

Black Forest Labs releases FLUX.3 Image Klein model in Diffusers

Source copy kept by wheresthe.ai for the story "Black Forest Labs releases FLUX.3 Image Klein model in Diffusers" (<https://wheresthe.ai/d/intl-black-forest-labs-releases-flux-3-image-klein-model-in-2026-10/>).

Source: <https://huggingface.co/blog/flux-3>

Text saved 18 Oct 2026 (the page as published).

SHA-256 of the saved text: d6f4b1a48304e6d9c1d0339222f513ec02ac62481ef3bf8d4da1a464fa31db58

[Back to Articles](#) a]:hidden">

Welcome FLUX.3 Image Klein in Diffusers: The latest open image generation model from Black Forest Labs

Published October 15, 2026

Update on GitHub Upvote 37

+31

Aritra Roy Gosthipaty ariG23498 Follow

Sayak Paul sayakpaul Follow

Linoy Tsaban linoyts Follow

:last-child]:mb-0">

Black Forest Labs FLUX.3 Image Klein is here, and it runs in Diffusers from day one. One model does text-to-image and multi-reference image-to-image, and it adds something new, the layout control . You write bounding boxes into the prompt to say where each part of the image should go, or which parts of a reference image to move.

Both open checkpoints are in the FLUX.3 Image Klein collection :

Checkpoint What it is Defaults

black-forest-labs/flux-3-image-klein Time step distilled, for fast inference 16 steps, no CFG

black-forest-labs/flux-3-image-klein-base Base model 50 steps, CFG

You can head on to the Hugging Face space and generate some images right now!

TL;DR

One model for text-to-image, multi-reference image-to-image, and bounding-box layout control.

The distilled checkpoint makes a 1024 × 1024 image in about 2 seconds on an H100.

NVFP4 and FP8 weights load with from_single_file , for smaller GPUs.

LoRA training scripts for both text-to-image and image editing.

Ships as a Modular Diffusers pipeline, so you can pop blocks out or add your own.

Contents

Getting started

What FLUX.3 can do

Base model and quantized variants

LoRA finetuning

What it costs to run

The architecture

Modular Diffusers

Try it

Getting started

Install the latest diffusers and transformers (for the Qwen3-VL text encoder):

```
pip install -U diffusers transformers
```

Load the distilled checkpoint and generate an image:

```
import torch from diffusers import ModularPipeline

pipe = ModularPipeline.from_pretrained( "black-forest-labs/flux-3-image-klein" )
pipe.load_components(dtype=torch.bfloat16) pipe.to( "cuda" )

pipe(prompt= "a red fox" , output= "images" )[ 0 ].save( "fox.png" )
```

What FLUX.3 can do

Text-to-image

FLUX.3 responds best to a full scene description with the subject, the setting, the light, the composition, and the style. The official prompting guide from Black Forest Labs has more tips. The distilled checkpoint generates a 1024 × 1024 image in 16 steps by default. Pass height , width , or num_inference_steps to change them.

```
prompt = ( "A wide-angle wildlife photograph of a red fox standing alert in a snow-covered forest clearing at golden  
hour. " "The fox is centered in the frame, its thick orange winter coat contrasting with the fresh white snow, breath  
" "visible in the cold air. Snow-laden pine trees fill the softly blurred background, and low warm sunlight rakes "  
"across the scene from the left, casting long blue shadows." )
```

```
fox = pipe( prompt=prompt, generator=torch.Generator().manual_seed( 0 ), output= "images" )[ 0 ] fox.save(  
"fox.png" )
```

Text to image

Image-to-image

Pass a reference image (or more) with the image argument. Write the prompt as a caption of the image you want , with the change included.

```
prompt = ( "The same red fox in the same snow-covered forest clearing, now at night under a full moon: cool blue  
" "moonlight, deep shadows, stars visible through the pine branches, the fox's eyes catching the light." ) night =  
pipe( prompt=prompt, image=fox, # from the previous text to image example  
generator=torch.Generator().manual_seed( 0 ), output= "images" , )[ 0 ] night.save( "night.png" )
```

