linearly with batch size and sequence length, therefore it is important to understand how this affects the cost of infrastructure you will need to deploy your chosen model.

To dive deeper into how to calculate memory costs when deploying models—and explore more ways to optimize inference—refer to this blog. For a broader look at how to balance cost, latency, and performance during inference, take a look at this ebook.

Chapter 4: Model Training Pipeline

After addressing data curation and model sourcing considerations, the remainder of the model training pipeline for a sovereign LLM initiative largely follows the same general structure as that of any other LLM. This includes:

- Pretraining or Continual Pretraining: The model is pretrained on a large corpus of text data to learn the patterns and structures of the language.
- Model Alignment: The pretrained model is fine-tuned on a smaller, task-specific dataset to align its performance with the desired downstream task. This can be achieved through various techniques, such as:
  - Supervised Fine-Tuning (SFT): The model is fine-tuned on a labeled dataset to optimize its performance on a specific task.
  - Direct-Preference Optimization (DPO): The model is fine-tuned to directly optimize a preference-based objective function.
  - Other techniques: Other model alignment techniques, such as reinforcement learning from human feedback or contrastive learning, may also be employed.

NVIDIA NeMo enables all these stages efficiently, and at scale.

However, there are some key differences, particularly in the early stages of the pipeline, such as the tokenization step, which is the first step in both model training and inference.

Tokenizers and Vocabularies

Tokenization involves breaking down text into individual words or subwords, known as tokens, which are then represented as unique numerical identifiers. LLMs are, in essence, next-token predictors. Given a sequence of input tokens, the LLM's primary objective is to predict the next token in the sequence. This is usually achieved through multi-layer perceptrons (MLPs) and attention mechanisms that allow it to capture long-range dependencies and context within the input text.

Detokenization is the reverse process of tokenization, where the model's output tokens are mapped back to their corresponding words or subwords, resulting in coherent human-readable text.

Tokenizers are tools that enable this tokenization and detokenization. They are trained on and operate within the boundaries of pre-defined "vocabularies". These vocabularies represent the entire scope of text that an LLM can recognize and understand. In other

words, the vocabulary of a tokenizer determines the set of words, subwords, or characters that the model can process and generate.

Multilingual Capabilities of Pretrained Models

When fine-tuning a pretrained LLM on a new language or task, it is often beneficial to start with a model that has existing multilingual capabilities, even if it is not particularly proficient in the target language. This is because the tokenizer's vocabulary is likely to contain some representation of the target language, even if it is limited. This initial representation can serve as a useful starting point for further fine-tuning and adaptation to the target language.

For example, if you are working with a language that uses a non-Latin script, such as Chinese, Arabic, or Hindi, it may be helpful to start with a pretrained model that has been trained on a multilingual dataset that includes text in these scripts. Although the model may not be fluently proficient in the target language, the presence of some representation in the tokenizer's vocabulary can still provide a useful foundation for further fine-tuning and adaptation.

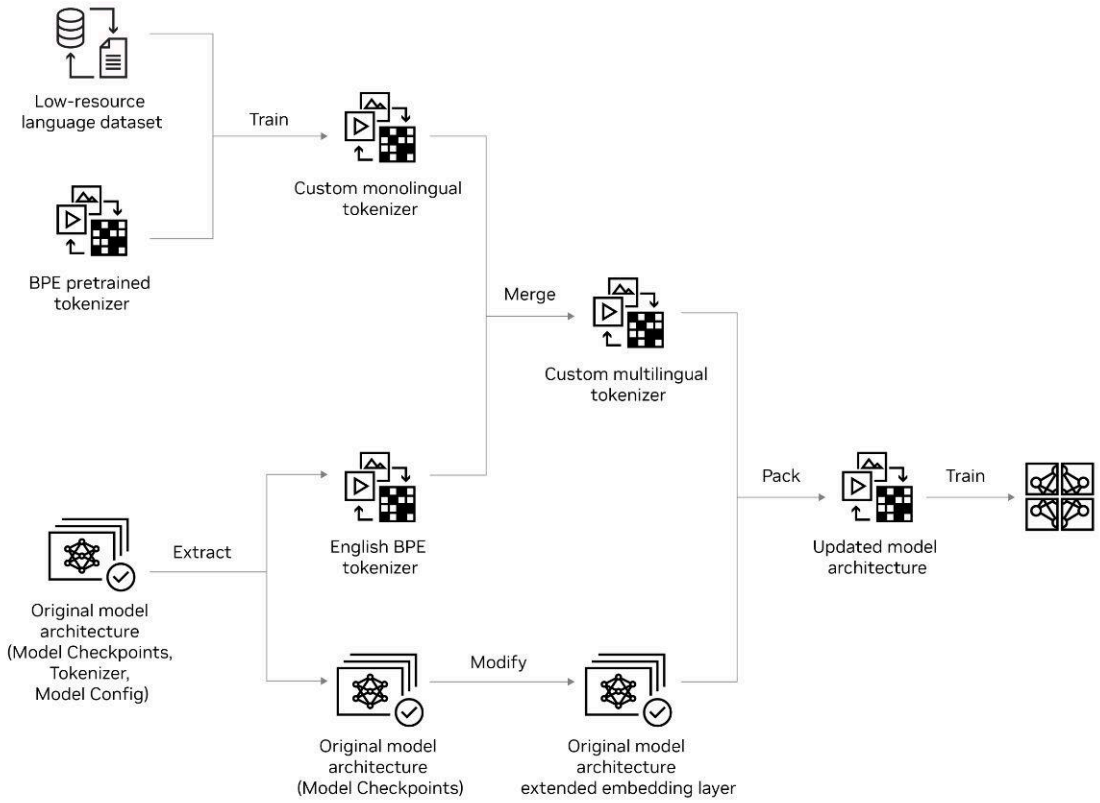
Customizing the Tokenizer

This relates to developing custom tokenizers that are specifically designed to handle the nuances of the target language. The tokenization process presents unique challenges when it comes to low resource languages, and as such leads to inefficient compression ratios and suboptimal model performance. Compression ratio refers to the efficiency with which a tokenizer can represent text as a sequence of tokens. It is typically measured as the reduction in length between the original text and the resulting token sequence. A higher compression ratio indicates that the tokenizer can represent the same information using fewer tokens, which can lead to improved processing efficiency and reduced computational costs for LLMs. For example, a tokenizer achieving a 4x compression ratio would represent 4 original tokens or characters with a single compressed token. Most foundation models adopt byte-pair encoding (BPE) tokenizers, a statistical method that creates tokens based on frequency in a corpus. Due to the scarcity of low-resource languages in the training corpora of these models, the original tokenizer does not adequately cover the unique characters, subwords, and morphology of the low-resource language. This requires extending the tokenizer with language-specific tokens and performing embedding distillation to ensure effective learning and adaptation.

