Enhancing Visual Captioning: Comparing LoRA Fine-Tuning and Few-Shot Prompting on the PixelProse Dataset

Kyle Merritt
University of Central Florida
CAP 5415: Computer Vision

Fall 2025

Abstract

Modern vision–language models can generate fluent image captions but often default to short, generic descriptions that underspecify relationships, context, and fine-grained details. This project investigates two strategies for improving detailed captioning: (1) parameter-efficient fine-tuning using Low-Rank Adaptation (LoRA) on a subset of the PixelProse dataset, and (2) multimodal few-shot prompting that conditions the model on exemplar image–caption pairs at inference time. Using the open-source InternVL 3.5 2B model as a base, I compare six configurations that vary both model state (pretrained versus LoRA-fine-tuned) and prompting style (neutral, detailed, and detailed with multimodal few-shot exemplars). Quantitative evaluation on 500 held-out PixelProse images uses BLEU, METEOR, CIDEr, BERTScore, and caption length. Results show that multimodal few-shot prompting substantially improves similarity to PixelProse captions for the pretrained model, while the limited LoRA fine-tuning run used in this study yields only modest additional gains. Combining LoRA with few-shot prompting degrades performance and produces repetitive, off-target captions, which I attribute to a mismatch between the single-image training prompt and the multi-image few-shot evaluation prompt. I also measure the additional VRAM cost of multimodal few-shot prompting and discuss the trade-off between inference-time memory, training cost, and caption quality.

1 Introduction

Dense, descriptive image captioning is important for applications such as assistive technologies, visual storytelling, and downstream reasoning in multimodal agents. In practice, however, many off-the-shelf vision–language models (VLMs) still produce short, generic captions that under-describe complex scenes (for example, "a woman wearing a jacket" instead of specifying pose, attributes, and context). Improving caption richness can be approached either by modifying the model itself through fine-tuning or by reformulating the prompts used at inference time.

In this project I investigate these two strategies in the context of InternVL 3.5 2B, a recent open-source multimodal model, on the PixelProse dataset of dense captions. The first strategy is using Low-Rank Adaptation (LoRA), a parameter-efficient fine-tuning method that keeps all original Transformer weights frozen and injects small trainable low-rank matrices into existing attention and MLP projection layers, allowing the model to specialize to PixelProse-style captions without modifying or adding full layers. The second strategy is multimodal few-shot prompting: I prepend exemplar image-caption pairs to the model's input so that it can condition on several high quality examples before describing a new query image.

The original project proposal envisioned training on millions of PixelProse examples with multi-GPU data-parallel fine-tuning and then running a large ablation over LoRA ranks, frozen-layer depths, and several few-shot prompting schemes. In practice, I was constrained to single-GPU training and a much smaller subset of the dataset, and I evaluated each configuration on 500 held-out images instead of the larger planned study. Despite these limitations, the experiment still answers three central questions:

- How effective is multimodal few-shot prompting for improving descriptive caption quality in this domain?
- How much additional benefit does LoRA fine-tuning provide beyond prompting when training is performed on a relatively small subset?
- Do fine-tuning and few-shot prompting combine synergistically, or can mismatched training and inference conditions actually hurt performance?

2 Related Work

2.1 Dense Captioning and PixelProse

Traditional captioning datasets such as MSCOCO emphasize short alt-text style captions that summarize only the most salient objects in an image. In contrast, PixelProse [3] introduces more than 16 million image-caption pairs featuring long, dense, paragraph-level descriptions intended to capture fine-grained attributes, relationships, and scene structure. Each record also includes metadata such as aesthetic scores and watermark probabilities, enabling quality-based filtering and analysis. Because PixelProse captions are significantly longer and more detailed than standard alt-text, the dataset provides a natural benchmark for studying models that aim to generate richly descriptive prose.

2.2 InternVL 3.5

InternVL 3.5 [4] is an open-source family of multimodal models that integrate a vision encoder with a large language model. The 2B-parameter variant used in this project is small enough for single- or dual-GPU setups while still supporting multi-image reasoning, conversation-style interactions, and flexible input formats. InternVL employs dynamic image tiling to handle high-resolution or non-square images by splitting them into patches, encoding each patch through its vision backbone, and injecting learned image tokens into the language model. This architecture enables the model to process varying image sizes without retraining.

2.3 LoRA Fine-Tuning

Low-Rank Adaptation (LoRA) [2] is a parameter-efficient fine-tuning method that freezes the original model weights and injects trainable low-rank matrices into targeted linear layers. Instead of updating full projection or feed-forward matrices, LoRA learns small rank-r updates that are added to the existing weights at inference time. This approach drastically reduces memory footprint and training compute, making it well suited for adapting large models on commodity hardware. In this project, LoRA is applied to the transformer blocks of the language model component of InternVL by targeting key attention and MLP projection layers; only the LoRA parameters are updated while the base model remains unchanged.

2.4 Few-Shot Prompting for Vision–Language Models

Few-shot prompting been used for years in large language models by providing example input-output pairs directly within the prompt. This idea extends naturally to multimodal models by including paired image-caption exemplars. Flamingo [1] showed that large pretrained vision-language models can achieve strong few-shot performance on captioning, VQA, and multimodal reasoning tasks simply by concatenating exemplar image-caption pairs with the query input, without any parameter updates. Such prompting enables rapid style conditioning, such as inducing longer, more descriptive captioning behavior, but increases inference time VRAM usage because all exemplar images must be embedded and stored in the attention context.

In this project, we evaluate whether multimodal few-shot prompting can effectively guide InternVL toward producing PixelProse-style long-form captions and compare its benefits and costs to those of LoRA fine-tuning.

3 Method

3.1 Dataset and Subset Construction

The full PixelProse dataset is too large to train on exhaustively within a single semester project, so I constructed several smaller subsets using a dedicated script (`scripts/build_pixelprose_subset.py`). The script downloads images and metadata from the Hugging Face PixelProse dataset, shuffles with a fixed seed, filters out failed downloads, and writes each subset to a directory of the form:

`data/pixelprose_subset{index}/`

Each subset contains:

- An `images/` directory with JPEG files named `000000.jpg`, `000001.jpg`, ...
- A `metadata.jsonl` file with one JSON record per image, including fields such as `id`, `url`, `image_file`, `caption`, `vlm_model`, aesthetic score, and watermark score.

For the main experiment I used **subset 2** (`data/pixelprose_subset2`), which contains 2,000+ images. I reserved all examples with `id` < 1000 for evaluation and used `id` ≥ 1000 for LoRA training and validation. The LoRA training script further splits this usable pool into a training set and a validation set using a fixed, deterministic partition.

In practice, due to runtime and memory constraints, I trained on a much smaller number of images than originally proposed. The final LoRA run used approximately 20,000 captioned examples sampled from the training split through multiple epochs, and I evaluated each configuration on 500 held-out images from the evaluation pool (except where otherwise noted).

3.2 Base Model and Image Preprocessing

All experiments use the `OpenGVLab/InternVL3_5-2B-Instruct` checkpoint. Images are preprocessed using the dynamic tiling pipeline implemented in the load image function within `src/test_internvl_caption.py`, which follows the official InternVL preprocessing strategy. The steps are:

1. Load the image from disk using PIL and convert it to RGB.
2. Use `dynamic_preprocess` to select an aspect-ratio–appropriate grid of tiles, producing up to 12 base patches; if more than one tile is used, a 448×448 thumbnail patch is appended, giving a maximum of 13 patches.
3. Resize each patch to 448×448 using bicubic interpolation and normalize using ImageNet mean and standard deviation.
4. Stack the patches into a tensor of shape $[N_{\text{patch}}, 3, 448, 448]$.

This preprocessing function is shared across both fine-tuning and evaluation, ensuring that the visual input distribution is consistent at training and test time.