Image to image

With several references, refer to them in the prompt as <ref_image_0> , <ref_image_1> , ... in the order they are passed to image :

```
red_fox = pipe(prompt= "a red fox" , generator=torch.Generator().manual_seed( 0 ), output= "images" )[ 0 ] bench  
= pipe(prompt= "a wooden bench" , generator=torch.Generator().manual_seed( 0 ), output= "images" )[ 0 ]  
combined = pipe( prompt= "The fox from <ref_image_0> sitting on the wooden bench from <ref_image_1> in a  
snowy setting." , image=[red_fox, bench], output= "images" , )[ 0 ]  
combined.save( "combined.png" )
```

<ref_image_0> : a red fox <ref_image_1> : a wooden bench The fox on the bench

Layout control

Add a JSON list of regions after the caption. Each region has an id , a bbox , and a desc , and the caption mentions it as <id> .

Boxes are written as [y_min, x_min, y_max, x_max] , y first , on a 0–1000 scale. Because the scale is normalized,

the same box works at any image size.

Placing regions in a new image

```
layout_prompt = ( "Minimalist graphic illustration featuring a black silhouette of a person <silhouette_1> centered against a " "solid, vibrant chartreuse background <background_1>. The figure is captured in a dynamic, mid-stride running " "pose, facing toward the left side of the frame. " [{"id": "background_1", "bbox": [0, 0, 1000, 1000], "desc": "A flat field of neon yellow-green with a subtle 'paper texture and fine, organic grain."}, {"id": "silhouette_1", "bbox": [150, 150, 850, 850], "desc": "A 'black silhouette of a person in motion, made of dense stippled ink. The leading edges are solid; the trailing 'arm and leg dissolve into coarse square pixels and scattered dots."}]' )
```

```
runner = pipe(prompt=layout_prompt, generator=torch.Generator().manual_seed( 0 ), output= "images" )[ 0 ]
```

Each region's desc is drawn inside its bbox

Moving regions within a reference image

For edits, each region says where its content comes from (src_bbox in the reference) and where it goes (tgt_bbox in the output). A region with the same two boxes stays where it is.

This example moves the knitted knight up and to the left, and pins the backdrop, the cliff, the needle, the dragon, and the fire:

```
from diffusers.utils import load_image
```

```
knight_image = load_image( "knight_orig.png" ) edit_prompt = ( "In <ref_image_0>, move the miniature grey amigurumi knight figure <knight_1> upwards and to the left along the " "yarn cliff <cliff_1>. Keep the rest of the image completely unchanged, preserving the blurred yarn backdrop " "<backdrop_1>, the large knitted dragon <dragon_1>, and the fiery orange thread <fire_1>. " [{"id": "knight_1", "from": "ref_image_0", "src_bbox": [500, 150, 850, 350], "tgt_bbox": [194, 55, 544, 255], ' "desc": "A miniature amigurumi knight crocheted from thick grey woolen yarn."}, ' {"id": "backdrop_1", "from": "ref_image_0", "src_bbox": [0, 0, 1000, 1000], "tgt_bbox": [0, 0, 1000, 1000], ' "desc": "A soft, out-of-focus backdrop of cream and indigo knitted yarn."}, ' {"id": "cliff_1", "from": "ref_image_0", "src_bbox": [300, 0, 1000, 450], "tgt_bbox": [300, 0, 1000, 450], ' "desc": "A steep cliff of stacked yarn skeins in mustard, terracotta, and indigo."}, ' {"id": "needle_1", "from": "ref_image_0", "src_bbox": [420, 250, 550, 450], "tgt_bbox": [420, 250, 550, 450], ' "desc": "An oversized polished steel tapestry needle."}, ' {"id": "dragon_1", "from": "ref_image_0", "src_bbox": [100, 600, 950, 1000], "tgt_bbox": [100, 600, 950, 1000], ' "desc": "A large, fluffy knitted dragon in terracotta and mustard yarn."}, ' {"id": "fire_1", "from": "ref_image_0", "src_bbox": [350, 400, 650, 650], "tgt_bbox": [350, 400, 650, 650], ' "desc": "Curled puffs of fiery orange angora thread suspended in mid-air."}]' )
```

```
moved = pipe(prompt=edit_prompt, image=knight_image, generator=torch.Generator().manual_seed( 0 ), output= "images" )[ 0 ]
```