Without a sufficiently expressive tokenizer, the model struggles to represent the low-resource language efficiently, leading to suboptimal performance. It is necessary to

build a customized tokenizer that enables the model to process and learn from the low-resource language data more effectively during continual pretraining.

Figure 2: Workflow for tokenizer customization



A few common approaches to alleviating this are as follows:

1. Extend the tokenizer of the model to include additional tokens from the target language. This significantly improves the compression ratio. Train a monolingual tokenizer and then merge it with the original English tokenizer. The advantage is that you can keep the original token mappings of English tokens and reuse the embedding layer of the foundation model. The time consumed for tokenizer training is much shorter.
2. Use a multilingual dataset that includes English to train the tokenizer from scratch. The advantage is that you can obtain a real distribution of the multilingual dataset. Training on cross-lingual translation data produces better performance compared to training on a monolingual corpus, but requires more training time and resources.

3. Instead of extending the tokenizer, replaced infrequent tokens with new tokens tailored for the target language. This method keeps the model capability the same by controlling the vocabulary and embedding table size. However, replacing too many tokens results in quality degradation.

Typical tokenizer customization involves the following flow:

1. Collect tokenizer training data by subsampling from the dataset prepared for continual pretraining.
2. Train the customized tokenizer by using the pretrained tokenizer as a starting point and train using custom data.
3. Manually merge both tokenizers.

In order to inspect the effectiveness of the new tokenizer on the target language and assess its impact on compression ratio, we can run both tokenizers on a sentence in the target language. For example, use the following Thai sentence as tokenizer input: **เมืองหลวงของประเทศไทยคือกรุงเทพฯ** (The capital of Thailand is Bangkok) and compare across both tokenizers.

Output of custom tokenizer:

```
['à¹Ģà¸¡', 'à¸·', 'à¸Ńà¸ĩà¸«à¸¥à¸§à¸ĩ', 'à¸Ĥà¸Ńà¸ĩà¸Ľà¸£à¸°à¹Ģà¸Ĺà¸¨à¹Ħà¸Ļà¸ç', 'à¸Ħ',
'à¸·', 'à¸Ńà¸ģà¸£', 'à¸,', 'à¸ĩà¹Ģà¸Ĺà¸ł à¸¯']
```

Output of original English tokenizer:

```
['à¹', 'Ģ', 'à¸', '¡', 'à¸', '·', 'à¸', 'Ń', 'à¸', 'ĩ', 'à¸', '«', 'à¸', '¥', 'à¸', '§',
'à¸', 'ĩ', 'à¸', 'Ĥ', 'à¸', 'Ń', 'à¸', 'ĩ', 'à¸', 'Ľ', 'à¸', '£', 'à¸', '°', 'à¹', 'Ģ',
'à¸', 'Ĺ', 'à¸', '¨', 'à¹', 'Ħ', 'à¸', 'Ĺ', 'à¸', 'ç', 'à¸', 'Ħ', 'à¸', '·', 'à¸', 'Ń',
'à¸', 'ģ', 'à¸', '£', 'à¸', ',', 'à¸', 'ĩ', 'à¹', 'Ģ', 'à¸', 'Ĺ', 'à¸', 'ł', 'à¸', '¯']
```

The customized tokenizer produces a much shorter list of tokens as compared to the original English tokenizer, as many Thai characters, especially those representing vowels and tonal markers, are likely to be considered out-of-vocabulary for an English tokenizer. The tokenizer may split these into individual bytes or replace them with a UNK token, increasing the token count. In fact, the original English-centric GPT2 tokenizer with a vocabulary size of 50k uses 3.8 times more tokens to encode Thai compared to a smaller tokenizer with a vocabulary size of 5k that was trained onThai. This will inevitably cost 3.8 times more compute in both training and inference. The same is exhibited in most low-resource languages.

Model Surgery - Extending the Embedding Layer

The embedding layer for LLMs needs to be extended when using a custom tokenizer because the new tokenizer introduces additional tokens, increasing the vocabulary size. This extension is necessary for several reasons:

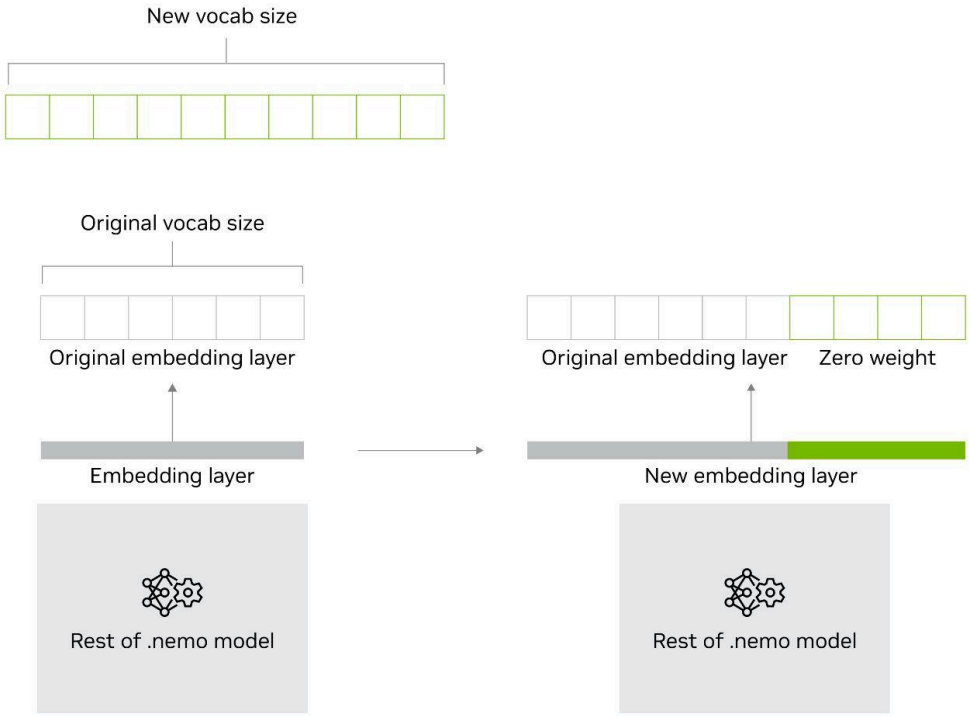
1. Vocabulary size mismatch: The custom tokenizer's vocabulary size is larger than the original pretrained model's vocabulary.
2. Token representation: Each token in the vocabulary needs a corresponding embedding vector in the embedding layer.
3. Knowledge preservation: Extending the embedding layer allows the model to retain knowledge from pretraining while accommodating new tokens.
4. Model compatibility: The embedding layer size must match the tokenizer's vocabulary size for the model to function correctly.