3.3 LoRA Configuration and Training Pipeline

LoRA fine-tuning is implemented in `src/train_lora.py`. The script loads the full `OpenGVLab/InternVL3_5-2B-Instruct` checkpoint and applies a LoRA adapter to the `language_model` component inside the InternVL architecture. The configuration is:

- Rank $r = 32$, LoRA scaling $\alpha = 64$, dropout $= 0.05$.
- Target modules: the query, key, value, and output projections (`q_proj`, `k_proj`, `v_proj`, `o_proj`), as well as the gate, up, and down projections in the feed-forward layers (`gate_proj`, `up_proj`, `down_proj`).
- Task type: causal language modeling, with cross-entropy loss over the full token sequence (image tokens, prompt, and caption).

The training dataset class `PixelProseLoraDataset` constructs each example from a single image and its PixelProse caption. It uses the same detailed prompt as the evaluation script found in: (`prompts/detailed.txt`):

> *"Provide a detailed, comprehensive caption describing all key objects, attributes, actions, and context."*

For each training sample, the multimodal input fed to the model is built in `build_mm_inputs_for_batch`: a block of special image tokens <img> ...<IMG_CONTEXT> ...</img> is generated, where the <IMG_CONTEXT> token is repeated in proportion to the number of image patches for that example. This image-token block is followed by the detailed prompt and finally the ground-truth caption. No few-shot exemplars are included during training; every example consists of a single query image and its caption.

The collate function concatenates all image patches in a batch into a single tensor and tracks the number of patches per sample as `num_patches_list`. This allows the training loop to (i) move all patches onto the GPU as one tensor and (ii) compute the correct number of visual tokens for each query when constructing the text sequence.

All experiments were run on a single RTX 3090 GPU. The training script is configured for single-GPU operation (`device_map = None`) and uses both a limited number of epochs and a hard cap on the total number of optimization steps (`max_steps`) to keep runs within the available compute budget. As a result, the LoRA adapter in this project should be viewed as a relatively small fine-tuning run rather than a fully converged model. I attempted to enable multi-GPU LoRA fine-tuning using HuggingFace Accelerate, but InternVL's multimodal architecture cannot be automatically sharded across devices without modifying the underlying model; because the model does not implement HF's parallelization hooks and contains custom cross-modal injection layers, multi-GPU training was not feasible within the project window.

By default, the adapter weights are saved under `checkpoints/internvl3_5_2b_lora_pixelprose/subset2_r32_a64/`. In my experiments, I also produced a variant checkpoint with an explicit training-size suffix (`subset2_r32_a64_train_size_20000`). The evaluation script `src/eval_captions.py` loads the LoRA adapter path (either the default or the suffixed directory) when running the fine-tuned (FT) configurations.

3.4 Prompting Configurations

Prompting is controlled by three instruction templates stored in the `prompts/` directory:

- **Neutral (N):** "Provide a factual caption for this image."
- **Detailed (D):** "Provide a detailed, comprehensive caption describing all key objects, attributes, actions, and context."
- **Detailed Few-Shot (D_FS):** "Provide a detailed, comprehensive caption describing all key objects, attributes, actions, and context in the query image. Use the exemplar image–caption pairs above as a guide for the level of detail and style, but focus entirely on the query image."

The main evaluation script `src/eval_captions.py` supports both pretrained (PT) and fine-tuned (FT) models and can evaluate multiple prompting methods on the same set of images. For this project, I focused on the following six configurations:

Model state	Prompting method
PT	N (neutral)
PT	D (detailed)
PT	D_FS (detailed + few-shot)
FT	N (neutral)
FT	D (detailed)
FT	D_FS (detailed + few-shot)

Few-shot prompting uses multimodal exemplars drawn from the same subset. For each run, I select a small, fixed set of exemplar records from the metadata file and log them to `logs/eval_subset{idx}_{PT/FT}_exemplars.jsonl`. To avoid test leakage, I ensure that these exemplar IDs are excluded from the evaluation pool and verify this with a dedicated script (`scripts/check_exemplar_leakage.py`).

In the multimodal few-shot setup, I prepend K exemplar image–caption pairs (typically $K = 3$) before the query image. At evaluation time, the text prompt looks conceptually like:

```
Exemplar-1:  <image>
Caption-1:  [caption text]
...
Query:  <image>
[detailed few-shot instruction]
```

This design allows the model to see multiple PixelProse-style captions before generating a description for the query image.

The three exemplar image caption pairs that were used in the few shot prompting configuration are shown below.

Exemplar 1 (outfit image, ID 0):

This image displays two sets of outfits, each containing a black T-shirt, a blue blazer, and a pair of blue jeans. The T-shirts have a white logo on them, reading: “S.T.A.R. Labs.” The blazers are both a deep royal blue color. The left blue blazer has a white lightning bolt emblem on the left sleeve. The right blazer has a gold emblem of an atom on the left sleeve. Both pairs of jeans are light blue and slightly distressed. The left outfit is paired with a pair of red high-top Converse sneakers, and a red purse with black edges and handles. The right outfit is paired with a pair of black high-top Converse sneakers and a silver necklace with a lightning bolt pendant. The background of the image is a white square.

Metadata:

Aesthetic score: 5.30 Watermark score: 0.83

Role in prompts:

This image and caption pair is used as the first exemplar in the multimodal few-shot prompt (PT_D_FS and FT_D_FS). It provides a long, style-rich description of a fashion-oriented image, demonstrating the level of detail and structure that the model is encouraged to imitate.

Figure 1: Exemplar 1 used in the multimodal few-shot prompt: a detailed fashion style image and caption pair.

Exemplar 2 (crest image, ID 1):

This image displays:
A crest with a shield divided into four sections. A lion, a snake, a badger, and a raven are in each section, respectively. The letters “H” and “P” are in the middle of the crest. A banner with the words “Draco Dormiens Nunquam Titillandus” is at the bottom. The crest has a gold background and a gold frame with flourishes.

Metadata:

Aesthetic score: 5.61 Watermark score: 0.90

Role in prompts:

This is the second exemplar in the multimodal few-shot prompt. It shows that PixelProse style captions can include emblematic or symbolic content with dense lexical detail, including text, heraldry elements, and layout information.

Figure 2: Exemplar 2 used in the multimodal few-shot prompt: a detailed crest image and caption pair.

Exemplar 3 (cartoon quote image, ID 2):

This image displays a pink background with black text. The text reads: "I just want to lose weight while staying in bed, watching TV, and eating Girl Scout cookies. Is that really too much to ask?" The text is in a white speech bubble on the left side of the image. On the right side of the image is a cartoon drawing of a person sitting on a couch with polka dots. The person is wearing pajamas and has their head in their hands. There is a box of Girl Scout cookies on the floor in front of the couch. The image is drawn in a simple cartoon style.

Metadata:

Aesthetic score: 5.18 Watermark score: 0.85

Role in prompts and failure analysis:

This is the third exemplar in the multimodal few-shot prompt. Its caption is later mistakenly reproduced almost verbatim by the FT + D_FS configuration when captioning a different query image (Figure 9), illustrating a clear case where the fine-tuned model copies an exemplar instead of describing the query.

Figure 3: Exemplar 3 used in the multimodal few-shot prompt: a long, stylized caption for a cartoon quote image. This exemplar is directly involved in the exemplar confusion failure shown in Figure 9.

Table 1: Quantitative results on 500 held-out PixelProse images from subset 2 (451 for FT + D_FS). Average lengths are in tokens.

Model	Prompt	#	BLEU	METEOR	CIDEr	BERTScore
PT	N	500	3.87	18.66	0.09	40.25
PT	D	500	8.80	28.97	0.18	43.20
PT	D_FS	500	**10.08**	**33.05**	**0.19**	**47.05**
FT	N	500	8.30	32.12	0.04	42.67
FT	D	500	8.69	33.41	0.04	43.86
FT	D_FS	451	7.06	30.60	0.01	38.58

3.5 Evaluation Protocol and Metrics

For each configuration, the evaluation script runs InternVL on up to 500 held-out images from `pixelprose_subset2`, producing logs of the form:

```
logs/eval_subset2_{PT/FT}_{N,D,D_FS}.jsonl
```

Each record includes the image ID, file name, ground-truth caption, prompt type, model tag (PT or FT), and the generated prediction.