The knight moves from its src_bbox to its tgt_bbox , and the rest of the scene stays put

Base model and quantized variants

Base model

Everything above uses the distilled checkpoint. The base checkpoint runs 50 steps with classifier-free guidance, and it's the one to fine-tune (see LoRA finetuning).

```
pipe = ModularPipeline.from_pretrained( "black-forest-labs/flux-3-image-klein-base" )
pipe.load_components(dtype={ "vae" : torch.float32, "default" : torch.bfloat16}) pipe.to( "cuda" )

pipe.guiders = pipe.guiders.new(guidance_scale= 5.0 ) # optional image = pipe(prompt=prompt, output= "images" )[ 0 ]
```

Single-file format checkpoints

Black Forest Labs also ships the transformer as a single file, in bf16 and in two quantized formats: fp8r (rowwise FP8) and nvfp4 . Load any of them with from_single_file , then load the other components by name so the bf16 transformer isn't downloaded too.

```

from diffusers import BFLQuantizationConfig, Flux3Transformer2DModel, ModularPipeline

model_id = "black-forest-labs/flux-3-image-klein-base" variant = "fp8r" # or "nvfp4"

transformer = Flux3Transformer2DModel.from_single_file( f"https://huggingface.co/ {model_id} /blob/main/
variants/ {variant} /model.safetensors" , config=model_id, subfolder= "transformer" ,
quantization_config=BFLQuantizationConfig(compute_dtype=torch.bfloat16), dtype=torch.bfloat16, )

pipe = ModularPipeline.from_pretrained(model_id) pipe.load_components(dtype=torch.bfloat16)
pipe.update_components(transformer=transformer) pipe.to( "cuda" )

```

For the unquantized single file, use https://huggingface.co/{model_id}/blob/main/model.safetensors and drop `quantization_config` .

Variant Fast kernels on Other GPUs

fp8r Compute capability 8.9+ (Ada, Hopper, Blackwell) Weights are dequantized each step: lower memory, no speedup

nvfp4 Blackwell, with flashinfer-python and flashinfer-cubin at the same version Same as above

LoRA finetuning

FLUX.3 works for both text-to-image and image-to-image, which makes it a great base for custom LoRAs. diffusers ships DreamBooth LoRA training scripts for both tasks (you can also train with Ostris' AI Toolkit).

Here is a text-to-image example that learns a tarot-card style from the multimodalart/1920-raider-waite-tarot-public-domain dataset, triggered with `trtrcd` :

```

accelerate launch train_dreambooth_lora_flux3_klein.py \
--pretrained_model_name_or_path= "black-forest-labs/flux-3-image-klein-base" \
--dataset_name= "multimodalart/1920-raider-waite-tarot-public-domain" \
--caption_column= "caption" \
--instance_prompt= "trtrcd" \
--output_dir= "trained-flux3-tarot-lora" \
--mixed_precision= "bf16" \
--resolution=1024 \
--use_aspect_ratio_buckets \
--train_batch_size=1 \
--rank=16 \
--lora_alpha=16 \
--learning_rate=1e-4 \
--lr_scheduler= "constant" \
--max_train_steps=1500 \
--validation_prompt= "a trtrcd of a cat, tarot style" \
--validation_epochs=25 \
--seed=0 \
--push_to_hub

```

To reduce memory usage, add a few optimization flags:

```
--gradient_checkpointing \
--cache_latents \
--use_8bit_adam
```

`--gradient_checkpointing` recomputes activations in the backward pass (lower VRAM, slightly slower), `--cache_latents` pre-encodes the images and frees the VAE, and `--use_8bit_adam` keeps the optimizer states in 8-bit.

A couple of knobs are worth experimenting with. The grid below trains the same tarot style with different LoRA ranks and optimizers; the adapters are on the Hub at [diffusers-modular/flux3-tarot-lora](#) . More training features and configurations are documented in the DreamBooth FLUX.3 README .

For instruction-based editing, the image-to-image script (`train_dreambooth_lora_flux3_klein_img2img.py`) trains on (reference image, target image, instruction) triplets, where the reference is addressed in the prompt as `<ref_image_0>` .