The embedding layer is thus extended using the following steps:

1. Create a new embedding layer with the desired increased vocabulary size.
2. Copy the existing weights from the original embedding layer.
3. Initialize the new vocabulary entries to zero or random weights, or the average of existing embeddings.
4. Replace the original embedding layer with the extended one.

This process ensures that the model can handle the additional tokens introduced by the custom tokenizer while maintaining its pretrained knowledge.

Figure 3: Extending the model embedding layer for the new tokenizer



In order to generate the new embedding layer, we calculate the dimension difference between the new embedding layer and the original embedding layer. We create a tensor based on this difference and concatenate it to the original embedding layer to form the new embedding layer. The difference is calculated based on the following parameters:

- Combined tokenizer vocabulary size
- Original embedding layer length
- A parameter in model_config.yaml, model.make_vocab_size_divisible_by

The difference is computed using the following equation:

$$Diff = \left[\frac{N}{X}\right] \times X - O$$

- Combined tokenizer vocabulary size = N
- Original NeMo embedding layer length = O
- model.make_vocab_size_divisible_by = X

In order to maximize computation efficiency, we pad the embedding layer to a number divisible by multiples of 8. Given a tokenizer vocabulary size, the model is expected to

have a padded embedding layer, and you must manually pad the new embedding layer accordingly.

Chapter 5: Model Evaluation

Once the AI model has been developed, the next step is to evaluate the model using a wide range of benchmarks to gauge its capabilities, in comparison to alternative solutions on the market. Benchmarks typically comprise question/task and solution pairs on a variety of topics and domains, with the vast majority being in English.

In addition to task accuracy, models should be evaluated on a number of other criteria:

- Translated benchmarks are useful to understand baseline model performance and how well the benchmark has absorbed any translated knowledge.
- Models should give locally correct answers to questions about history, geography, and important works of local literature.
- Benchmarks should measure how well the model responds to questions about local culture, taboos, norms, and laws.
- Models should be evaluated against other local dialects, knowledge, and for important current news if these things are applicable.

In the context of sovereign AI, benchmarks can be broadly organized into several technical categories:

- **Culture-agnostic benchmarks** are benchmarks that are, for a large part, based on universal or scientific truth. Examples include:
  - Coding and software engineering: SWE bench, humanEval, CodeXGLUE
  - Mathematics: GMS-8K, Math-500
  - Data science: MLE bench
  - Standardized test: SAT, ACT.

  While being presented in English, these benchmarks are largely language and culture-agnostic, for they involve math and engineering skills.

- **Western-oriented benchmarks**: these include benchmarks covering many subjects from STEM, humanities and social sciences, common sense and reasoning. Examples include MMLU, TruthfulQA, HotpotQA, BigBench, HellaSwag, ARC. While containing elements related to universal truth in logic, math, and the sciences, these benchmarks also comprise many elements that reflect the cultural and societal values and norms of Western countries.

- **Local benchmarks**: to gauge AI system proficiency across diverse linguistic and cultural contexts, recently there have been efforts to establish local benchmarks. Examples include MMLU-like benchmarks such as ArabicMMLU, CMMLU (Chinese),

PersianMMLU, TurkishMMLU, VNHSGE, VMLU (Vietnamese), which cover mostly academic knowledge. These benchmarks cover both region-agnostic as well as region-related questions. In the style of MMLU, these benchmarks comprise thousands of multiple-choice questions covering a few dozen distinct subjects, distributed across four overarching domains: STEM, Humanities, Social Sciences, and others.

Most of these localized versions of MMLU only support the local language or English in addition. Include is a recent project which established a much larger benchmark of 197,243 QA pairs from local exam sources to measure the capabilities of multilingual LLMs in a variety of regional contexts.

In addition to academic and professional benchmarks, there are culturally oriented benchmarks, for example:

- BLEnD: A Benchmark for LLMs on Everyday Knowledge in Diverse Cultures and Languages: a culturally diverse, multilingual benchmark of 52.6k handcrafted question-answer pairs from 16 regions and 13 languages.
- CVQA: Culturally-diverse Multilingual Visual Question Answering Benchmark featuring 10k culturally-driven questions in 31 languages across 30 countries, designed to evaluate and improve the cultural and linguistic diversity of multimodal models, revealing challenges for current state-of-the-art models.
- EtiCor: Corpus for Analyzing LLMs for Etiquettes: a region-specific etiquette corpus from five global regions, designed to evaluate LLMs' understanding of social norms, revealing that state-of-the-art models struggle with non-Western etiquettes.
- Benchmarking Hindi LLMs: A New Suite of Datasets and a Comparative Analysis: this work introduces a new suite of five datasets to address the lack of quality benchmarks for evaluating Hindi LLMs, and thoroughly describes the curation process which combines human annotation and a "translate-and-verify" method. This enables a more accurate and culturally nuanced comparative analysis of current Hindi LLM capabilities.

