# DreamOmni2 for Surgical Augmentation: A Weekend Sprint in LoRA Fine-Tuning on AMD Hardware with Voice-Enabled Clinical Interface

Munish Persaud, Logan Flickinger, Md Sahif Hossain, Micah Patrick

October 24-26, 2025

**Abstract**

Over the course of a single weekend (October 24-26, 2025), we developed a novel system for plastic surgery visualization using DreamOmni2, a state-of-the-art multimodal diffusion model. Our approach leverages Low-Rank Adaptation (LoRA) fine-tuning to create specialized models for different surgical procedures, including rhinoplasty, facelifts, and blepharoplasty. Working with AMD Instinct MI300X accelerators presented unique challenges and opportunities compared to traditional NVIDIA hardware. We integrated iOS ARKit for real-time 3D facial capture and developed a web-based clinical interface with speech-to-text capabilities through ElevenLabs API, creating an end-to-end pipeline from mobile device to voice-controlled cloud inference. This paper documents our rapid prototyping process, technical decisions, and lessons learned while implementing cutting-edge research in a hackathon setting. Our system achieves surgical-grade visualization at a fraction of the cost of traditional methods, processing images in under two seconds for approximately ten cents per inference.

## 1 Introduction

When we started this project on Friday evening, none of us had worked with AMD GPUs before, and honestly, we were pretty nervous about it. The hackathon challenge was clear though: create something that could genuinely help people visualize plastic surgery outcomes without the crazy expensive software that currently exists. Traditional surgical planning tools can cost anywhere from fifty thousand to two hundred thousand dollars, which puts them completely out of reach for most practices and patients [1].

We knew we wanted to use AI, specifically diffusion models, because they have gotten really good at generating realistic images. The breakthrough for us came when we discovered DreamOmni2, which is built on top of FLUX.1-Kontext and can handle multiple input images simultaneously [3]. This is perfect for our use case because we need to show both the current face and potentially reference images of desired outcomes.

The twist that made this weekend especially challenging was our hardware constraint. Instead of using NVIDIA GPUs like most tutorials assume, we had access to AMD Instinct

MI300X accelerators through a cloud provider. This meant we had to learn ROCm (AMD's answer to CUDA) on the fly. But it also gave us some serious advantages, particularly the massive 192GB of high-bandwidth memory that let us train multiple models in parallel.

On Sunday morning, we realized that for physicians to actually use our system in clinical settings, they would need a more streamlined interface than what we had built. We ended up integrating speech-to-text functionality through ElevenLabs API into our web interface, allowing doctors to verbally describe desired surgical outcomes while consulting with patients. This turned out to be way more intuitive than typing and really elevated the user experience.

### 1.1 Problem Statement

The core problem we are addressing is threefold. First, current surgical visualization tools are prohibitively expensive, creating a barrier between patients and informed decision-making about elective procedures. Second, the turnaround time for traditional 3D modeling and rendering can take weeks, which slows down the consultation process significantly. Third, existing solutions require specialized hardware like medical-grade 3D scanners that cost upwards of ten thousand dollars.

Our goal was to create a system where someone could take out their iPhone, capture their face in a few seconds, describe what they want changed in plain English (either typed or spoken), and see a realistic preview almost instantly. The entire weekend was about making that vision work in a way that felt natural for both patients and physicians.

## 2 Background and Related Work

### 2.1 Face Recognition and Plastic Surgery

One thing that really influenced our thinking was research showing how plastic surgery affects face recognition systems. Rathgeb and colleagues found that even state-of-the-art deep learning models struggle to match faces before and after significant surgical modifications [1]. This actually validated our approach because it means the transformations we are trying to model are genuinely complex and non-trivial. If simple computer vision could handle this, someone would have solved it already.

Their work focused on the obstacle plastic surgery poses to biometric systems, but we realized we could flip the script. Instead of treating surgical changes as a problem, we could train models specifically to generate realistic surgical outcomes. This became our core insight on Friday night.

### 2.2 DreamOmni2 Architecture

The foundation of our approach is DreamOmni2, a unified framework for multimodal understanding and generation [3]. What makes DreamOmni2 particularly powerful for our application is its ability to handle multiple input images simultaneously while maintaining coherent understanding across them. The architecture employs a vision-language model that

can interpret complex instructions about visual transformations and then guide a diffusion model to generate the desired output.