See the DreamBooth FLUX.3 README for all options and the full image-to-image example.

What it costs to run

In this section, we will talk about the memory consumption and the speed of generation of the models.

Memory

name peak allocated (after load) GiBs peak allocated (after inference) GiBs

nvfp4 (RTX6000) 13.4 15.3

fp8r (H100) 14.35 16.27

bf16 (H100) 18.80 20.72

If you have less memory:

Load a quantized transformer .

Run the text encoder separately and free it. It only runs once per image, before denoising (see Encode once, generate many).

Turn on automatic CPU offloading, which moves each component to the GPU only while it runs (we didn't benchmark this)

```
from diffusers import ComponentsManager, ModularPipeline
```

```
manager = ComponentsManager() manager.enable_auto_cpu_offload(device= "cuda" )
```

```
pipe = ModularPipeline.from_pretrained( "black-forest-labs/flux-3-image-klein" , components_manager=manager)  
pipe.load_components(dtype={ "vae" : torch.float32, "default" : torch.bfloat16}) # no pipe.to("cuda"): the manager  
handles device placement
```

Speed

End-to-end timings for the distilled checkpoint (16 steps), averaged over three runs after a warmup:

Workflow	Output size	Time	Peak memory
----------	-------------	------	-------------

Text-to-image	1024 × 1024	1.93 s	23.8 GiB
---------------	-------------	--------	----------

Image-to-image, one 1024 × 1024 reference	1024 × 1024	3.87 s	23.8 GiB
---	-------------	--------	----------

Layout editing (the knight), one 1360 × 768 reference	1360 × 768	4.10 s	23.8 GiB
---	------------	--------	----------

Single NVIDIA H100 80GB, no offloading, default attention backend.

A reference image roughly doubles the time. Reference tokens join the same sequence as the text and the image being generated, so a 1024 × 1024 reference adds 4,096 tokens to every transformer call

Stacked bars of tokens per transformer call: text-to-image 4,256 tokens; image-to-image 8,272; layout text-to-image 4,336; layout image-to-image 8,800

The architecture

Let's open the model up, one piece at a time. The model takes three kinds of tokens:

text

the noisy image being generated

and optional reference images

From pixels to tokens

FLUX.3 is a latent-space flow model for image generation. This means that the denoiser takes latents as inputs and produces latents as outputs. We use a VAE to decode the latents back to pixels. FLUX.3 uses the same VAE architecture as Flux.2.

The FLUX.2 VAE compresses an image 8× in each direction into 32 channels, and then every 2 × 2 patch of that latent is packed into a single token with 128 channels.

So one token covers a 16 × 16 pixel patch , and a 1024 × 1024 image becomes a 64 × 64 grid of 4,096 tokens. The noisy image starts as random noise of exactly that shape. A reference image goes through the same VAE encoder and packing, and keeps its own size.

From prompt to tokens

The text encoder is Qwen3-VL, a vision-language model. FLUX.3 doesn't just take its last layer. It reads the hidden states of eight of its 36 layers (layers 4, 8, ..., 32) and stacks them, so each text token becomes a 8 × 2560 = 20,480-dimensional vector before a linear layer projects it to the transformer's width of 3072.

This gives the transformer features from several depths of the language model, not just its final layer. The prompt is padded to a multiple of 80 tokens, and the transformer attends to the padding too, which is why the pipeline

takes one prompt per call.

Two stages

The transformer processes the tokens in two stages:

Stage 1: per-mode blocks. Each stream goes through 5 blocks with its own weights. Text has its own blocks, the noisy image has its own, and the reference images have their own.

Stage 2: joint blocks. The streams are concatenated into one sequence, [text | image | reference] , which goes through 28 blocks that share one set of weights.

The two stages differ in who can attend to whom .

In stage 1, each stream only attends to itself, so each kind of token gets refined on its own terms first. In stage 2, everything attends to everything, where the image learns from the text and from the references. For comparison, FLUX.2's double-stream blocks also give text and image separate weights, but let them attend to each other from the start. FLUX.3 keeps the streams fully apart first, then merges them.