In addition to these academic and cultural benchmarks, there are dialog benchmarks such as Chatbot Arena. This is where users cast their preference when two models generate responses for the same prompt. Based on these pairwise “match”, an Elo score can be calculated for each model. Chatbot Arena is currently only available in English, but once a local LLM community grows strong, each country could establish its own arena.

NVIDIA NeMo Evaluator is a microservice that enables easy benchmarking of LLMs, both in English, as well as regional languages on various tasks, including common-sense

reasoning, multilingual question-and-answer, and multilingual translation across multiple languages.

Chapter 6: Model Safety and Governance

Model Safety and Governance refers to the practices and frameworks that ensure AI models and systems are secure, compliant, and aligned with ethical standards throughout their lifecycle. AI safety is a collection of operational guidelines and practices in place to ensure that foundation models comply with local cultural, ethical standards and local laws. At the most basic level, these models should exhibit helpful and harmless behavior to users. To mitigate any residual safety and security risk, guardrails are designed to keep the model staying within its intended applications.

NVIDIA NeMo Guardrails provides **programmable safety at runtime** for Large Language Models (LLMs), offering a robust framework to ensure AI outputs are **safe and accurate**.

Building local-language support into NeMo Guardrails requires thoughtful strategies to balance performance, safety, and scalability. Whether the goal is to maintain a single-language core or adopt region-specific fine-tuning, several approaches can be employed to meet the sovereign nation's needs.

Some of the best practices for enabling regional capabilities in NeMo Guardrails using LLMs range from building translational layers to directly fine-tuning models on regional languages. The key approaches discussed include:

1. **Building Translation Layers with the NeMo Guardrails Core in English** - Maintaining English as the primary processing language while adding translation layers for input and output rails.
2. **Working with Models where the NeMo Guardrails Core is Translated** - Creating a regional language guardrails core that processes rails in the native language. Leveraging LLMs trained on vast regional datasets can further facilitate seamless performance by integrating it with NeMo Guardrails.
3. **Finetuning Safety Models on a Single Regional Language** - Integrating NeMo Guardrails with finetuned safety models tailored to specific regional languages ensures the delivery of culturally nuanced, linguistically accurate, and safe outputs, aligned with local norms.
4. **Safety and Security Evaluation**

Building Translation Layers with the NeMo Guardrails Core in English

To add the translational layers, one way to achieve this is by taking advantage of Machine Translation models, like the NVIDIA Riva Translate-1.5B-Translate Neural Machine Translation model. This model is based on a Mistral-8B LLM (Large Language Model) that has been 'shrunk' to 1.5B parameters to specialize in the specific task of translation. In addition to the text that needs to be translated, it can receive instructions or samples of translation pairs to copy (using the few-shots technique). By doing this, you can achieve improved results during inference time without needing to fine-tune the entire model. The Riva Translate-1.5B-Instruct model allows for single-sentence translation among multiple languages, including Spanish, Portuguese, Hindi, and Vietnamese. This model leverages the few-shots technique to enhance translation accuracy by providing examples of translation pairs, which helps the model to better understand and generate more accurate translations.

To integrate these translational layers with NeMo Guardrails, the initial guardrails configuration is still written in English, with the model settings and model behaviors in terms of rails. An example configuration is shown below:

```
├── config
|   ├── config.yml
|   ├── prompts.yml
|   ├── dialog.co
```

The config.yml consists of the translation model along with the LLM and can be integrated with rails as follows:

```
%%writefile config/config.yml
models:
  - type: main
    engine: nim
    model: meta/llama-3.1-70b-instruct

  - type: main
    engine: nim
    model: nvidia/megatron-1b-nmt
```

The user prompt is first translated from the source regional language to the target English language before the input rails safeguard the input prompt. This behavior can be added into the guardrails configuration as follows:

```
%%writefile config/config.yml
models:
  - type: main
    engine: nim
    model: meta/llama-3.1-70b-instruct

  - type: nmt
    engine: nim
    model: nvidia/megatron-1b-nmt

rails:
  input:
    flows:
      - topic translation input $model="nmt"
      - self check input
    flows:
      - self check output
      - topic translation output $model="nmt"
```

Though this approach to integrating safety and security into multi-lingual applications is quite scalable, it is still dependent on the quality of the translation and adds to latency because of the translational layers. A more robust approach can include changing the core of the NeMo Guardrails to support multilingual LLMs.

Working with Models where the NeMo Guardrails Core is Translated

Most newer LLMs come with their own safety policies, but when dealing with low-resource regional languages, the translation of unsafe English prompts can effectively bypass safeguards. This issue arises due to linguistic inequalities in safety training data, where low-resource languages are underrepresented, leading to weaker safeguards. Incorporating these findings into NeMo Guardrails can align with the advantages of having a translated core, which offers higher compliance and lower latency compared to translational layers. By integrating a core mechanism that inherently supports low-resource languages, NeMo Guardrails could address these vulnerabilities more effectively.

When defining the configuration of guardrails for the translated core, the model configurations are still built in the same way as above with no additional flows of translation in the rails attribute and with a regional-language LLM.

For example, if the regional language is Hindi and we need to translate the NeMo Guardrails core to Hindi language, the most important change is in defining the prompts for the respective Hindi-language LLM. As mentioned earlier, NVIDIA's Nemotron-4-Mini-Hindi-4B, a Hindi-language LLM offered as a NIM for easier deployment and faster inference can be easily integrated with NeMo Guardrails as follows:

```
%%writefile config/config.yml
instructions:
  - type: general
    content: |
      नीचे एक बॉट और एक उपयोगकर्ता के बीच हाल की नौकरी की रिपोर्टों के बारे में बातचीत की गई है। बॉट तथ्यात्मक और संक्षिप्त है। यदि बॉट को किसी प्रश्न का उत्तर नहीं पता है, तो यह सच्चाई से कहता है कि यह नहीं जानता है।

models:
  - type: main
    engine: nim
    model: nvidia/nemotron-4-mini-hindi-4b-instruct
```

If the translated core is used within a chatbot application, it is important to understand the intention of extending support for the regional-LLMs, in the form of customized prompts. Since, the interaction with the LLM is designed in a task-oriented manner, i.e., each time the LLM is called, it must perform a specific task. To understand which tasks are important in the guardrails process, refer to the documentation on prompt customization with NeMo Guardrails.