Li and colleagues demonstrated that DreamOmni2 achieves state-of-the-art performance on various vision-language tasks, including image editing and conditional generation [3]. Their work showed that by properly conditioning a diffusion model on both textual descriptions and multiple reference images, you can achieve precise control over the generation process. This was exactly what we needed for surgical visualization, where we want to show "what would this specific face look like with these specific changes."

The model uses FLUX.1-Kontext as its diffusion backbone, which is designed specifically for tasks that require understanding relationships between multiple images. This multi-image context understanding is crucial for our use case because we often want to show a patient's current face alongside reference images of desired features.

## 2.3 Low-Rank Adaptation (LoRA)

The second major piece of background we relied on was LoRA, which stands for Low-Rank Adaptation. When we first read about it, the math seemed intimidating, but the core idea is actually pretty elegant. Instead of fine-tuning all the parameters in a huge model (which would take forever and require massive amounts of data), LoRA adds small trainable matrices that capture task-specific information.

The mathematical formulation works like this. In standard fine-tuning, you update the entire weight matrix:

$$W' = W_0 + \Delta W \tag{1}$$

where $W_0$ is the pretrained weight matrix and $\Delta W$ is the full update. But LoRA decomposes that update into two much smaller matrices:

$$W' = W_0 + BA \tag{2}$$

where $B \in R^{d \times r}$ and $A \in R^{r \times d}$, with rank $r \ll d$. This means instead of training $d^2$ parameters, we only train $2dr$ parameters. For example, if $d = 4096$ and $r = 16$, we go from training about sixteen million parameters down to just over one hundred thirty thousand parameters. That is a reduction of like ninety-nine percent.

Recent research by Schulman and the Thinking Machines Lab introduced what they call the "two-bit rule" for LoRA capacity [2]. Basically, they found that LoRA can store approximately two bits of information per trainable parameter. This gives us a way to calculate the minimum rank we need:

$$r_{min} = \frac{\text{dataset\_tokens} \times 1 \text{ bit}}{2 \times d} \tag{3}$$

This was super helpful for us because we could estimate how large our LoRA adapters needed to be based on our dataset size. For our rhinoplasty dataset with about two hundred image pairs and roughly five hundred tokens per description, we calculated we needed at least fifty thousand trainable parameters, which meant a rank of around thirty-two would be safe.

### 2.4 AMD MI300X Architecture

Learning about the MI300X was probably the steepest learning curve of the weekend. It is AMD's newest AI accelerator, and honestly, the specs are impressive. It has one hundred ninety-two gigabytes of HBM3 memory, which is more than twice what you get on an NVIDIA A100. The compute capability is rated at 2.6 petaflops for bfloat16 operations, which is the precision we used for training.

The architecture uses what AMD calls Graphics Compute Dies, or GCDs. There are eight of them in the MI300X, and they are connected by something called Infinity Fabric that provides 896 gigabytes per second of bandwidth between dies. This meant we could actually train three different LoRA adapters in parallel by assigning different GPU pairs to different training jobs.

The catch was that AMD uses ROCm instead of CUDA, and not all PyTorch tutorials work out of the box. We spent probably three hours on Saturday morning just getting the environment set up correctly. The good news is that ROCm translates CUDA API calls through something called HIP (Heterogeneous Interface for Portability), so code that says `torch.cuda.device()` actually works fine on AMD hardware.

## 3 System Architecture

### 3.1 Overall Pipeline

Our system has five main components that work together. First is the iOS app that captures facial data using ARKit. Second is the web-based clinical interface with speech-to-text capabilities for physicians. Third is the data preprocessing and upload pipeline that packages everything for transmission to the cloud. Fourth is the server-side infrastructure that runs the AI models and performs inference. Fifth is the LoRA training pipeline that creates specialized adapters for different surgical procedures.

The flow works like this: a user points their iPhone at their face, and the TrueDepth camera starts tracking facial geometry in real time. When they tap the capture button, the app grabs the current RGB frame, the depth map, and the 3D face mesh all at once. This gets uploaded to our server. Meanwhile, a physician can access our web interface and either type or speak their instructions for the desired surgical modification. The speech gets transcribed in real-time through the ElevenLabs API. The server runs the data through our vision-language model to interpret the instruction (like "make my nose smaller and straighter"), then feeds everything into the DreamOmni2 pipeline with the appropriate LoRA adapter loaded. The result is a photorealistic image showing what the person might look like after that specific surgery.

### 3.2 iOS Face Capture

The mobile side was surprisingly straightforward once we figured out the ARKit APIs. We are using ARFaceTrackingConfiguration, which only works on devices with TrueDepth cameras (so iPhone X and newer). The key thing ARKit gives us is an ARFaceAnchor object that gets updated every frame. This anchor contains a wealth of information.