At the end, the text and reference tokens are simply dropped. Only the image tokens go through the output layers, which predict how to move them one step toward a clean image.

Inside one block

Every block, in either stage, has the same shape. Attention and the MLP run in parallel on the same normalized input, and their sum is added back to the residual stream.

Each block starts with a LayerNorm that has no learned scale or shift of its own. The scale, shift, and gate come from a timestep embedding instead (this is called modulation). Inside the attention, the queries and keys are RMS-normalized and get rotary position embeddings (RoPE), which we look at below.

In a joint block, all tokens share the same weights, but text, image, and reference tokens each get their own scale, shift, and gate.

```
def block ( x, shift, scale, gate ): # shift, scale, gate come from the token's timestep embedding
h = layer_norm(x)
# no learned scale or shift
h = ( 1 + scale ) * h + shift
q, k, v = to_q(h), to_k(h), to_v(h)
q, k = rope(rms_norm(q)), rope(rms_norm(k))
out = attention(q, k, v) + mlp(h) # parallel, MLP is SwiGLU
return x + gate * out
```

Every token gets its own timestep

In FLUX.1 and FLUX.2, one timestep embedding modulates every token. In FLUX.3, the modulation depends on the kind of token.

The noisy image tokens are modulated by the current timestep t . The text and reference tokens are always modulated by timestep 0 , which means "clean, nothing to denoise here". So even a reference that looks almost the same as the output is marked as given context, not as something to generate.

The distilled checkpoint is a flow map . It is also told the timestep each step jumps to. That second timestep gets its own embedding, which is added to every token's modulation, so each of its 16 steps can make a larger jump.

Positions in four dimensions

Attention on its own doesn't know where a token is (this phenomenon is called permutation equivariance). RoPE fixes that by rotating each query and key by an amount that depends on the token's position. FLUX.3 uses four position axes, (t, h, w, l) (timestep, height, width, length of sequence), and gives each one a quarter (32 dimensions) of every 128-dimensional attention head:

Tokens t: timestep h: height w: width l: length of sequence

text token i 0 0 0 i

image token at (row, col) 0 row / 2 col / 2 0

reference k at (row, col) 0 row / 2 col / 2 -(k + 1)

Text tokens line up along l , like words in a sentence. The image and the references are 2D grids in (h, w) , separated from each other along l . Because a reference uses the same (h, w) coordinates as the output, "the

same spot in the reference" has a precise meaning to the model, which is what the `src_bbox` / `tgt_bbox` editing relies on.

The denoising loop

FLUX.3 is a flow-matching model. Generation starts from pure noise, and at each of the 50 steps the transformer predicts a velocity. This is the direction from noise toward a clean image. The scheduler takes one step along it.

At every step you can also read off the model's current guess of the final image. Decoding that guess at a few steps shows how the image takes shape:

Step 1 Step 5 Step 10 Step 15 Step 25 Step 50

The fox, its pose, and the light are already in place by step 10, while the noise level is still 0.97. The snow-laden branches and the texture of the fur only appear in the last few steps. FLUX.3's scheduler uses a large time shift (6.93), which keeps the noise level high for most of the run: 44 of the 50 steps happen above $\sigma = 0.5$. We built this figure by adding one small block to the pipeline, which brings us to Modular Diffusers.

If you are new to flow matching, we have a beginner friendly tutorial and a video for you to understand how it works.

Modular Diffusers

FLUX.3 ships as a Modular Diffusers pipeline.

See the workflows

You can list the workflows and print the exact blocks of each one, without loading any weights.

```
from diffusers import Flux3AutoBlocks
```

```
blocks = Flux3AutoBlocks() print (blocks.available_workflows) # ['text2image', 'image_conditioned']  
  
print ( list (blocks.get_workflow( "text2image" ).sub_blocks)) # ['text_encoder', 'denoise.input',  
'denoise.prepare_latents', 'denoise.set_timesteps', # 'denoise.prepare_rope_inputs', 'denoise.denoise',  
'denoise.after_denoise', 'decode']
```

`get_workflow()` resolves the auto branches and gives you the plain sequence of steps that will run.
`print(blocks.get_workflow("image_conditioned").doc)` prints the components, inputs, defaults, and outputs of that workflow.