`general`: generates the next bot message based on the history of user and bot messages. In the case of the translated core, the prompt for generating this from a canonical form can be added in the prompts.yml as follows:

```
%%writefile config/prompts.yml
prompts:
 - task: general
   content: |-
     {{ general_instructions }}
```

```
    {{ history | user_assistant_sequence }}
    Assistant:

- task: generate_next_steps
  mode: compact
  content: |-
    # कार्य: एक वार्तालाप में अगले चरणों को उत्पन्न करें।

    # इन उदाहरणों का प्रयोग करें:
    {{ examples | remove_text_messages }}
    {{ history | colang | remove_text_messages | last_turns(1) }}
```

Following this, developers can add additional topic safety checks using dialog flows. One such configuration example is as follows:

```
%%writefile config/dialog.co
define user express greeting
  "नमस्कार"
  "नमस्कार, आप कैसे हैं?"

define bot express greeting
  "नमस्ते! मैं आपकी कैसे मदद कर सकता हूं?"

define user ask capabilities
  "आप मेरी क्या मदद कर सकते हैं?"

define bot inform capabilities
  "मैं एक एआई सहायक हूं जो दिए गए ज्ञान आधार के आधार पर सवालों के जवाब देने में मदद करता है।"
```

Once these configurations are built as above, it can be wrapped around the model and any application in the usual way as mentioned in the configuration guide for NeMo Guardrails.

In the next section we can explore training of the translated core with diverse linguistic datasets, particularly focusing on underrepresented languages to understand how to improve the robustness of the foundation of safety measures. Additionally, real-time multilingual safety checks can be embedded directly into the core, allowing for faster and more accurate detection of unsafe inputs in any language. These enhancements not only mitigate the risks but also capitalize on the advantages of a translated core, ensuring both compliance and performance in safeguarding AI interactions across a wide range of languages.

Finetuning Safety Models on a Single Regional Language

Finetuning safety models on a single regional language is a crucial step toward addressing vulnerabilities in low-resource regional languages. By focusing on a single regional language, the safety model can be tailored to the specific linguistic nuances, cultural contexts, and idiomatic expressions of that language, which are often missed in generalized multilingual training. This target finetuning involves creating or curating high-quality, diverse datasets in the chosen language, encompassing both safe and unsafe examples to train the model effectively. Finetuning ensures the model can accurately detect context-specific risks, such as hate speech, misinformation, jailbreak, and other unsafe content unique to the language. Additionally, integrating real-time multilingual safety checks into the NeMo Guardrails core allows these finetuned models to seamlessly interact with safety frameworks, ensuring faster and more contextually accurate responses when handling inputs in the targeted language.

Safety and Security Evaluation

While LLMs are extensively evaluated on language understanding and generation benchmarks their evaluation on safety benchmarks has been limited. Some key safety benchmarks include Aegis and Garak.

Aegis is a content safety benchmark, which includes annotated data and fine-tuned classifier LLMs. The benchmark consists of safe and unsafe texts; the unsafe texts belong to 13 critical risk and 9 sparse safety risk categories. The unsafe prompts violate one or more categories of content safety, such as being sexual in nature or evoking violence. The CultureGuard work expands on this by creating a multilingual version of the Aegis dataset, supporting a total of 9 languages.

Garak is an LLM vulnerability scanner that probes the LLMs for common attacks. It probes for hallucination, data leakage, prompt injection, misinformation, toxicity generation, jailbreaks, and other weaknesses.

These benchmarks have been specifically designed for English and are not directly applicable to other regional languages. So to enable these frameworks for non-English languages, a multi-stage translation and back-translation based approach can be leveraged. In the context of Hindi language, the input prompt is first translated to Hindi and the response from LLMs is likewise in Hindi. The response is back-translated to

English so that we can run the Aegis LLM classifiers to identify the safe/unsafe responses. A similar strategy is adopted for Garak, where the response is back-translated before we pass it on to the detectors. However, some of these detectors might not work well on the translated output and might need to be modified. Although these translation-based methods are not ideal methods for safety benchmarking, they give us a fair estimate of the safety capabilities of LLMs. The ideal solution would be to come up with region-specific datasets and detectors/classifiers.

Chapter 7: Model Deployment

If training and customization are what make models, then inference is what puts a model into everyday use. NVIDIA NIM microservices offer scalable, high-throughput, low-latency inference for AI models, including large-scale LLMs.

Depending on how the localized model was trained, there are different deployment approaches.

NIM for Base Model Serving

The term base models, in this context, means the weights of the transformers layers and without any further modification on top, i.e. extra weight deltas that modify these weights and hence, model behavior (e.g., low-rank adapters). NIM allows base LLMs to be served with several simple steps. The NIM for a Nemotron Hindi model is one such example.

The process of starting a NIM can be summarized as follows. For further information, see Getting Started in the NIM documentation.

Step 1. Pulling and starting the NIM docker container on a GPU server.

```
export NGC_API_KEY=<PASTE_API_KEY_HERE>
export LOCAL_NIM_CACHE=~/.cache/nim
mkdir -p "$LOCAL_NIM_CACHE"
docker run -it --rm \
    --gpus all \
    --shm-size=16GB \
    -e NGC_API_KEY \
    -v "$LOCAL_NIM_CACHE:/opt/nim/.cache" \
    -u $(id -u) \
    -p 8000:8000 \
nvcr.io/nim/nvidia/llama-3.3-nemotron-super-49b-v1:latest
```

Step 2. Carry out inference using an HTTP REST request.