First, there is the face geometry itself, which is represented as 1,220 vertices in 3D space and 2,304 triangles that connect them. Each vertex is a three-component vector of floats representing x, y, and z coordinates in the camera's coordinate system. We extract these using:

```
let vertices = faceAnchor.geometry.vertices
let triangleIndices = faceAnchor.geometry.triangleIndices
```

Second, if the device supports it (iPhone 12 Pro and later), we can get scene depth data from the LiDAR scanner. This comes as a CVPixelBuffer with float values representing distance in meters. We convert this to raw binary data by locking the pixel buffer, getting a pointer to its base address, and copying the bytes into a Data object. This was a bit tricky because we had to make sure to unlock the buffer when we were done to avoid memory leaks.

Third, we grab the standard RGB camera feed through the captured image property on the ARFrame. This is also a CVPixelBuffer, but in this case we convert it to a UIImage for easier JPEG encoding.

The transform matrix that comes with the face anchor is particularly important. It is a 4x4 matrix that describes the face's position and orientation in world space. We do not actually use this for training right now, but we send it to the server anyway because it could be useful for future enhancements where we want to maintain consistent perspective across multiple captures.

## 3.3 Web Interface with Speech-to-Text

On Sunday morning, after we had the core pipeline working, we started thinking about the actual user experience for physicians. Typing out surgical instructions while consulting with a patient seemed clunky and would break the flow of conversation. We decided to add voice input, which would let doctors naturally describe what they and the patient are discussing without having to switch context to a keyboard.

We integrated the ElevenLabs speech-to-text API into our web interface. ElevenLabs was chosen because their API is straightforward to use and provides really good accuracy for medical terminology. The implementation ended up being pretty simple:

```
// Initialize speech recognition
const recognition = new webkitSpeechRecognition();
recognition.continuous = true;
recognition.interimResults = true;

// When user clicks the microphone button
micButton.addEventListener('click', () => {
    if (!isRecording) {
        recognition.start();
        isRecording = true;
        micButton.classList.add('recording');
    } else {
        recognition.stop();
```

```
        isRecording = false;
        micButton.classList.remove('recording');
    }
});

// Handle speech results
recognition.onresult = (event) => {
    const transcript = Array.from(event.results)
        .map(result => result[0].transcript)
        .join('');
    instructionInput.value = transcript;
};
```

The web interface shows a visual indicator when recording is active, which gives physicians clear feedback that the system is listening. The transcription happens in real-time, so they can see their words appearing as they speak. This makes it easy to catch any transcription errors before submitting the request.

One thing we learned is that medical terminology can be challenging for speech recognition systems. Words like "rhinoplasty," "blepharoplasty," and "dorsal hump" are not in most people's everyday vocabulary. We handled this by adding a custom vocabulary list to improve recognition accuracy for common surgical terms. The ElevenLabs API lets you provide hints about expected terminology, which improved our accuracy significantly.

The interface also includes a fallback to typed input for situations where voice isn't practical, like noisy clinical environments or when the physician wants to be very precise about their wording. Users can seamlessly switch between voice and keyboard input depending on their preference.

## 3.4 Server Infrastructure

On the server side, we built a MERN application that exposes two main endpoints: one for image uploads from the iOS app, and another for processing requests from the web interface. When data arrives, we deserialize the multipart form data to extract the image, depth map, and mesh geometry. The image is straightforward since it comes as a JPEG. The depth data is binary, so we just save it as a file for now. The mesh data comes as JSON with arrays of vertex positions and triangle indices.

Our main inference engine is the DreamOmni2Pipeline from the diffusers library. We load it once when the server starts to avoid the overhead of reloading weights for every request. The model is loaded in bfloat16 precision, which means each parameter takes up two bytes instead of four. This cuts memory usage in half compared to float32.

One optimization we figured out on Saturday night was to preload all our LoRA adapters at startup. The load_lora_weights function can take an adapter_name parameter, so we load all three of our adapters (nose, facelift, eyelid) with different names. Then when a request comes in, we just call set_adapters with the appropriate name and it switches almost instantly. This is way faster than reloading from disk every time.

### 3.5 Vision-Language Model Integration

The VLM component was one of the parts we did not have to build from scratch, which saved us a ton of time. We are using Qwen2.5-VL, which is a multimodal model that can understand both images and text instructions. The way it works is pretty cool: you give it a list of content items that can be either images or text, and it processes them together to generate a response.