Automatic metrics are computed by `scripts/compute_metrics.py`, which aggregates:

- BLEU (via `sacrebleu`),
- METEOR (via NLTK),
- CIDEr (via a Hugging Face `evaluate` implementation),
- BERTScore (F1, rescaled to 0–100),
- Average ground-truth and predicted caption lengths (in tokens).

The main summary for subset 2 and the PT/FT configurations is stored in `metrics/metrics_subset2_PT_FT.csv`. For most configurations I obtained results on all 500 examples. The FT + D_FS configuration was ended early because multiple repetitions and incorrect descriptions were observed, resulting in 451 evaluated samples.

In addition to quantitative metrics, I manually reviewed generated captions to identify typical successes and failure modes. To make these qualitative differences visible, I prepared galleries pairing each image with its PixelProse caption and the generated captions under several configurations. Section 4.3 describes how these galleries are organized in the report.

4 Experiments and Results

4.1 Quantitative Results

Table 1 summarizes the main automatic metrics for the six configurations on subset 2. All metrics are reported on a 0–100 scale for easier comparison.

Several trends are immediately apparent:

- For the pretrained model, simply switching from the neutral prompt (PT + N) to the detailed prompt (PT + D) yields a large improvement across all metrics. Average predicted caption length increases from roughly 40 tokens to nearly 100 tokens, and BLEU, METEOR, CIDEr, and BERTScore all improve.
- The few shot prompting configuration (PT + D_FS) produces the best overall scores for the pretrained model. METEOR and BERTScore in particular increase noticeably, suggesting that the model's style and content become more aligned with PixelProse captions when it sees exemplar image-caption pairs.

- LoRA fine-tuning with the neutral prompt (FT + N) dramatically increases average caption length (often exceeding 190 tokens) and provides higher METEOR than PT + N. However, CIDEr remains low, indicating that the increased verbosity does not always translate into better n-gram agreement with the ground truth.
- The fine-tuned model with the detailed prompt (FT + D) performs similarly to PT + D in terms of BLEU and METEOR, with slightly higher BERTScore but still modest gains compared to the impact of multimodal few-shot prompting.
- Surprisingly, combining LoRA fine-tuning with detailed few-shot prompting (FT + D_FS) *hurts* performance. All metrics drop relative to PT + D_FS, and BERTScore decreases substantially. Qualitative inspection reveals that this configuration often produces repetitive or off-topic captions and occacionally copies an examplar answer.

Overall, on this training budget, multimodal few-shot prompting provides the clearest benefit over the base model, whereas the small-scale LoRA fine-tuning run yields only modest additional improvements and interacts poorly with few-shot prompting.

4.2 VRAM Usage and Computational Cost

One of the original goals of this project was to compare not only caption quality but also computational cost. Although I did not perform a full runtime and energy audit, I did track GPU memory usage during key configurations.

When using multimodal few-shot prompting with three exemplar images plus one query image, InternVL must process all patches from four images in a single forward pass. On my hardware, this configuration used approximately 50 GB of VRAM across the 4x RTX 3090 system. This is significantly higher than the memory required for single-image captioning or for running the fine-tuned model without exemplars.

In contrast, LoRA fine-tuning itself can be performed with more modest per-batch memory (once the model is loaded), and the resulting fine-tuned model can be deployed with the same inference-time memory footprint as the pretrained model, as long as no few-shot exemplars are used. This highlights a key trade-off: few-shot prompting can be very effective at steering behavior but may be difficult to scale when memory is tight or when many images need to be processed in parallel.

4.3 Qualitative Galleries

To better understand how each configuration behaves beyond aggregate similarity metrics, I constructed a series of qualitative galleries. Each gallery focuses on a single image from the PixelProse subset and shows:

- The input image (top),
- The PixelProse reference caption, and
- The generated captions from five configurations: pretrained factual (PT + N), pretrained detailed (PT + D), pretrained detailed multimodal few-shot (PT + D_FS), fine-tuned factual (FT + N), and fine-tuned detailed (FT + D).

These galleries make it easier to see how prompting style and LoRA fine-tuning jointly affect caption length, style, and failure modes such as hallucination and repetition.

PixelProse (reference):

This image displays:
A woman wearing a white and gray zip-up jacket with a hood. The jacket has two front pockets. The hood is also gray on the inside. The woman is looking to the side, and away from the camera. The woman is standing with her arms at her side. She has a neutral, perhaps slightly happy expression on her face. The background is white without any other objects or scenery. The image is a photograph of a woman posing in front of a white background, and the style is fashion-oriented.

PT + N (pretrained, factual prompt):

A woman wearing a grey hoodie.

PT + D (pretrained, detailed prompt):

A woman wearing a grey hoodie and white pants.

PT + D_FS (pretrained, detailed few-shot):

This image displays a person wearing a light gray hoodie with a hood and white pants. The hoodie has a pattern of small, dark gray shapes on the front. The person is standing with their hands in their pockets, facing away from the camera. The background is plain white.

FT + N (fine-tuned, factual prompt):

A young woman is wearing a grey hoodie sweatshirt with a white drawstring. She is standing with her left hand in her hoodie pocket and her right hand hanging by her side. She is looking at the camera with a slight smile on her face. The background is a white wall. The image is a photograph taken from a slightly elevated angle. The woman is wearing a neutral expression on her face. The image is well lit and the colors are muted. The style of the image is casual.

FT + D (fine-tuned, detailed prompt):

This image displays a young woman wearing a grey hoodie sweatshirt. The sweatshirt has a hood and long sleeves. The woman is standing with her left hand in her hoodie pocket and her right hand hanging by her side. She is looking at the camera with a slight smile on her face. The background is white and there is a shadow on the left side of the woman. The image is a photograph taken from a slightly elevated angle. The woman is wearing a neutral expression on her face. The image is well lit and the colors are muted. The style of the image is casual and relaxed. *(The model begins to repeat this description later in the full output, indicating a repetition tendency after fine-tuning.)*

Figure 4: Qualitative gallery for ID 3 (woman in gray hoodie).

PixelProse (reference):

This image displays a stainless steel folding table. The table is made of stainless steel and has a foldable design. It is shown folded in half. The table is shown against a white background. The table is not being used, and there are no people present. The image is a photograph and is factual and terse but complete.

PT + N (pretrained, factual prompt):

The image shows a stainless steel industrial refrigeration unit with multiple compartments. It is designed for commercial use, likely in a food service or retail setting, to keep perishable items cold. The unit has a sturdy metal frame and a flat top surface.

PT + D (pretrained, detailed prompt):

The image features a stainless steel kitchen cart or cart with a drawer. The cart has a sturdy metal frame and a drawer that is open, revealing the interior of the cart. The drawer is made of stainless steel and has a handle on the side. The cart is designed for use in a kitchen or food preparation area, providing a convenient and functional storage solution.

PT + D_FS (pretrained, detailed few-shot):

This image displays a stainless steel industrial freezer with multiple compartments. The freezer is designed with a sturdy metal frame and features a sliding door mechanism. The compartments are divided into sections, each with a clear, reflective surface, likely for easy visibility of the contents. The freezer appears to be set up for commercial use, possibly in a food service or retail environment. The overall design suggests functionality and durability, suitable for storing perishable goods.

FT + N (fine-tuned, factual prompt):

A stainless steel kitchen cart with a countertop and a drawer. The drawer is open and the countertop is empty. The cart is on a white background. The image is a photograph taken from a top-down perspective. The cart is made of metal and has a simple, utilitarian design. The countertop is made of stainless steel and has a smooth, shiny surface. The drawer is made of stainless steel and has a simple, utilitarian design. *(The caption continues by repeating this description.)*

FT + D (fine-tuned, detailed prompt):

This is a stainless steel kitchen cart with a countertop and a drawer. The cart is made of metal and has a silver finish. The countertop is made of stainless steel and has a smooth surface. The drawer is made of stainless steel and has a hinged door. The cart is supported by four metal legs. The cart is empty and there is nothing on the countertop. The background is white and there is no scenery. The image is a 3D render and is not a photograph. *(The model then repeats the "3D render" statement multiple times.)*

Figure 5: Qualitative gallery for ID 4 (stainless steel folding table).