Encode once, generate many

Pop the text encoder out, encode the prompt once, free it, and generate from the cached embeddings.

```
import torch from diffusers import ModularPipeline
```

```
model_id = "black-forest-labs/flux-3-image-klein-base"
```

```
blocks = ModularPipeline.from_pretrained(model_id).blocks text_blocks = blocks.sub_blocks.pop( "text_encoder" )  
# `blocks` now starts at the VAE encoder
```

```
text_pipe = text_blocks.init_pipeline(model_id) text_pipe.load_components(dtype=torch.bfloat16) text_pipe.to(  
"cuda" )
```

```
state = text_pipe(prompt=prompt, negative_prompt= "" ) # no guider here, so ask for the negative embeddings
```

```
prompt_embeds = state.get( "prompt_embeds" ) negative_prompt_embeds = state.get(  
"negative_prompt_embeds" )
```

```
del text_pipe torch.cuda.empty_cache()
```

```
image_pipe = blocks.init_pipeline(model_id) # no text encoder to load image_pipe.load_components(dtype={  
"vae" : torch.float32, "default" : torch.bfloat16}) image_pipe.to( "cuda" )
```

```
for seed in range ( 3 ): image = image_pipe( prompt_embeds=prompt_embeds,  
negative_prompt_embeds=negative_prompt_embeds, generator=torch.Generator().manual_seed(seed), output=  
"images" , )[ 0 ] image.save( f" {seed} .png" )
```

Add a step inside the loop

The denoising figure above came from one extra block that records the model's guess of the clean image at every step:

```
from diffusers.modular_pipelines import Flux3AutoBlocks, InputParam, ModularPipelineBlocks, OutputParam

class Flux3PreviewStep ( ModularPipelineBlocks ): model_name = "flux3"

@property def inputs ( self ): return [InputParam( "latents" ), InputParam( "noise_pred" )]

@property def intermediate_outputs ( self ): return [OutputParam( "x0_previews" , type_hint= list )]

@torch.no_grad() def __call__ ( self, components, block_state, i, t ): if i == 0 : block_state.x0_previews = [] sigma
= t / 1000 # timesteps are sigmas scaled to [0, 1000] block_state.x0_previews.append(block_state.latents - sigma
* block_state.noise_pred) # x0 = x_t - sigma * v return components, block_state

blocks = Flux3AutoBlocks() denoise_loop = blocks.sub_blocks[ "denoise" ].sub_blocks[ "text2image" ].sub_blocks[
"denoise" ] denoise_loop.sub_blocks.insert( "preview" , Flux3PreviewStep(), 1 ) # between "denoiser" and
"after_denoiser"

pipe = blocks.init_pipeline(model_id) pipe.load_components(dtype={ "vae" : torch.float32, "default" :
torch.bfloat16}) pipe.to( "cuda" )

out = pipe(prompt=prompt, generator=torch.Generator().manual_seed( 0 ), output=[ "x0_previews" , "images" ])
```

Try it

Models: [FLUX.3 Image Klein collection](#)

Docs: [FLUX.3 in Diffusers](#)

Fine-tune: [DreamBooth LoRA scripts](#)

Prompting: [Black Forest Labs prompting guide](#)

Built something with FLUX.3? Share your LoRAs and pipelines on the Hub. We'd love to see them! 🥳

Models mentioned in this article 3

Datasets mentioned in this article 1

Spaces mentioned in this article 2

Collections mentioned in this article 1

More Articles from our Blog

[diffusers flux quantization](#)

[Diffusers welcomes FLUX-2](#)

+4

191 November 25, 2025

[guide diffusers quantization](#)

[\(LoRA\) Fine-Tuning FLUX.1-dev on Consumer Hardware](#)

+1

108 June 19, 2025

[Community](#)

[Edit Preview](#)

Upload images, audio, and videos by dragging in the text input, pasting, or clicking here . Tap or paste here to upload images

Comment · Sign up or log in to comment

Upvote 37

+25

Models mentioned in this article 3

Datasets mentioned in this article 1

Spaces mentioned in this article 2

Collections mentioned in this article 1