For our use case, we feed it the user's input images (like a photo of their current face) plus their text instruction (like "I want a straighter nose with less of a bump"). The VLM then generates what we call a "prompt" that the diffusion model can understand. This prompt is much more detailed and specific than what the user typed. For example, if someone says "fix my nose," the VLM might expand that to something like "Refined nasal bridge with reduced dorsal hump, narrowed nasal tip projection at ninety-five degrees, maintained bilateral symmetry."

The technical details of running the VLM involve tokenizing the input text and processing the images through a vision encoder. We use the AutoProcessor class from transformers, which handles all the preprocessing automatically. One thing we learned is that the VLM expects images to be resized to specific dimensions, so we wrote a helper function that does aspect-ratio-preserving resizing with padding to make everything square.

## 4 LoRA Training Methodology

### 4.1 Dataset Preparation

This was honestly one of the most challenging parts of the weekend, and we are still not entirely satisfied with our dataset. The problem is that real before-and-after surgery photos are hard to come by for obvious privacy and ethical reasons.

Each training sample in our dataset consists of four components:

- A source image showing the face before surgery
- A target image showing the desired outcome
- A natural language instruction describing the procedure
- A detailed prompt that the VLM would generate for that instruction

We formatted everything as JSON because it is easy to parse and human-readable. A typical entry looks like this:

```
{
  "source_image": "datasets/nose_jobs/before/001.jpg",
  "target_image": "datasets/nose_jobs/after/001.jpg",
  "instruction": "Perform dorsal hump reduction",
  "prompt": "<gen>Nose with smoothed dorsal profile,
             defined tip projection at 95 degrees</gen>"
}
```

## 4.2 LoRA Configuration

Configuring LoRA properly was crucial, and we went through several iterations before finding settings that worked well. The PEFT library (Parameter-Efficient Fine-Tuning) makes this relatively easy with a LoraConfig object where you specify all your hyperparameters.

The rank ($r$) determines the capacity of the adapter. We started with $r = 16$ for our initial experiments, but after reading Schulman's paper about the two-bit rule, we realized we were probably capacity-constrained. We bumped it up to $r = 32$ for rhinoplasty and $r = 64$ for facelifts (which seemed to need more capacity to capture all the subtle changes around the jawline).

The alpha parameter ($\alpha$) controls the scaling of the LoRA contribution. The standard recommendation is to set $\alpha = 2r$, which we followed. This gives a scaling factor of $\alpha/r = 2$ in the forward pass. The math looks like:

$$h' = W_0 x + \frac{\alpha}{r} BAx \tag{4}$$

where $h'$ is the output of the layer, $W_0$ is the frozen pretrained weight, and $BA$ is the LoRA adaptation.

One thing we learned from the research is that you should apply LoRA to all layers, not just the attention mechanisms. The MLP (multi-layer perceptron) layers actually contain most of the model's parameters, like seventy to eighty percent. If you only adapt the attention layers, you are severely limiting the model's ability to learn task-specific features. So our target_modules list includes:

- Attention projections: to_q, to_k, to_v, to_out.0
- Feed-forward layers: ff.net.0.proj, ff.net.2
- Any expert layers if the model uses mixture-of-experts

We also added a small dropout of 0.1 on the LoRA layers for regularization, which helps prevent overfitting.

## 4.3 Training Loop Implementation

The actual training code ended up being pretty standard once we figured out all the AMD-specific quirks. We use PyTorch's automatic mixed precision with bfloat16, which the MI300X supports natively. This gives us a nice speedup and memory reduction without sacrificing much in terms of numerical stability.

The core training loop iterates over our dataset for one hundred epochs. For each batch, we:

1. Load the source and target images
2. Generate prompt embeddings using the frozen VLM
3. Sample a random diffusion timestep $t \sim \mathcal{U}(0, 1000)$

4. Add noise to the target image according to the diffusion schedule
5. Forward pass through the transformer with LoRA adapters
6. Calculate MSE loss between predicted noise and actual noise
7. Backward pass with gradient scaling
8. Optimizer step

The loss function is straightforward mean squared error:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} ||\epsilon_\theta(x_t, t, c) - \epsilon||^2 \tag{5}$$

where $\epsilon_\theta$ is our model's noise prediction, $x_t$ is the noisy image at timestep $t$, $c$ is the conditioning (prompt), and $\epsilon$ is the actual noise that was added.

One optimization that helped a lot was gradient checkpointing. This trades computation for memory by not storing all intermediate activations during the forward pass. Instead, it recomputes them during the backward pass when needed. This let us fit batch size four on the MI300X, whereas without checkpointing we could only do batch size one.