PixelProse (reference):

This image displays: a cowboy on a horse. The cowboy has a lasso in his hand and is wearing a hat. The horse is rearing up and has one front hoof off the ground. Both the cowboy and the horse are in mid-action. There is no background scenery. The image is a line drawing in black and white.

PT + N (pretrained, factual prompt):

A cowboy riding a horse, depicted in a dynamic pose, with the horse rearing up on its hind legs. The cowboy is wearing a hat and traditional attire, holding the reins as the horse appears to be in motion.

PT + D (pretrained, detailed prompt):

The image features a black and white line drawing of a cowboy riding a horse. The cowboy is depicted in a dynamic pose, with one arm extended and the other holding the reins. The horse is in mid-gallop, with its legs lifted off the ground, suggesting movement and speed. The cowboy's attire includes a hat, boots, and a long-sleeved shirt, typical of Western wear. The horse's mane and tail are flowing, adding to the sense of motion. The background is plain, focusing attention on the cowboy and horse. There are no additional objects or details in the image, making it a simple yet evocative representation of a classic Western scene.

PT + D_FS (pretrained, detailed few-shot):

The image is a black and white line drawing of a cowboy riding a horse. The cowboy is wearing a hat, a long-sleeved shirt, and jeans. He is holding the reins of the horse with one hand and has his other hand extended outward. The horse is depicted in mid-gallop, with its front legs lifted off the ground. The cowboy appears to be in motion, suggesting a sense of speed and action. The drawing is detailed, with clear lines defining the shapes and features of both the cowboy and the horse.

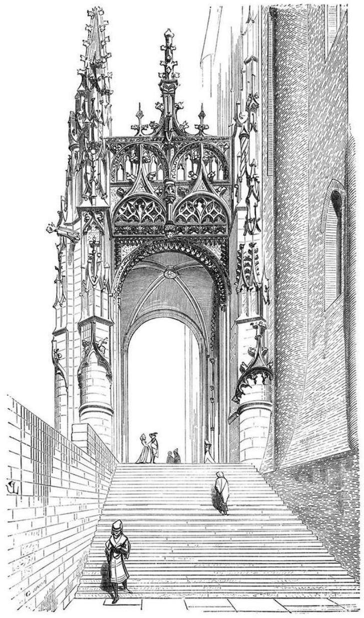
FT + N (fine-tuned, factual prompt):

A line drawing of a cowboy riding a horse. The cowboy is wearing a cowboy hat and a long-sleeved shirt. The horse is brown and has a saddle on its back. The cowboy is holding the reins in his right hand and is looking to the left. The horse is standing on all four legs and is in a galloping position. The background is white and there is no scenery. The drawing is in a cartoon style and is drawn in black and white. *(The caption then repeats similar sentences several times.)*

FT + D (fine-tuned, detailed prompt):

This is a line drawing of a cowboy riding a horse. The cowboy is wearing a cowboy hat, a long-sleeved shirt, and jeans. He is holding the reins in his right hand and is looking to the left. The horse is brown and has a saddle on its back. The horse is standing on all four legs. The background is white. The drawing is in a cartoon style. *(The remainder of the caption largely repeats this description, illustrating repetition after fine-tuning.)*

Figure 6: Qualitative gallery for ID 5 (cowboy on a horse, line drawing).



PixelProse (reference):

This image displays:
A black-and-white drawing of an ornate Gothic doorway. The doorway has two staircases leading up to it, and is flanked by two large buttresses. The doorway itself is topped by a gable, and is decorated with statues, pinnacles, and other intricate carvings.

PT + N (pretrained, factual prompt):

The image is a detailed black and white illustration of a grand Gothic-style archway, likely part of a cathedral or significant historical building. The archway features intricate carvings and ornate stonework, with pointed arches and elaborate tracery. A wide staircase leads up to the arch, with several figures depicted ascending and descending the steps. The scene captures the architectural grandeur and historical significance of the structure.

PT + D (pretrained, detailed prompt):

The image is a detailed black and white illustration of a grand, ornate Gothic-style archway, likely part of a cathedral or significant historical building. The archway is intricately decorated with pointed arches, pinnacles, and flying buttresses, characteristic of Gothic architecture. The scene is set on a wide staircase leading up to the archway, with several figures depicted in various poses, suggesting movement and activity. The figures appear to be dressed in period attire, indicating a historical setting. The overall composition conveys a sense of grandeur and historical significance, with attention to architectural detail and the depiction of human figures in a dynamic scene.

PT + D_FS (pretrained, detailed few-shot):

The image depicts a detailed black and white illustration of a grand, ornate Gothic-style archway. The arch is richly decorated with intricate carvings and pointed spires, characteristic of Gothic architecture. The structure features a large central arch flanked by two smaller arches, all adorned with elaborate tracery and statues. The steps leading up to the arch are wide and steep, with a few figures ascending and descending them. The figures are dressed in period clothing, suggesting a historical or fantasy setting. The overall composition conveys a sense of grandeur and historical significance.

FT + N (fine-tuned, factual prompt):

A black and white line drawing of a Gothic cathedral with a large entrance and a set of stairs leading up to it. There are three people on the stairs, two of them are standing and one is sitting on the steps. The cathedral is made of stone and has a large tower on the left side. The entrance to the cathedral is made of stone and has a large archway decorated with intricate carvings. *(Later in the full output, the model emphasizes repeatedly that the image is a drawing rather than a photograph.)*

FT + D (fine-tuned, detailed prompt):

This is a black and white line drawing of a Gothic cathedral. The cathedral has a large central nave with a pointed arch and a large rose window at the top. There are two towers on either side of the nave, each with a pointed roof and a large spire on top. The walls of the cathedral are made of stone and there are many small windows along the walls. There is a large staircase leading up to the cathedral from the bottom of the image, with three people on the stairs. *(The caption later repeats that the image is a drawing and not a photograph.)*

Figure 7: Qualitative gallery for ID 6 (ornate Gothic doorway drawing).

PixelProse (reference):

This image displays: a field of tall, yellow grass in the foreground. There is a large hill covered in green trees in the background. A rainbow can be seen in the sky above the hill. The rainbow is a full rainbow, with the red end on the left and the purple end on the right. There are no people or other objects visible in the image. The image is bright and sunny, and it appears to have been taken from a high angle. The answer to the question in the original caption is: yes, your sister did find the pot of gold, or at least the end of the rainbow.

PT + N (pretrained, factual prompt):

A rainbow is visible in the sky above a grassy field with trees in the background.

PT + D (pretrained, detailed prompt):

The image captures a serene rural landscape with a vibrant rainbow arching across the sky. The rainbow, with its distinct colors, adds a touch of magic to the scene. The foreground features a field, possibly a farm or a pasture, with a fence running along its edge. Beyond the field, there's a line of trees and bushes, providing a natural boundary to the area. The background is dominated by a hill or mountain, covered in lush greenery, suggesting a location that is rich in natural beauty. The overall atmosphere of the image is peaceful and idyllic, evoking a sense of tranquility and connection with nature.

PT + D_FS (pretrained, detailed few-shot):

The image depicts a serene landscape featuring a vibrant rainbow arching across the sky. The rainbow displays a spectrum of colors, including red, orange, yellow, green, blue, indigo, and violet, creating a striking contrast against the clear sky. Below the rainbow, there is a lush, green field with tall grasses and a variety of trees and bushes. The trees are dense and appear to be in full foliage, suggesting a healthy, thriving environment. The background consists of a hill or mountain covered with dense forest, adding depth to the scene. The overall atmosphere of the image is peaceful and picturesque, evoking a sense of tranquility and natural beauty.

FT + N (fine-tuned, factual prompt):

A rainbow is seen in the sky above a green forest. The rainbow is a spectrum of colors, with red on the left and violet on the right. The colors are orange, yellow, green, blue, indigo, and violet. The rainbow is a natural phenomenon that occurs when sunlight is refracted by water droplets in the air. The rainbow is a beautiful and awe-inspiring sight that can be seen by anyone who is willing to look. It is a reminder of the beauty and wonder of the natural world. *(The caption continues by repeating a generic explanation of rainbows.)*

FT + D (fine-tuned, detailed prompt):

This image shows a rainbow in the sky. The rainbow is a spectrum of colors, with red on the left and violet on the right. The colors are orange, yellow, green, blue, indigo, and violet. The rainbow is in the sky, and there is a hill in the background. The hill is covered in trees. The sky is blue. The image is a photograph. *(Later in the caption, the model repeats the explanation about the rainbow and viewpoint several times.)*

Figure 8: Qualitative gallery for ID 7 (rainbow over hill).

