A Unified 8B Transformer for Autoregressive Text, Image, and Audio Generation

Model Architecture Overview

The core model is a unified **decoder-only Transformer** that processes text, image, and audio tokens in a single autoregressive framework. Text input is tokenized (e.g. BPE) and embedded as usual. Image inputs are converted into patch tokens: the 1024×1024 RGB image is split into fixed-size patches (e.g. 16×16) which are flattened and linearly projected into the model's embedding space. (As in Vision-Transformer style, each patch embedding is augmented with positional information.) Audio input (when present) is first discretized into tokens via a neural codec (e.g. SoundStream or HuBERT-based quantizer) to produce a sequence of "sound tokens". All modality tokens are then concatenated into one sequence (with special "start-of-modality" markers if needed) and fed through a stack of Transformer layers. In effect, the model treats an image as a "foreign language" sequence of patch tokens, and similarly treats audio as a stream of quantized tokens. This single Transformer stack uses causal self-attention so that each predicted token (be it an image patch code or an audio code) attends only to previous tokens. After the Transformer layers, two parallel output heads produce the final predictions: one head projects to the *image token vocabulary* (for image generation) and the other to the *audio token vocabulary*. Image-generation proceeds autoregressively (next-patch/token prediction) until the full image is generated, and audio-generation similarly predicts one audio token at a time. This design follows recent "joint sequence" strategies (no separate vision encoder) where images and sounds are simply embedded as additional tokens.

Transformer Hyperparameters

To meet the **8B-parameter budget**, we choose dimensions similar to scaled language models. For example, a hidden dimension $d=4096d=4096$ with $\sim$[illegible]$36|sim|!36–3838$ layers and multihead size $h=32h=32$ (each head of size $d/h=128d/h=128$) gives on the order of $8\times1098|times10\text{^}9$ parameters. In table form, a feasible configuration is:

Component	Value
Hidden size (dd)	4096
Number of layers (NN)	36–38
Attention heads (hh)	32
Feedforward dim ($dffd\_\{ff\}$)	4×4096 = 16384
Context length	~4K–8K tokens (flexible)

These settings yield roughly $12Nd^2 \approx 8 12Nd^2 \approx 8$B parameters for the self-attention and FFN weights, plus embedding and output-projection weights. (For reference, many ~7B–8B LLMs use $d=4096, h=32, N \approx 32 d=4096, h=32, N \approx 32$–36.)

Embedding Strategies

- **Text Tokens:** We use standard subword tokenization (e.g. BPE or SentencePiece) with a vocabulary on the order of $20 – 50 20–50$K tokens. Tokens are mapped to d-dimensional embeddings. We add positional encodings (absolute or sinusoidal, or rotary/relative) to preserve word order.
- **Image Patch Tokens:** Each image patch (e.g. $16 \times 16 \times 3 16 \times 16 \times 3$) is flattened and projected by a linear layer into a d-dimensional "patch embedding". We also add spatial positional encodings (either 2D learned/PE or relative) so the model knows each patch's location. Optionally, one can use a quantization-based tokenizer (e.g. VQ-VAE or the IBQ method) to map patches to discrete codebook IDs, which are then embedded. Either way, image tokens enter the Transformer in the same space as text tokens.
- **Audio Tokens:** Audio waveforms are tokenized using a neural codec. For example, a SoundStream encoder produces quantized codes (e.g. 1024-way codebooks at multiple resolutions) at an effective bitrate of a few kbps. We embed each discrete audio code into a d-dimensional vector and add a 1D temporal position encoding. (Recent work compresses audio to as low as ~0.23 kbps in tokens.) We also introduce special <|beginaudio|>/<|endaudio|> tokens to mark audio segments.

All token embeddings (text, image, audio) share the same hidden dimension and layer-norm scheme. Modality or token-type embeddings can also be added so the model can distinguish text vs image vs audio tokens. In practice we unify them by type tags or special start tokens. Notably, work has shown that simply concatenating vision and text tokens (without separate cross-attention modules) can work well, essentially treating the image as a "foreign language."

Vocabulary and Token Representation

- **Text Vocabulary:** Subword tokens (BPE) with size $\sim 50K \sim 50K$. Includes special tokens (e.g. <|start|>, <|end|>, padding, etc.).
- **Image Token Vocab:** If using vector quantization, the codebook size might be on the order of 10410^4 (e.g. 16K entries). Each output token is one code index. If using patch embeddings directly, there is no fixed vocab – we generate the continuous patch vectors via the AR decoder and feed them into a pretrained image decoder (e.g. VQGAN's decoder) to reconstruct pixels. In either case, autoregressive generation is over a discrete token set.
- **Audio Token Vocab:** Using a neural codec like SoundStream yields multiple quantizers each of size ~1024. Practically, we can interleave or group these so that each "audio token" prediction

corresponds to one code index out of ~1024 possible values. We add explicit <|beginaudio|>/<|endaudio|> to the vocabulary to mark audio sequences.

Thus the model's output heads are two softmax layers: one over the image-code vocabulary and one over the audio-code vocabulary (plus implicit EOS tokens).

Image Generation Decoder Details

Images are generated autoregressively in "patch token" order. During decoding, a causal mask ensures each new patch token is predicted only from previously generated tokens (text or image). For example, we may enumerate patches in raster-scan order. At each step the model's output head produces a probability distribution over the image token vocabulary, and we sample (or take argmax) the next token. Once all tokens for a $1024\times1024|times1024$ image are produced, a separate (pretrained) decoder network converts the discrete tokens into pixels. This follows the common approach of discrete token AR image models. In fact, recent work has extended autoregressive Transformers to very high-resolution images by using efficient tokenizations (e.g. merging patches) and next-token prediction. In our design, we keep a one-to-one mapping between image patch and token for simplicity.