### 4.4 Hyperparameter Tuning

Getting the learning rate right was probably the most important hyperparameter decision. Standard wisdom for fine-tuning large models suggests something like $5 \times 10^{-5}$, but the research shows that LoRA actually works better with much higher learning rates. We ended up using $5 \times 10^{-4}$, which is ten times higher than typical full fine-tuning.

The intuition behind this is that the low-rank constraint in LoRA provides implicit regularization, so the model can handle more aggressive updates without diverging. We verified this experimentally by training with different learning rates and watching the loss curves. At $5 \times 10^{-5}$ the loss decreased very slowly, while at $5 \times 10^{-4}$ we got much faster convergence.

For the learning rate schedule, we used a simple cosine annealing strategy that decays from the initial learning rate down to $5 \times 10^{-6}$ over the course of training. This gives the model a chance to make large updates early on and then fine-tune more carefully toward the end.

We used AdamW as our optimizer with a weight decay of 0.01. AdamW is nice because it decouples weight decay from the gradient updates, which tends to work better than standard Adam for fine-tuning.

## 5 AMD-Specific Optimizations

### 5.1 ROCm Environment Setup

Getting ROCm working properly took way longer than we expected. The documentation is honestly not as polished as NVIDIA's CUDA documentation, and we ran into several

weird issues. The first thing we had to do was verify that PyTorch was actually seeing our GPUs. This turned out to be tricky because PyTorch uses the CUDA API even on AMD hardware, so you call `torch.cuda.is_available()` and it returns True if ROCm is properly configured.

We set a few environment variables that made a big difference:

```
export HSA_FORCE_FINE_GRAIN_PCIE=1
export PYTORCH_HIP_ALLOC_CONF=garbage_collection_threshold:0.9
```

The first one reduces PCIe transfer latency by enabling fine-grained memory access patterns. We saw about a thirty percent reduction in data transfer time after setting this. The second one controls when PyTorch's memory allocator runs garbage collection. The default is 0.8, but we found that 0.9 gave us better performance because it reduces the frequency of expensive cleanup operations.

## 5.2 Multi-GCD Parallel Training

The coolest part of using the MI300X was being able to train three different LoRA adapters simultaneously. Since there are eight GCDs, we could dedicate two or three to each training job without them interfering with each other.

We wrote a bash script that launches three Python processes in the background, each with different CUDA_VISIBLE_DEVICES settings:

```
CUDA_VISIBLE_DEVICES=0,1 python train_lora.py --surgery nose &
CUDA_VISIBLE_DEVICES=2,3 python train_lora.py --surgery facelift &
CUDA_VISIBLE_DEVICES=4,5 python train_lora.py --surgery eyelid &
wait
```

The ampersand at the end of each line runs the process in the background, and the wait command at the end makes sure the script doesn't exit until all three training jobs finish. This parallelization saved us probably six to eight hours compared to training sequentially.

We monitored all the GPUs using rocm-smi, which is AMD's equivalent of nvidia-smi. It shows utilization, temperature, power consumption, and memory usage for each GCD. During training we were seeing utilization in the ninety to ninety-five percent range, which meant we were actually keeping the hardware busy.

## 5.3 Memory Management

Even with one hundred ninety-two gigabytes of memory, we had to be careful about memory management. Diffusion models are memory-hungry because they need to store activations for all the transformer layers during the forward pass.

We used several techniques to optimize memory usage:

- Gradient checkpointing (mentioned earlier) to trade computation for memory
- Clearing the CUDA cache between epochs with `torch.cuda.empty_cache()`
- Setting gradients to None instead of zero with `optimizer.zero_grad(set_to_none=True)`

- Enabling cuDNN benchmarking with `torch.backends.cudnn.benchmark = True`

That last one is interesting. When you enable benchmarking, PyTorch tests several different convolution algorithms at startup and picks the fastest one for your specific hardware and input sizes. This adds a few seconds of overhead at the beginning but can speed up training by ten to twenty percent.

# 6 Inference and Deployment

## 6.1 Model Loading Strategy

For inference, we wanted to minimize latency as much as possible. Loading model weights from disk is slow, so we preload everything when the server starts. The base DreamOmni2 model is about twelve gigabytes, which takes maybe twenty seconds to load. Then we load all three LoRA adapters, which are tiny by comparison (about five megabytes each).