PixelProse (reference):

This image displays:
A woman wearing a white and gray zip-up jacket with a hood. The jacket has two front pockets. The hood is also gray on the inside. The woman is looking to the side, and away from the camera. The woman is standing with her arms at her side. She has a neutral, perhaps slightly happy expression on her face. The background is white without any other objects or scenery. The image is a photograph of a woman posing in front of a white background, and the style is fashion-oriented.

FT + D_FS (fine-tuned, detailed few-shot, failure case):

This image displays a pink background with black text. The text reads: "I just want to lose weight while staying in bed, watching TV, and eating Girl Scout cookies. Is that really too much to ask?" The text is in a white speech bubble on the left side of the image. On the right side of the image is a cartoon drawing of a person sitting on a couch with polka dots. The person is wearing pajamas and has their head in their hands. There is a box of Girl Scout cookies on the floor in front of the couch. The image is drawn in a simple cartoon style.

Explanation of failure: Instead of captioning the query image (a woman wearing a gray jacket), the FT + D_FS model reproduced the caption of *Exemplar 3* almost verbatim. This demonstrates the primary pathology observed when combining LoRA fine-tuning with multimodal few-shot prompting:

- The model overfits to the fine-tuned single-image prompt format, and
- When multiple exemplar images are introduced, it sometimes "locks on" to an exemplar and ignores the query image entirely.

This is one of the clearest examples of cross-image confusion caused by mixing LoRA-adapted layers with a multi-image context structure not seen during training.

Figure 9: Failure case for ID 3: the FT + D_FS model confuses an exemplar with the query image and outputs the exemplar's caption verbatim.

5 Discussion

The experiments suggest several key observations about the relative strengths and weaknesses of LoRA fine-tuning and multimodal few-shot prompting for dense captioning.

Caveat About Automated Metrics An important caveat in interpreting these results is that all automatic metrics are computed against the PixelProse captions themselves, not against the ground truth of the image. This means that any inaccuracies or subjective choices in PixelProse are effectively treated as ground truth. For example, in one of the few-shot exemplars the caption describes a necklace chain as silver even though it is visually gold; a model that correctly calls the chain gold will be penalized by BLEU, METEOR, and CIDEr because it diverges from the reference wording. As a result, the reported scores primarily measure stylistic and lexical similarity to PixelProse rather than objective visual correctness. Future work could address this by incorporating more independent evaluation signals, such as LLM-as-judge setups that directly inspect both the image and caption, or auxiliary task-specific metrics that score factual grounding of attributes and relations. The qualitative galleries are therefore essential: they reveal when a configuration achieves high metric scores by mimicking PixelProse quirks, and when it produces visually faithful captions that may nonetheless receive lower automatic scores due to label noise in the dataset.

Effectiveness of Few-Shot Prompting. For the pretrained model, multimodal few-shot prompting with three exemplars consistently improves automatic metrics and produces captions that are qualitatively closer to PixelProse outputs. The exemplars seem to serve two roles: they demonstrate the desired level of detail, and they implicitly bias the model toward the PixelProse style (including sentence structure and lexical choices). This indicates that few-shot prompting is a viable way to adapt general-purpose VLMs to dense captioning without any weight updates.

Limited Gains from Small-Scale LoRA Fine-Tuning. Due to resource constraints and the inability to run stable multi-GPU training, the LoRA fine-tuning in this project was limited to a relatively small subset of PixelProse and a modest number of optimization steps. Under these conditions, the fine-tuned model shows only small improvements over the pretrained model when using the same prompts. In particular, FT + D is competitive with PT + D but does not dramatically outperform it. This suggests that when training data and compute are limited, prompt engineering and few-shot prompting may yield comparable or larger gains than a small LoRA run.

Failure of the Combined FT + Few-Shot Setting. The most surprising result is that combining LoRA fine-tuning with detailed few-shot prompting (FT + D_FS) significantly *degrades* performance. The generated captions are often extremely long, repetitive, or misaligned with the query image. In several examples, the model appears to describe one of the exemplar images instead of the query, or it loops over phrases already present in the exemplars.

A plausible explanation is that the LoRA adapter was trained solely on single-image inputs paired with the detailed prompt, without any few-shot structure. During training, the model never sees multiple images in the same sequence or captions that refer to "exemplar" and "query" images. At evaluation time, the FT + D_FS configuration suddenly introduces multiple images and a more complex prompt. The adapted language model may therefore overfit to patterns in the exemplar captions and struggle to disentangle which visual features correspond to the current generation target.

This effect underscores an important lesson: fine-tuning and prompting must be aligned. If a model is fine-tuned under one prompt distribution and deployed under a different prompt distribution (for example, adding few-shot examples or changing instruction phrasing), can degrade performance on the dense captioning task.

Memory Cost of Multimodal Few-Shot Prompting. Another important observation is the memory footprint of multimodal few-shot prompting. When using three exemplar images plus the query, the system consumed roughly 50 GB of VRAM across my GPUs. In settings where GPU memory is limited or where batched inference over many images is required, this overhead may be prohibitive. In contrast, once a LoRA adapter is trained, it has essentially no additional inference-time memory cost beyond the base model.

This trade-off suggests that few-shot prompting is especially attractive in scenarios with ample memory and relatively low throughput requirements (for example, interactive captioning tools), while fine-tuning may be more appropriate when serving large numbers of requests or running on edge devices assuming the quality of fined tuned responses can be improved by reptition reduction techniques and better training conditions.

Implications for Future Systems. Taken together, these results argue for more holistic design of captioning systems. Few-shot prompting is a powerful tool, but aligning the training and inference prompts is crucial. When fine-tuning is used, it should ideally be performed with the same prompt structure (including few-shot exemplars) that will be used at deployment. Conversely, when only prompting is feasible, careful selection of exemplars and anti-repetition decoding strategies can yield strong gains without any training.

6 Conclusion

This project set out to compare LoRA fine-tuning and multimodal few-shot prompting as strategies for improving detailed image captioning with InternVL 3.5 2B on the PixelProse dataset. Within the constraints of a single-semester project, single-GPU training, and multi-GPU inference, I found that:

- Multimodal few-shot prompting with three exemplar image–caption pairs substantially improves similarity to PixelProse captions for the pretrained model, both quantitatively and qualitatively.
- The small-scale LoRA fine-tuning run used here provides only modest additional gains when evaluated under the same prompts, likely due to limited training data and issues with repetition.
- Combining LoRA fine-tuning with a prompt structure that is very different from the training prompt (detailed few-shot) can lead to worse performance and pathological behavior such as repetition and off-target descriptions.
- Few-shot prompting carries a significant VRAM cost when multiple images are processed in a single context, which may limit its applicability in memory-constrained settings.

From a learning perspective, I gained practical experience setting up a modern multimodal pipeline, implementing LoRA fine-tuning for a large vision–language model, and designing careful evaluations that compare prompting and training-based approaches. The project also reinforced the importance of aligning training and inference conditions and of combining automatic metrics with qualitative analysis.

Future work could pursue several directions: scaling LoRA training to multi-GPU settings and larger subsets of PixelProse, explicitly training the model on few-shot prompts that mirror the evaluation setup, experimenting with anti-repetition loss functions or decoding strategies, and exploring more diverse exemplar selection strategies to reduce overfitting to a small set of examples.

My Contribution

I worked alone on this project, so all aspects of the work were my responsibility. Specifically, I:

- Implemented the PixelProse subset construction script, downloaded and filtered the data, and organized the dataset into train, validation, and evaluation splits.
- Set up the InternVL 3.5 2B environment, including image tiling, preprocessing, and the captioning interface.
- Designed and implemented the LoRA fine-tuning pipeline on the language model component of InternVL, including dataset loading, batching, multimodal input construction, and checkpoint saving.
- Implemented the evaluation script for running multiple prompting configurations (neutral, detailed, and detailed few-shot) on both pretrained and fine-tuned models, and ensured exemplar IDs were excluded from the evaluation pool.