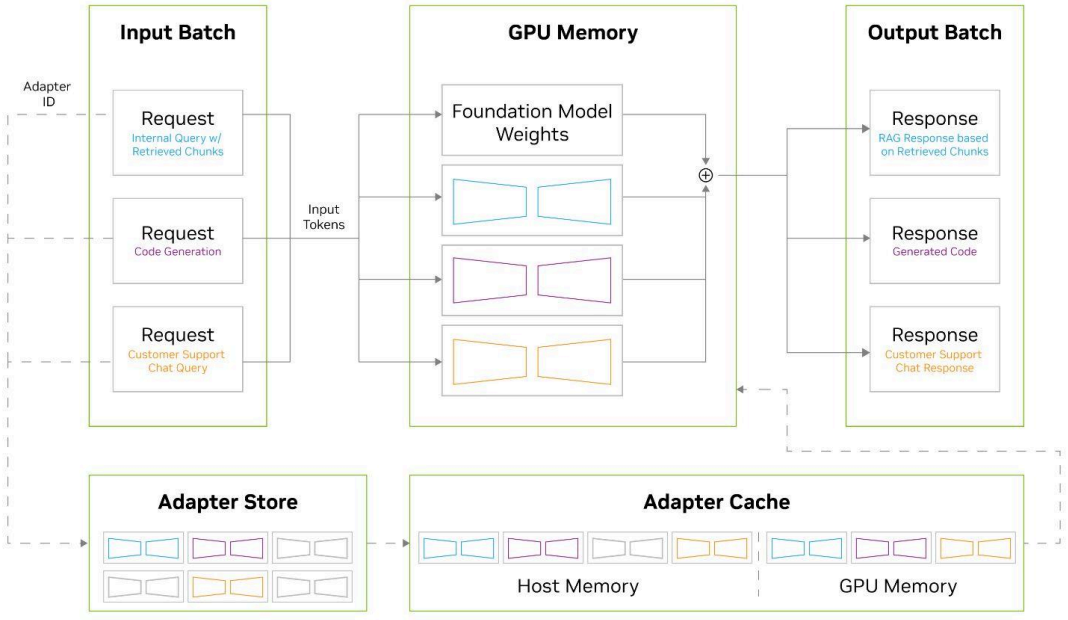
```
curl -X 'POST' \
'http://0.0.0.0:8000/v1/chat/completions' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
```

```
-d '{
    "model": "mistralai/mixtral-8x7b-instruct-v0.1",
    "messages": [{"role":"user", "content":"Write a limerick about the wonders of GPU
computing."}],
    "max_tokens": 64
}'
```

NIM for LoRA Model Serving

NVIDIA NIM supports LoRA adapters trained using either HuggingFace or NVIDIA NeMo, which can be used to add more robust support for low-resources languages on top of any base model supported by NIM. LoRA is especially suited for multi-language support, as it enables serving hundreds or even thousands of tuned models. A single base LLM may have many LoRA-tuned variants per language, all of which can be dynamically deployed as relevant. A standard system would require loading all the models independently, taking up large amounts of memory capacity.

Figure 4: The architecture of dynamic multi-LoRA with NIM



Deploying LoRA tuned models with NIM is similar to base model deployment, but requires an additional step to set up the LoRA model store. For example, we can use Chinese and

Hindi LoRA-tuned models directly from HuggingFace. In the case of HuggingFace, the LoRA must contain an adapter_config.json file and one of {adapter_model.safetensors, adapter_model.bin} files. The supported target modules for NIM are ["gate_proj", "o_proj", "up_proj", "down_proj", "k_proj", "q_proj", "v_proj"].

```
export LOCAL_PEFT_DIRECTORY=~/loras
mkdir $LOCAL_PEFT_DIRECTORY
pushd $LOCAL_PEFT_DIRECTORY

git clone https://huggingface.co/AdithyaSK/LLama3-Gaja-Hindi-8B-Instruct-alpha
git clone https://huggingface.co/shibing624/llama-3-8b-instruct-262k-chinese-lora

popd

chmod -R 777 $LOCAL_PEFT_DIRECTORY
```

The LOCAL_PEFT_DIRECTORY should be organized according to the structure below:

```
loras
├── llama-3-8b-instruct-262k-chinese-lora
|   ├── adapter_config.json
|   └── adapter_model.safetensors
└── LLama3-Gaja-Hindi-8B-Instruct-alpha
    ├── adapter_config.json
    └── adapter_model.safetensors
```

NIM enables loading of multiple LoRAs with different rank sizes, but sets an upper limit at 32. In the case of Hindi LoRA, we need to set the maximum rank above the default, as this particular model is tuned with a rank of 64.

```
export NIM_MAX_LORA_RANK=64
```

Now serve the NIM with LoRAs. The command is similar to running the Llama 3 foundation model, but will also load all the LoRA models stored in LOCAL_PEFT_DIRECTORY.

```
docker run -it --rm --name=$CONTAINER_NAME \
    --runtime=nvidia \
    --gpus all \
    --shm-size=16GB \
    -e NGC_API_KEY \
```

```
    -e NIM_PEFT_SOURCE \
    -e NIM_PEFT_REFRESH_INTERVAL \
    -e NIM_MAX_LORA_RANK \
    -v $NIM_CACHE_PATH:/opt/nim/.cache \
    -v $LOCAL_PEFT_DIRECTORY:$NIM_PEFT_SOURCE \
    -p 8000:8000 \
    nvcr.io/nim/meta/llama3-8b-instruct:1.0.0
```

Once served, use the following cURL command for inference, specifying the name of any of the LoRA models available in the model store. For example, run the Chinese adapter as shown below:

```
curl -X 'POST' \
  'http://0.0.0.0:8000/v1/completions' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
"model": "llama-3-8b-instruct-262k-chinese-lora",
"prompt": "介绍一下机器学习",
"max_tokens": 512
}'
```

A similar command is used for the Hindi LoRA adapter:

```
curl -X 'POST'   'http://0.0.0.0:8000/v1/completions'   -H 'accept: application/json'
-H 'Content-Type: application/json'   -d '{
"model": "LLama3-Gaja-Hindi-8B-Instruct-alpha",
"prompt": "मैं अपने समय प्रबंधन कौशल को कैसे सुधार सकता हूँ? मुझे पांच बिंदु बताएं और उनका वर्णन करें।",
"max_tokens": 512
}'
```

Serving Models with NVIDIA TensorRT-LLM

For finer granularity and control of deployment optimization, we can use TensorRT-LLM and NVIDIA Triton™ Inference Server. Compared to NIM, this method requires additional setup, but also enables developers with more control over optimization strategies.