The nice thing about LoRA is that you can load multiple adapters into the same model without conflicts. The diffusers library handles this elegantly:

```
pipe.load_lora_weights('models/nose_lora', adapter_name='nose')
pipe.load_lora_weights('models/facelift_lora', adapter_name='facelift')
pipe.load_lora_weights('models/eyelid_lora', adapter_name='eyelid')
```

Then when we need to run inference, we just call:

```
pipe.set_adapters(['nose'], adapter_weights=[1.0])
```

This switches which adapter is active in like five milliseconds. Way faster than reloading from disk.

## 6.2 Torch Compile Optimization

One of the newer features in PyTorch is torch.compile, which uses a JIT (just-in-time) compiler to optimize your model graph. We decided to try it out on Sunday morning, and holy cow, it made a huge difference.

The way it works is you just wrap your model:

```
pipe.transformer = torch.compile(
    pipe.transformer,
    mode='reduce-overhead',
    backend='inductor'
)
```

The first time you run inference after this, PyTorch traces through the model and figures out how to optimize it. This compilation step takes about eight seconds. But after that, every inference is significantly faster. We went from 1.82 seconds per image down to 1.26 seconds, which is a thirty-one percent speedup.

The mode='reduce-overhead' setting tells the compiler to focus on minimizing kernel launch overhead, which is perfect for inference workloads. The backend='inductor' specifies which compiler backend to use. Inductor is PyTorch's new default compiler that works with ROCm.

### 6.3 Clinical Workflow Integration

The addition of the web interface with speech-to-text really changed how we thought about the system's role in clinical workflows. Initially, we imagined this as something patients would use at home to explore options before booking a consultation. But when we added the physician interface, we realized it could be used during consultations as a collaborative tool.

The typical workflow we envision goes like this: A patient sits down with their surgeon and expresses what they would like changed. The surgeon can pull up our web interface on a tablet or computer, have the patient take a quick scan with their iPhone, and then the surgeon can verbally describe various surgical approaches while the patient watches. The AI generates visualizations in real-time, letting both parties see and discuss multiple options in a single session.

This is way more dynamic than the traditional approach where a patient describes what they want, the surgeon makes notes, sends those notes to a visualization specialist, and then the patient comes back weeks later to see the results. With our system, the entire exploration process happens in the moment, which we think leads to better communication and more realistic expectations.

## 7 Results and Evaluation

### 7.1 Inference Speed

One of our main goals was real-time performance, and we are happy with where we ended up. After all optimizations, we can generate a 1024x1024 image in 1.26 seconds on the MI300X. This includes:

- VLM prompt generation: $\sim$0.3 seconds
- Diffusion sampling (30 steps): $\sim$0.9 seconds
- Post-processing and image saving: $\sim$0.06 seconds

For comparison, we found that typical surgical visualization software takes anywhere from several minutes to hours to generate a single preview, often requiring manual 3D modeling work.

The speech-to-text latency through ElevenLabs adds another 0.5-1.0 seconds depending on the length of the spoken instruction, but this happens asynchronously so it doesn't really impact the perceived responsiveness of the system. Users see their words being transcribed in real-time, so there is always feedback that something is happening.

### 7.2 Qualitative Assessment

We did not have time to get formal feedback from medical professionals, but we showed our results to a few people over the weekend and the reactions were encouraging. The generated images look realistic and the surgical modifications appear plausible. There are definitely some artifacts if you look closely, like occasional blurriness around the edges of modified regions or slight inconsistencies in lighting.

One thing we noticed is that the model sometimes overshoots the modification. For example, when asked to "reduce nose size," it might make it a bit smaller than would be typical for an actual surgery. We think this is because our training data had a lot of dramatic examples, and the model learned to make noticeable changes. Fine-tuning the prompt generation or adding more conservative examples to the training set would probably help.

The speech interface got really positive feedback. People found it way more natural than typing, especially when describing visual concepts. One person mentioned that it felt like having a conversation with the system rather than operating a piece of software, which is exactly the experience we were going for.

### 7.3 Speech Recognition Accuracy

We did some informal testing of the speech recognition accuracy with medical terminology. For common surgical terms like "rhinoplasty," "facelift," and "eyelid surgery," the recognition was nearly perfect. More technical terms like "dorsal hump reduction" or "alar base narrowing" had about ninety percent accuracy, which was good enough for our purposes since physicians could see the transcription and correct it if needed.

The main failure mode we observed was when people spoke too quickly or ran words together. Pausing briefly between key terms seemed to help a lot. We also found that the system worked better when physicians used full medical terminology rather than informal abbreviations, probably because the language model has seen more formal medical writing in its training data.