- Computed and analyzed automatic metrics (BLEU, METEOR, CIDEr, BERTScore), created summary CSV files, and constructed qualitative galleries of PixelProse and generated captions.
- Performed the analysis and discussion of results, including the identification of failure modes when combining LoRA fine-tuning with few-shot prompting and the assessment of VRAM usage during multimodal prompting.
- Wrote this report and organized the figures, tables, and references.

References

[1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. Flamingo: a visual language model for few-shot learning, 2022.

[2] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[3] Vasu Singla, Kaiyu Yue, Sukriti Paul, Reza Shirkavand, Mayuka Jayawardhana, Alireza Ganjdanesh, Heng Huang, Abhinav Bhatele, Gowthami Somepalli, and Tom Goldstein. From pixels to prose: A large dataset of dense image captions, 2024.

[4] Weiyun Wang, Zhangwei Gao, Lixin Gu, Hengjun Pu, Long Cui, Xingguang Wei, Zhaoyang Liu, Linglin Jing, Shenglong Ye, Jie Shao, Zhaokai Wang, Zhe Chen, Hongjie Zhang, Ganlin Yang, Haomin Wang, Qi Wei, Jinhui Yin, Wenhao Li, Erfei Cui, Guanzhou Chen, Zichen Ding, Changyao Tian, Zhenyu Wu, Jingjing Xie, Zehao Li, Bowen Yang, Yuchen Duan, Xuehui Wang, Zhi Hou, Haoran Hao, Tianyi Zhang, Songze Li, Xiangyu Zhao, Haodong Duan, Nianchen Deng, Bin Fu, Yinan He, Yi Wang, Conghui He, Botian Shi, Junjun He, Yingtong Xiong, Han Lv, Lijun Wu, Wenqi Shao, Kaipeng Zhang, Huipeng Deng, Biqing Qi, Jiaye Ge, Qipeng Guo, Wenwei Zhang, Songyang Zhang, Maosong Cao, Junyao Lin, Kexian Tang, Jianfei Gao, Haian Huang, Yuzhe Gu, Chengqi Lyu, Huanze Tang, Rui Wang, Haijun Lv, Wanli Ouyang, Limin Wang, Min Dou, Xizhou Zhu, Tong Lu, Dahua Lin, Jifeng Dai, Weijie Su, Bowen Zhou, Kai Chen, Yu Qiao, Wenhai Wang, and Gen Luo. Internvl3.5: Advancing open-source multimodal models in versatility, reasoning, and efficiency, 2025.