Unlike diffusion models, our decoder does **not** use attention over a separate image encoder; instead the unified Transformer itself predicts the next image token. (This is analogous to "text-to-image" GPT-style models.) We maintain causal self-attention throughout. Optionally, one could use classifier-free guidance at generation time, but training remains standard autoregressive NLL.

Attention Mechanisms

We use standard **multi-head self-attention** in every Transformer layer, with causal masking. To improve efficiency and handle long sequences (images and audio can produce thousands of tokens), we employ modern optimizations: *FlashAttention* (or similar fused kernels) to reduce memory/compute costs and achieve close to linear scaling. We incorporate **relative positional encodings** (such as Rotary Positional Embeddings) for all sequences. For example, [12] used rotary embeddings in its Conformer-based audio model. Relative encodings (e.g. ALiBi or RoPE) allow stable generalization to longer contexts than seen in training.

In principle, one could also use sparse or local attention patterns for efficiency, but for an 8B model on a single GPU full attention with FlashAttention should suffice. (If needed, *chunking* or *memory-efficient attention* methods from LongFormer/Megatron could be integrated.) We maintain 32 heads per layer (each of dimension 128) and full connectivity.

Training and Losses

We train the model end-to-end on paired (text, image) and (text, audio) examples. The objective is **next-token prediction** (cross-entropy) over the mixed sequence of text, image, and audio tokens. In practice we often separate tasks: e.g. "text+optional image → image" or "text → audio" as two types of batches. The loss is the sum of negative log-likelihoods for the target image tokens and the target audio tokens. If both modalities are generated jointly, the combined loss is simply the sum (possibly weighted) of each sequence's CE loss. (One may need to adjust weights to balance the modalities.) For example, Liu et al. found it helpful to upweight new audio tokens by a factor of 10 in an audio-LLM setting. In our case, equal weighting with proper scheduling usually works if datasets are balanced.

Image-specific losses: If using a learned image tokenizer (e.g. VQ-VAE), we separately ensure it reconstructs images well. In joint training, we freeze or pretrain the tokenizer and focus on the AR loss. One could optionally include a perceptual or pixel-space loss on reconstructions, but it's more straightforward to rely on the codebook.

Audio-specific losses: Similarly, we assume a pretrained audio codec (like SoundStream) that provides token embeddings; we train only on the AR loss for audio tokens. Optionally, one could pretrain a "semantic" stage (HuBERT) followed by an "acoustic" stage (SoundStorm), but here we unify into a single AR pass from text to sound tokens.

For optimization, standard practice applies: use AdamW with appropriate weight decay, and learning-rate warmup/decay schedules. We train on large batches with teacher forcing. Recent best practice suggests also using techniques like *z-loss* (a logit stabilization term) for very large vocabularies, though this is optional.

Critically, the **training data** should be high-quality and richly annotated. We train on large-scale image-caption corpora with *detailed, long-form captions*. Empirical studies show that multimodal models benefit more from rich descriptions: synthetic "dense" captions that mention objects, attributes, and context significantly improve performance. In line with these findings, our datasets (e.g. filtered LAION or custom caption sets) emphasize descriptive alt-text. Synthetic re-captioning (via an LLM captioner) may be used to enhance sparse web text. For audio, we train on paired text-speech or text-audio datasets (e.g. audiobooks, transcripts) that align spoken or environmental sounds with text.

GPU Feasibility Strategies

To fit an 8B-parameter model on a single modern GPU (e.g. 48GB A100), we employ several memory-saving techniques:

- **Mixed-Precision Training:** We train in FP16/BF16. Half-precision cuts memory use roughly in half with minimal accuracy loss. This allows a much larger model or batch to fit. Nvidia's guides show FP16 can yield up to 3× speedup and ~2× memory savings on Volta/Ampere GPUs.

- **Gradient Checkpointing:** We activate activation (gradient) checkpointing on blocks. This frees most intermediate activations at the cost of ~20–30% extra compute. The memory savings are approximately $O(N)O(\sqrt{N})$ of the original, which is crucial for deep stacks. (As noted by Hugging Face, this "significantly less GPU memory" is achieved with only modest slowdown.)
- **Gradient Accumulation:** We split large batches into micro-batches. By accumulating gradients over several forward/backward passes, we effectively increase batch size without extra memory for activations. This is essential if we want an effective batch larger than what memory allows. Gradient accumulation trades off wall-clock speed (more iterations) for fit in memory.
- **8-bit Optimizers:** We use 8-bit variants of AdamW (via BitsAndBytes). These store optimizer states in 8-bit instead of 32-bit, reducing optimizer memory by ~4×. HF's benchmarks show finetuning large models uses 75% less memory and runs ~4× faster with 8-bit Adam, without hyperparameter changes. This is highly recommended for large models on limited GPUs.
- **Efficient Attention Kernels:** We utilize implementations like FlashAttention (v2) which reduce attention memory to linear scaling. As Li et al. report, FlashAttention transforms the usual quadratic attention cost to a more manageable form on single GPUs. Using such kernels (and PyTorch 2.0's `torch.compile` if available) maximizes throughput.
- **Batch/Sequence Packing:** We pack sequences tightly and avoid padding overhead. We may pack multiple shorter contexts into one batch to fill the GPU. We also ensure our batch sizes and layer dimensions are multiples of 8 or 64 to leverage GPU tensor core efficiency.
- **8-bit Quantization (Inference):** Although not for training, note that at inference time we can quantize the model weights (e.g. to 4- or 8-bit) to fit even in 16GB GPUs.

By combining these practices – mixed precision, checkpointing, accumulation, 8-bit optimizers, and optimized kernels – an 8B-parameter Transformer can be trained and fine-tuned on a single high-end GPU within current 2025 capabilities.