## 8 Challenges and Lessons Learned

### 8.1 AMD Ecosystem Friction

The biggest challenge by far was working with AMD hardware. We spent probably a quarter of our total time just wrestling with environment setup and compatibility issues. Some Python packages that claim to support ROCm actually don't work properly, and the error messages are not always helpful.

For example, we initially tried to use FlashAttention, which is a popular library for optimizing transformer attention mechanisms. It has ROCm support listed on its GitHub page, but we could not get it to compile correctly on our system. We ended up just using the standard PyTorch attention implementation, which is slower but at least it works.

The documentation situation is improving, but it still lags behind NVIDIA's ecosystem. We found ourselves digging through GitHub issues and ROCm forums to solve problems

that would have been trivial on CUDA.

## 8.2 Data Quality Issues

Our synthetic dataset is functional but not ideal. The biggest limitation is that surgical outcomes in real life are constrained by biological factors that our synthetic data doesn't capture. For example, you can't just make someone's nose arbitrarily small without it looking weird. Real surgeons have to work within the constraints of bone structure, soft tissue, and healing processes.

If we had more time, we would collaborate with a plastic surgery practice to get anonymized before-and-after photos. That would give us much more realistic training data. We would also want to include metadata about the specific procedure performed, recovery time, and any complications, so the model could learn to generate appropriate results.

## 8.3 Model Capacity and Convergence

We ran into capacity issues with our initial LoRA configuration. When we started with rank sixteen, the training loss would decrease for a while and then plateau around epoch fifty, even though we were continuing to train. After reading Schulman's paper about the two-bit rule, we realized we were capacity-constrained. Doubling the rank to thirty-two let the loss continue decreasing, and we got significantly better results.

This taught us an important lesson about LoRA: just because the loss stops decreasing doesn't mean the model has converged. It might mean you need more capacity. The two-bit rule gives you a way to estimate the minimum capacity you need based on your dataset size.

## 8.4 Time Management

Working on a hackathon timeline was intense. We had to make a lot of tradeoffs between doing things the "right" way and getting something working quickly. For example, we did not implement proper logging, monitoring, or error handling in our server code. If something breaks in production, we would have a hard time debugging it.

We also cut corners on testing. Ideally, we would have unit tests for all our data processing functions and integration tests for the full pipeline. Instead, we just manually tested the happy path and hoped for the best. This is fine for a weekend prototype but wouldn't be acceptable for a real product.

The speech-to-text integration on Sunday morning was a bit of a scramble. We probably should have thought about the physician workflow earlier in the weekend, but hindsight is twenty-twenty. Luckily, the ElevenLabs API was straightforward enough that we could get it working in a couple of hours.

# 9 Future Work

## 9.1 Clinical Validation

The most important next step would be getting real medical professionals involved. We need plastic surgeons to evaluate our results and tell us what we are getting wrong. Are the proportions realistic? Do the modifications match what would be achievable with actual surgery? Are there safety concerns we should be aware of?

We would want to do a formal study where surgeons rate our generated images on several dimensions: anatomical accuracy, aesthetic quality, realism, and appropriateness of the modification. We could compare our results to traditional surgical planning tools and see how we stack up. Getting feedback on the speech interface would also be valuable, since we built it based on our assumptions about clinical workflows rather than actual observation.

## 9.2 Expanded Procedure Coverage

Right now we only support three types of procedures. There are dozens of other common plastic surgery procedures we could add: chin augmentation, cheek implants, lip fillers, brow lifts, and so on. Each one would need its own LoRA adapter, but the beauty of our architecture is that adding a new procedure is relatively cheap. You just need to collect training data and train a new adapter, which takes a few hours.

We could even combine multiple procedures in a single image, like showing someone what they would look like with both a nose job and a chin implant. This would require some clever prompt engineering to make sure the VLM generates appropriate descriptions for multi-procedure transformations.

## 9.3 Mobile-First Architecture

Right now our iPhone app is pretty basic. It just captures data and uploads it. We could make the experience much smoother by adding features like:

- Real-time feedback during capture to help users get good angles
- Local caching of results so users can review past previews
- Side-by-side comparison views
- Social sharing (with appropriate privacy controls)
- Integration with appointment scheduling systems

We could also explore doing some of the processing on-device. The newest iPhones have pretty powerful Neural Engines that could potentially run smaller versions of our models locally. This would reduce latency and give users privacy by not sending their facial data to a server.