A eval_captions.py

```python
# src/eval_captions.py
#
# Evaluate InternVL 3.5 2B on a PixelProse subset under three prompting styles:
#   0 = PT/FT + Neutral prompt (N)
#   1 = PT/FT + Detailed prompt (D)
#   2 = PT/FT + Detailed + Few-shot (multimodal) (D_FS)
#
# This script:
#   - Selects `num_exemplars` exemplars from the subset metadata.
#   - Logs exemplars to logs/eval_subset{idx}_{PT/FT}_exemplars.jsonl.
#   - Excludes exemplar IDs from evaluation.
#   - Runs all three prompting styles on the SAME set of evaluation images.
#   - Writes three eval logs:
#       logs/eval_subset{idx}_{PT/FT}_N.jsonl
#       logs/eval_subset{idx}_{PT/FT}_D.jsonl
#       logs/eval_subset{idx}_{PT/FT}_D_FS.jsonl
#
# Example:
#   python -m src.eval_captions --subset-index 0 --max-eval 100 --num-exemplars 2

import argparse
import json
import os
from pathlib import Path
```

```python
from typing import Dict, Any, List, Set, Tuple

import torch
from transformers import AutoModel, AutoTokenizer

from src.test_internvl_caption import load_image
from peft import PeftModel


PROMPT_FILES = {
    0: "neutral.txt",          # N
    1: "detailed.txt",         # D
    2: "detailed_fewshot.txt", # D_FS instruction text
}

EXEMPLAR_MAX_NUM = 12
EXEMPLAR_INPUT_SIZE = 448


# ---------------------------------------------------------------------------
# Helpers
# ---------------------------------------------------------------------------

def load_metadata(meta_path: Path) -> List[Dict[str, Any]]:
    records = []
    with meta_path.open("r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
            records.append(json.loads(line))
    return records


def load_prompt(prompting_method: int, prompt_dir: Path = Path("prompts")) -> str:
    if prompting_method not in PROMPT_FILES:
        raise ValueError(f"Unknown prompting_method {prompting_method}")

    prompt_file = prompt_dir / PROMPT_FILES[prompting_method]
    if not prompt_file.exists():
        raise FileNotFoundError(f"Prompt file not found: {prompt_file}")

    text = prompt_file.read_text(encoding="utf-8").strip()
    if not text:
        raise ValueError(f"Prompt file is empty: {prompt_file}")
    return text


def load_model(use_finetuned: bool):
    """
    Load the base or LoRA-finetuned InternVL model.

    For FT:
    - load the base InternVL 3.5 2B Instruct
    - load the LoRA adapter on the *language_model* submodule
    - plug the LoRA-wrapped LM back into the full InternVLChatModel
    """
    base_id = "OpenGVLab/InternVL3_5-2B-Instruct"

    if use_finetuned:
        lora_path = os.getenv(
            "INTERNVL_FINETUNED_PATH",
            "checkpoints/internvl3_5_2b_lora_pixelprose/subset2_r32_a64",
        )
        print(
            f"[INFO] use_finetuned=True. "
            f"Base: {base_id}, LoRA adapter (language_model): {lora_path}"
        )

        # 1) Load the full InternVL base model
        base_model = AutoModel.from_pretrained(
            base_id,
            torch_dtype=torch.bfloat16,
            low_cpu_mem_usage=True,
            use_flash_attn=False,
            trust_remote_code=True,
            device_map="auto",
        )

        # 2) Wrap ONLY the language_model submodule with the saved LoRA adapter
        lm = base_model.language_model
        lm = PeftModel.from_pretrained(
            lm,
```

```python
            lora_path,
            is_trainable=False,  # eval-time, no gradients
        )
        lm.eval()
        base_model.language_model = lm

        model = base_model.eval()

    else:
        print(f"[INFO] use_finetuned=False. Loading pretrained model: {base_id}")
        model = AutoModel.from_pretrained(
            base_id,
            torch_dtype=torch.bfloat16,
            low_cpu_mem_usage=True,
            use_flash_attn=False,
            trust_remote_code=True,
            device_map="auto",
        ).eval()

    tokenizer = AutoTokenizer.from_pretrained(
        base_id,
        trust_remote_code=True,
        use_fast=False,
    )

    return model, tokenizer


def select_exemplars(
    records: List[Dict[str, Any]],
    num_exemplars: int,
) -> Tuple[List[Dict[str, Any]], Set[int]]:
    """
    Deterministically select the first `num_exemplars` records as exemplars.
    Returns (exemplar_records, exemplar_ids).
    """
    num_ex = min(num_exemplars, len(records))
    if num_ex <= 0:
        raise RuntimeError("No records available to select exemplars from.")

    exemplars = records[:num_ex]
    exemplar_ids = {int(r["id"]) for r in exemplars}
    return exemplars, exemplar_ids


# ------------------------------------------------------------------------
# Evaluation for one prompting method
# ------------------------------------------------------------------------

def evaluate_prompting_method(
    prompting_method: int,
    model,
    tokenizer,
    subset_dir: Path,
    eval_records: List[Dict[str, Any]],
    prompt_text: str,
    exemplar_pixels: List[torch.Tensor],
    exemplar_captions: List[str],
    model_tag: str,
    subset_index: int,
    max_eval: int,
) -> None:
    """
    Run evaluation for a single prompting method and write a log file.
    """
    if prompting_method == 0:
        prompt_tag = "N"
    elif prompting_method == 1:
        prompt_tag = "D"
    else:
        prompt_tag = "D_FS"

    out_dir = Path("logs")
    out_dir.mkdir(parents=True, exist_ok=True)

    out_path = out_dir / (
        f"eval_subset{subset_index}_{model_tag}_{prompt_tag}.jsonl"
    )
    print(f"[INFO] [{prompt_tag}] Writing outputs to: {out_path}")

    # NOTE: the config below has been modified since the final run. I am demonstrating what I would do to
    # reduce repetition here in practice. However, since I want a fair comparison between PT and FT, the final
    # run ran without repetition penalty or no_repeat_ngram_size
```

```python
    generation_config = dict(
        max_new_tokens=200,
        do_sample=False,              # keep deterministic for eval
        repetition_penalty=1.15,      # >1.0 penalizes repeats
        no_repeat_ngram_size=4,       # disallow exact 4-gram repeats
    )

    num_evaluated = 0

    with out_path.open("w", encoding="utf-8") as outf:
        for rec in eval_records:
            if num_evaluated >= max_eval:
                break

            rec_id = int(rec["id"])
            img_rel = rec["image_file"]
            img_path = subset_dir / img_rel
            gt_caption = rec["caption"]

            # Load query image tiles (CPU tensor)
            query_pv = load_image(str(img_path), max_num=12)

            if prompting_method in (0, 1):
                # Single-image case
                pixel_values = query_pv.to(torch.bfloat16).cuda()

                question = f"<image>\n{prompt_text}"

                pred_caption = model.chat(
                    tokenizer,
                    pixel_values,
                    question,
                    generation_config,
                )

            else:
                # Few-shot multimodal: exemplars + query image
                assert exemplar_pixels is not None and exemplar_captions is not None

                # 1) Move exemplars and query onto GPU
                ex_pv_gpu = [
                    pv.to(torch.bfloat16).cuda(non_blocking=True)
                    for pv in exemplar_pixels
                ]
                query_pv_gpu = query_pv.to(torch.bfloat16).cuda(non_blocking=True)

                # 2) Concatenate patches and build num_patches_list
                pixel_values_all = torch.cat(ex_pv_gpu + [query_pv_gpu], dim=0)
                num_patches_list = [
                    pv.size(0) for pv in ex_pv_gpu
                ] + [query_pv_gpu.size(0)]

                # 3) Build multimodal few-shot prompt
                lines = []
                for i, cap in enumerate(exemplar_captions):
                    lines.append(f"Exemplar-{i+1}: <image>")
                    lines.append(f"Caption-{i+1}: {cap}")
                    lines.append("")

                lines.append("Query: <image>")
                lines.append(prompt_text.strip())  # detailed_fewshot instruction

                question = "\n".join(lines)

                # Debug: print patch counts
                print("----- FEWSHOT INPUT DEBUG -----")
                print(f"Num exemplars: {len(ex_pv_gpu)}")
                for i, pv in enumerate(ex_pv_gpu):
                    print(f"  Exemplar-{i+1} patches: {pv.size(0)}")
                print(f"Query patches: {query_pv_gpu.size(0)}")
                print(f"num_patches_list: {num_patches_list}")
                print(f"Total patches: {pixel_values_all.size(0)}")
                print("------------------------------")

                pred_caption = model.chat(
                    tokenizer,
                    pixel_values_all,
                    question,
                    generation_config,
                    num_patches_list=num_patches_list,
                )

            out_rec = {
```

```python
                "id": rec_id,
                "image_file": img_rel,
                "gt_caption": gt_caption,
                "prompting_method": prompting_method,
                "prompt_text": question if prompting_method == 2 else prompt_text,
                "model_config": model_tag,
                "prediction": pred_caption,
            }
            outf.write(json.dumps(out_rec, ensure_ascii=False) + "\n")

            num_evaluated += 1
            if num_evaluated % 10 == 0:
                print(f"[INFO] [{prompt_tag}] Evaluated {num_evaluated}/{max_eval} examples...")

    print(f"[INFO] [{prompt_tag}] Evaluation complete. Total evaluated: {num_evaluated}")


# ---------------------------------------------------------------------------
# Main
# ---------------------------------------------------------------------------

def main():
    parser = argparse.ArgumentParser(
        description="Evaluate InternVL captions on a PixelProse subset (all prompting styles)."
    )
    parser.add_argument(
        "--subset-index",
        type=int,
        default=0,
        help="PixelProse subset index (uses data/pixelprose_subset{index}). Default: 0",
    )
    parser.add_argument(
        "--use-finetuned",
        action="store_true",
        help="Use LoRA-finetuned model (FT) instead of pretrained (PT). Default: False",
    )
    parser.add_argument(
        "--max-eval",
        type=int,
        default=100,
        help="Maximum number of examples to evaluate (excluding exemplars). Default: 100",
    )
    parser.add_argument(
        "--num-exemplars",
        type=int,
        default=2,
        help="Number of exemplars to use for few-shot prompting. Default: 2",
    )

    args = parser.parse_args()

    subset_dir = Path(f"data/pixelprose_subset{args.subset_index}")
    meta_path = subset_dir / "metadata.jsonl"

    assert subset_dir.exists(), f"Subset dir not found: {subset_dir}"
    assert meta_path.exists(), f"Metadata not found: {meta_path}"

    print(f"[INFO] Using subset dir: {subset_dir}")
    print(f"[INFO] Reading metadata from: {meta_path}")

    records = load_metadata(meta_path)
    print(f"[INFO] Loaded {len(records)} records total")

    # 1) Select exemplars from metadata
    exemplar_recs, exemplar_ids = select_exemplars(records, args.num_exemplars)
    print(f"[INFO] Selected {len(exemplar_recs)} exemplars from metadata:")
    print(f"[INFO] Exemplar IDs (excluded from eval): {sorted(exemplar_ids)}")

    # 2) Build evaluation records (excluding exemplars)
    eval_records = [r for r in records if int(r["id"]) not in exemplar_ids]
    print(f"[INFO] Eval records after excluding exemplars: {len(eval_records)}")

    # 3) Load all prompt texts
    prompt_texts = {
        m: load_prompt(m) for m in (0, 1, 2)
    }
    for m, txt in prompt_texts.items():
        print(f"[INFO] Prompt {m} text (first line): {txt.splitlines()[0] if txt else ''}")

    # 4) Load model
    model, tokenizer = load_model(args.use_finetuned)

    model_tag = "FT" if args.use_finetuned else "PT"
```

```python

    # 5) Log exemplars used for this run
    out_dir = Path("logs")
    out_dir.mkdir(parents=True, exist_ok=True)
    exemplar_log_path = out_dir / (
        f"eval_subset{args.subset_index}_{model_tag}_exemplars.jsonl"
    )
    print(f"[INFO] Writing exemplar info to: {exemplar_log_path}")
    with exemplar_log_path.open("w", encoding="utf-8") as exf:
        for ex in exemplar_recs:
            exf.write(json.dumps(ex, ensure_ascii=False) + "\n")

    # 6) Prepare exemplar tensors for few-shot prompting
    exemplar_pixels: List[torch.Tensor] = []
    exemplar_captions: List[str] = []
    for ex in exemplar_recs:
        img_rel = ex["image_file"]
        img_path = subset_dir / img_rel
        cap = ex["caption"]

        pv = load_image(
            str(img_path),
            input_size=EXEMPLAR_INPUT_SIZE,
            max_num=EXEMPLAR_MAX_NUM,
        )
        exemplar_pixels.append(pv)
        exemplar_captions.append(cap)

    print(f"[INFO] Prepared {len(exemplar_pixels)} exemplar tensors for few-shot.")

    # Clip eval_records to max_eval for consistency across all methods
    eval_records = eval_records[: args.max_eval]
    print(f"[INFO] Using {len(eval_records)} eval records for all prompting styles.")

    # 7) Run all prompting methods in sequence on the SAME eval set
    for prompting_method in (0,1,2):
        print(f"\n[INFO] Starting evaluation for prompting_method={prompting_method}...")
        evaluate_prompting_method(
            prompting_method=prompting_method,
            model=model,
            tokenizer=tokenizer,
            subset_dir=subset_dir,
            eval_records=eval_records,
            prompt_text=prompt_texts[prompting_method],
            exemplar_pixels=exemplar_pixels,
            exemplar_captions=exemplar_captions,
            model_tag=model_tag,
            subset_index=args.subset_index,
            max_eval=len(eval_records),
        )

    print("\n[INFO] All prompting styles evaluated.")


if __name__ == "__main__":
    main()