One of the benefits of using TensorRT-LLM, is the ability to select any numeric precision. For example, we can perform post-training quantization (PTQ) in order to benefit from the more efficient INT4 numeric weight representation, saving significant memory

bandwidth and capacity. PTQ requires a representative small dataset to update the weights while maintaining statistical similarity.

PTQ requires a calibration dataset, and for low-resource languages it is critical to use a dataset containing a mix of English and the low-resource language at hand. For example, see the Dicta calibration dataset, consisting of a mix of Hebrew and English tokens. This will significantly improve INT4 accuracy, in comparison to using a generic English-only calibration dataset.

```
git clone https://huggingface.co/datasets/dicta-il/dictalm2.0-quant-calib-dataset
```

Then, quantize to INT4 precision using the calibration dataset:

```
python3 examples/quantization/quantize.py --kv_cache_dtype fp8 --dtype float16 --qformat
int4_awq --output_dir ./quantized_mistral_int4 --model_dir /dictalm-2-instruct
--calib_size 32
```

Finally, build the TensorRT-LLM engine:

```
trtllm-build --checkpoint_dir quantized_mistral_int4/ --output_dir
quantized_mistral_int4_engine/ --max_batch_size 64 --max_output_len 1024
--weight_only_precision int4 --gemm_plugin float16 --paged_kv_cache enable
```

After the engine is built, you can deploy the model with Triton Inference Server. The Triton Inference Server backend for TensorRT-LLM leverages the TensorRT-LLM C++ runtime for rapid inference execution and includes LLM specific optimization techniques such as in-flight batching and paged KV caching.

With low-resource language LLMs, the main point to keep in mind is to set up the tokenizer directories for Triton Inference Server, given we always train custom tokenizers. This step needs to be performed for both pre-processing and post-processing stages, and can be done using the *fill_template* script.

```
python3 tools/fill_template.py -i
all_models/inflight_batcher_llm/preprocessing/config.pbtxt
tokenizer_dir:${HF_MODEL},tokenizer_type:auto,triton_max_batch_size:32,preprocessing_inst
ance_count:1
python3 tools/fill_template.py -i
all_models/inflight_batcher_llm/postprocessing/config.pbtxt
tokenizer_dir:${HF_MODEL},tokenizer_type:auto,triton_max_batch_size:32,postprocessing_ins
tance_count:1
```

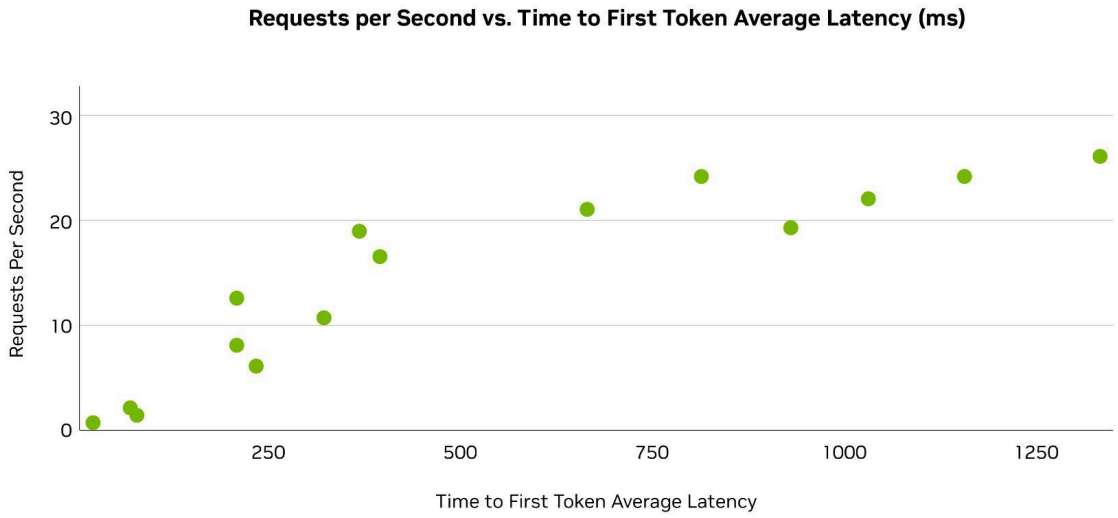
Performance Benchmarking and Hardware Provisioning

Once an instance of the model has been deployed, the next step is to benchmark the instance performance to get the instance throughput and latency metrics. Read the technical blog LLM Inference Benchmarking: Fundamental Concepts for a quick overview of key metrics.

This information, together with quality of service requirements (e.g. max latency) and expected peak demand (e.g. max concurrent users or request/s) will help estimate the required hardware. In turn, this information also helps with estimating the total cost of ownership (TCO) of the given solution.

NVIDIA **GenAI-Perf** is a client-side LLM-focused benchmarking tool, providing key metrics such as time to first token (TTFT), inter-token latency (ITL), token per sec (TPS), requests per sec (RPS) and more. For LLMs deployed with NIM, step by step documentation is provided to measure the instance performance easily.

Figure 5: Running GenAI-Perf to compare requests per second vs time to first token average latency (ms)



The performance data provided by GenAI-perf can be used to establish the latency-throughput trade-off curve, as exemplified below:

Each dot on this graph corresponds to a "concurrency" level, i.e. the number of concurrent users (and hence, concurrent requests put into the system). In most cases:

- At low concurrency, the system serves only a small number of users, hence the latency is low (left bottom part of the graph).
- At high concurrency, the system can leverage the batching effect to serve more users efficiently, however, this comes at the cost of increased latency (right top part of the graph).