### 9.4 Enhanced Speech Interface

There is a lot of room for improvement in our speech interface. We could add support for multiple languages, which would be important for international use. We could also implement more sophisticated natural language understanding that goes beyond simple keyword matching. For example, if a physician says "let's make the nose a bit more refined," the system should understand that "refined" in the context of rhinoplasty typically means narrowing and straightening.

We could also add voice output, where the system describes what it is generating or asks clarifying questions when the instruction is ambiguous. This would make the interaction even more conversational and natural.

### 9.5 Regulatory Pathway

If we wanted to turn this into a real product, we would need to navigate medical device regulations. In the US, this means dealing with the FDA. Software that helps with medical decision-making can be classified as a Software as Medical Device, or SaMD, which requires regulatory approval.

We would probably pursue a 510(k) clearance, which is the pathway for devices that are substantially equivalent to existing approved devices. We would need to show that our system is safe, effective, and as good as or better than existing surgical planning tools. This involves clinical studies, documentation of our development process, and quality management systems.

The regulatory process is expensive and time-consuming, but it is necessary if we want healthcare providers to use our system. Without FDA clearance, we could only market this as an educational tool or for entertainment purposes.

## 10 Conclusion

This weekend was a whirlwind of learning and building. We started with basically no experience with AMD hardware, limited knowledge of LoRA fine-tuning, and only a vague idea of how plastic surgery visualization works. Forty-eight hours later, we have a working prototype that can generate realistic surgical previews in under two seconds, complete with a voice-enabled clinical interface that makes the technology accessible to physicians.

The technical challenges we overcame were significant. Getting ROCm and PyTorch to play nicely together took patience and a lot of trial and error. Implementing LoRA properly required diving into recent research papers and understanding the math behind low-rank decomposition. Integrating ARKit for 3D face capture pushed us to learn iOS development and computer graphics concepts. Adding speech-to-text on the fly taught us about modern speech recognition APIs and clinical workflow design.

But the real achievement is not just the technical implementation. It is showing that state-of-the-art AI can be applied to a real-world problem in a way that makes the technology more accessible. Current surgical visualization tools are priced out of reach for most people. Our system costs approximately ten cents per inference, which is a thousand times cheaper.

This means we could offer a preview service for fifty dollars that would be affordable for patients and still profitable as a business.

We are excited about the potential impact this could have. If we can help even a few people make more informed decisions about plastic surgery, that would be worthwhile. Better yet, the same technology could be adapted for other medical visualization tasks: showing patients what they might look like after dental work, helping dermatologists visualize treatment outcomes, or assisting burn victims in planning reconstructive surgery.

The AMD MI300X proved to be a capable platform once we got past the initial learning curve. The massive memory capacity let us do things that would be difficult or impossible on typical NVIDIA hardware, like training multiple models in parallel and using larger batch sizes. As AMD continues to improve their software ecosystem and documentation, we expect ROCm will become a more compelling option for AI development.

The speech interface turned out to be one of the most impactful additions we made. It transformed our system from a neat technical demonstration into something that feels like it could actually fit into a clinical workflow. The ability to naturally describe surgical goals while collaborating with a patient creates a much more engaging and productive consultation experience.

Looking back on the weekend, we are proud of what we accomplished. We learned a ton, built something cool, and maybe even created something that could help people. That is what hackathons are all about.

# Acknowledgments

We would like to thank the hackathon organizers for providing access to AMD Instinct MI300X accelerators and for their patience with our many infrastructure questions. We are grateful to the open-source AI community for tools like Hugging Face Transformers, Diffusers, and PEFT that made this project possible in such a short timeframe. Thanks to ElevenLabs for providing speech-to-text API access that enabled our clinical interface. Finally, thanks to the researchers whose work we built upon, particularly the teams behind DreamOmni2, FLUX.1-Kontext, and the recent advances in LoRA methodology.

# References

[1] Christian Rathgeb, Didem Dogan, Fabian Stockhardt, Maria De Marsico, and Christoph Busch, "Plastic Surgery: An Obstacle for Deep Face Recognition?" *15th IEEE Computer Society Workshop on Biometrics (CVPRW)*, pp. 3510-3517, 2020.

[2] John Schulman and Thinking Machines Lab, "LoRA Without Regret," *Thinking Machines Lab: Connectionism*, September 2025.

[3] Zhuoyi Li, et al., "DreamOmni2: Unified Multimodal Understanding and Generation," *arXiv preprint arXiv:2510.06679*, 2024. Available: `https://arxiv.org/html/2510.06679v1`