TCO Calculator: Walk-through Example

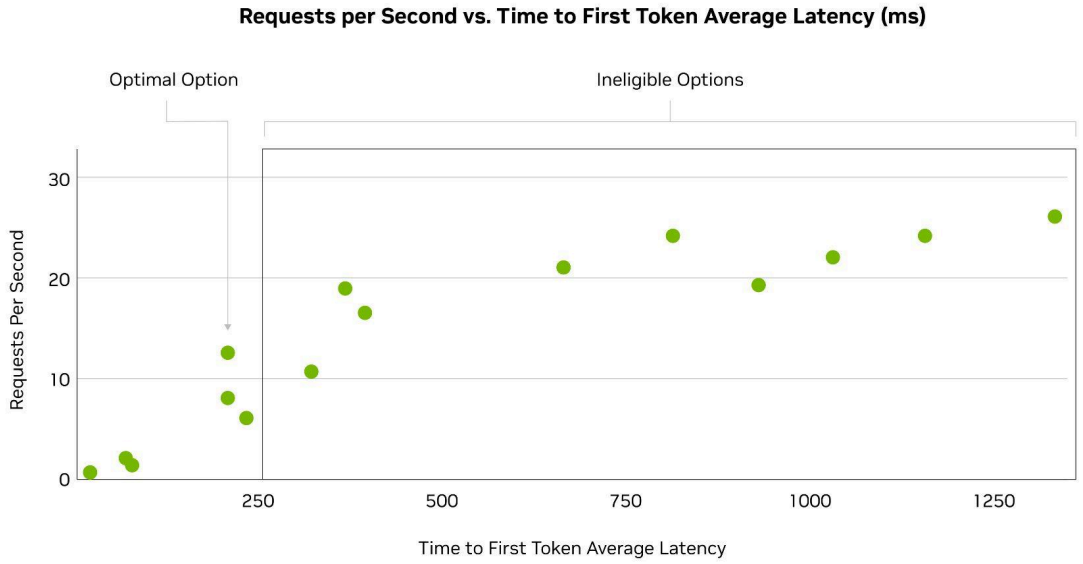
To estimate the amount of hardware required and the associated cost, follow these steps.

Step 1: Identify the model, GPU type, and the use case (input/output sequence length). Use the GenAI-perf tool to measure the performance metrics, which allows you to plot the latency-throughput trade-off curve.

Step 2: Identify the constraints:

- Type of latency and its maximum allowable value, for example, average time to first token at 250 ms.
- Planned peak request/s: how many concurrent requests the system is expected to serve.
  - Using this information, rule out the ineligible part of the performance chart (in this example, any data point on the right of the 250ms line.

Figure 6: Identifying the optimal deployment option



Of the remaining data points that satisfy the latency constraint, we want to select the one with the highest throughput since that will be the most economical option. This chart assumes that all the deployment options employ the same number of GPUs. If that is not the case, then the request per second metric should be normalized to request per second per GPU to allow a common ground for comparison.

Note the optimally achievable requests per second figure from the chart. The number of GPUs used by each instance should also be recorded.

Step 3. Identify the cost information

Table 1: Identifying hardware cost and example value

Hardware cost	Example value

1x Server cost	$100,000
GPUs per server	8
Server Depreciation Period (years)	4
Hosting Cost per Server per Year (USD)	$3000

Table 2: Identifying software cost and example value

Software cost	Example value
Yearly software licensing cost	$3500

Step 4. Calculate the costs

- Minimum Number of Model Instances Required: this is calculated as Planned peak request/s divided by optimally achievable requests per second per instance.

$$numberOfInstances = \frac{PlannedPeakRequests/s}{RPSperInstance}$$

- Number of servers: this is calculated as number of instances times the GPUs per instance, divided by the number of GPUs per server

$$numberOfServers = \frac{numberOfInstances \times GPUsPerInstance}{GPUsPerServer}$$

- Yearly server cost: this is calculated as the initial server cost divided by the depreciation period (in years), adding the yearly software licensing and hosting costs per server.

$$yearlyServerCost = \frac{initialServerCost}{depreciationPeriod} + yearlySoftwareCost + yearlyHostingCost$$

- Total server cost: this is calculated as the number of servers required, multiplied by the yearly cost per server.

$$TotalCost = yearlyServerCost \times numberOfServers$$

- Cost per 1000 prompts: this is calculated as the yearly server cost, divided by the total number of requests that a server can serve in a year, assuming 100% uptime. This can be adjusted for the actual uptime fraction.

$$costPer1000Prompts = \frac{yearlyServerCost}{RPSperInstance \times instancesPerServer \times 3600 \times 24 \times 365} \times 1000$$

For a deeper walkthrough on calculating inference costs, read the technical blog [Benchmarking LLM Inference Costs for Smarter Scaling and Deployment](#).

Chapter 8: Conclusion and Next Steps

Congratulations! You are now equipped with the information that you need to take on the technical challenges of sovereign AI.

In this technical brief, we have covered all different aspects of training and fine-tuning sovereign AI models, from curating data, selecting architecture, training from scratch or fine-tuning an existing model, evaluating a trained model and finally deploying them in production. Guidance on benchmarking the performance, sizing the inference infrastructure and estimating the total cost of ownership is also provided. We hope these topics have well equipped you with the necessary knowledge to build and deploy LLM applications in your own languages or domains.

With the fast moving landscape of LLM and agentic AI, there are topics that we leave for future revision, including training reasoning models in local languages, and agent systems.

To get started, you are encouraged to look into NVIDIA Enterprise AI Factory, mentioned earlier under hardware infrastructure in the introductory section. The AI Factory being a full-stack validated design, offers guidance for enterprises to build and deploy their own on-premises AI factory. In addition, the AI Factory is designed to support a wide range of AI-enabled enterprise applications, agentic and physical AI workflows, autonomous decision-making, and real-time data analysis.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA, the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

Arm

Arm, AMBA, and Arm Powered are registered trademarks of Arm Limited. Cortex, MPCore, and Mali are trademarks of Arm Limited. All other brands or product names are the property of their respective holders. ″Arm″ is used to represent Arm Holdings plc; its operating company Arm Limited; and the regional subsidiaries Arm Inc.; Arm KK; Arm Korea Limited.; Arm Taiwan Limited; Arm France SAS; Arm Consulting (Shanghai) Co. Ltd.; Arm Germany GmbH; Arm Embedded Technologies Pvt. Ltd.; Arm Norway, AS, and Arm Sweden AB.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com