```

B build_pixelprose_dataset.py

```python
# scripts/build_pixelprose_subset.py

import json
from io import BytesIO
from pathlib import Path
from typing import Optional

import requests
from datasets import load_dataset
from PIL import Image
from tqdm import tqdm


def download_image(url: str, out_path: Path, timeout: float = 10.0) -> bool:
    """Download image from URL and save as JPEG. Returns True on success, False otherwise."""
    try:
        resp = requests.get(url, timeout=timeout)
        resp.raise_for_status()
        img = Image.open(BytesIO(resp.content)).convert("RGB")
        out_path.parent.mkdir(parents=True, exist_ok=True)
```

```python
        img.save(out_path, format="JPEG", quality=95)
        return True
    except Exception as e:
        print(f"[WARN] Failed to download {url} -> {out_path}: {e}")
        return False


def main(dataset_index: int, num_samples: int):
    """
    Build a PixelProse subset with num_samples successfully downloaded images,
    if possible.

    Args:
        dataset_index: integer index used to choose output directory and shuffle seed.
        num_samples: number of successful image downloads to collect.
    """
    # Output dir: data/pixelprose_subset{index}
    out_dir = f"data/pixelprose_subset{dataset_index}"
    seed = 42 + dataset_index

    out_dir_path = Path(out_dir)
    img_dir = out_dir_path / "images"
    out_dir_path.mkdir(parents=True, exist_ok=True)
    img_dir.mkdir(parents=True, exist_ok=True)

    print(f"=== Building PixelProse subset {dataset_index} ===")
    print(f"Output dir      : {out_dir_path}")
    print(f"Requested images: {num_samples}")
    print(f"Shuffle seed    : {seed}")

    print("Loading PixelProse from Hugging Face...")
    ds = load_dataset("tomg-group-umd/pixelprose", split="train", streaming=False)

    print("Columns:", ds.column_names)
    total_size = len(ds)
    print(f"Dataset size    : {total_size}")

    print("Shuffling...")
    ds = ds.shuffle(seed=seed)

    meta_path = out_dir_path / "metadata.jsonl"

    kept = 0
    tried = 0

    # Progress bar tracks kept images
    pbar = tqdm(total=num_samples, desc="Collected images", unit="img")

    with meta_path.open("w", encoding="utf-8") as f:
        # Iterate over the entire shuffled dataset until we have num_samples
        for ex in ds:
            if kept >= num_samples:
                break

            tried += 1

            url: Optional[str] = ex.get("url")
            caption: Optional[str] = ex.get("vlm_caption") or ex.get("original_caption")

            if not url or not caption:
                # Missing necessary fields; skip
                continue

            img_fname = f"{kept:06d}.jpg"  # IDs local to this subset: 0..num_samples-1
            img_path = img_dir / img_fname

            ok = download_image(url, img_path)
            if not ok:
                continue

            record = {
                "id": kept,
                "uid": ex.get("uid"),
                "url": url,
                "image_file": str(img_path.relative_to(out_dir_path)),
                "caption": caption,
                "vlm_model": ex.get("vlm_model"),
                "aesthetic_score": ex.get("aesthetic_score"),
                "watermark_class_score": ex.get("watermark_class_score"),
            }
            f.write(json.dumps(record, ensure_ascii=False) + "\n")
            kept += 1
            pbar.update(1)
```

```python

    pbar.close()

    print(f"Done scanning dataset.")
    print(f"Tried examples   : {tried}")
    print(f"Kept images      : {kept}")
    print(f"Requested images : {num_samples}")
    print(f"Data dir         : {out_dir_path}")
    print(f"Metadata         : {meta_path}")

    if kept < num_samples:
        print(
            f"[WARN] Could not collect the requested {num_samples} images. "
            f"Only {kept} valid downloads were available in the entire shuffled dataset."
        )


if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(
        description=(
            "Build a PixelProse subset with a given index and a target number of "
            "successfully downloaded images."
        )
    )
    parser.add_argument(
        "dataset_index",
        type=int,
        help="Integer index to distinguish this subset (used in directory name and shuffle seed).",
    )
    parser.add_argument(
        "num_samples",
        type=int,
        help="Number of successfully downloaded images to collect.",
    )

    args = parser.parse_args()
    main(args.dataset_index, args.num_samples)

```

C lora_config.py

```python
# src/lora_config.py
from peft import LoraConfig, TaskType

TARGET_MODULES = [
    "q_proj",
    "k_proj",
    "v_proj",
    "o_proj",
    "gate_proj",
    "up_proj",
    "down_proj",
]

def make_default_lora_config(
    r: int = 32,
    alpha: int = 64,
    dropout: float = 0.05,
) -> LoraConfig:
    return LoraConfig(
        r=r,
        lora_alpha=alpha,
        lora_dropout=dropout,
        bias="none",
        task_type=TaskType.CAUSAL_LM,
        target_modules=TARGET_MODULES,
    )

```

D train_lora.py

```python
# src/train_lora.py
#
#   - Data loading and fixed-size split.
#   - Dataset that loads images + text.
#   - DataLoader + collate_fn that batches variable-length text and variable
```

```python
#     numbers of image patches.


import argparse
import json
from pathlib import Path
from typing import Any, Dict, List, Tuple
import time

import torch
from torch.utils.data import Dataset, DataLoader
from transformers import AutoModel, AutoTokenizer

from src.eval_captions import load_prompt
from src.test_internvl_caption import load_image
import contextlib

from peft import get_peft_model, PeftModel

from src.lora_config import make_default_lora_config
from src.test_internvl_caption import load_image


MODEL_ID = "OpenGVLab/InternVL3_5-2B-Instruct"
MIN_TRAIN_ID = 1000

IMG_START_TOKEN = "<img>"
IMG_END_TOKEN = "</img>"
IMG_CONTEXT_TOKEN = "<IMG_CONTEXT>"


def load_metadata(meta_path: Path) -> List[Dict[str, Any]]:
    records: List[Dict[str, Any]] = []
    with meta_path.open("r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
            records.append(json.loads(line))
    return records


def split_train_val_by_counts(
    records: List[Dict[str, Any]],
    min_train_id: int,
    train_size: int,
    val_size: int,
) -> Tuple[List[Dict[str, Any]], List[Dict[str, Any]]]:
    """
    Filter to ids >= min_train_id, then take the first train_size for train
    and the next val_size for val. Deterministic: no shuffling for now.
    """
    usable = [r for r in records if int(r["id"]) >= min_train_id]
    n_usable = len(usable)

    if n_usable == 0:
        raise RuntimeError(
            f"No usable records with id >= {min_train_id}. "
            "Check your metadata or subset index."
        )

    if train_size + val_size > n_usable:
        print(
            f"[WARN] Requested train_size ({train_size}) + val_size ({val_size}) "
            f"= {train_size + val_size} but only {n_usable} usable records exist. "
            "Truncating to fit."
        )
        train_size = min(train_size, n_usable)
        val_size = min(val_size, max(0, n_usable - train_size))

    if train_size == 0 or val_size == 0:
        raise RuntimeError(
            f"After adjustment, train_size={train_size}, val_size={val_size}. "
            "Both must be > 0."
        )

    train_records = usable[:train_size]
    val_records = usable[train_size:train_size + val_size]

    return train_records, val_records


class PixelProseLoraDataset(Dataset):
```

```python
    """
    Minimal dataset for LoRA:

    - One image per example (same load_image as eval).
    - Labels are the same as input_ids.
    """

    def __init__(
        self,
        records: List[Dict[str, Any]],
        subset_dir: Path,
        tokenizer,
        max_length: int = 512,
    ) -> None:
        self.records = records
        self.subset_dir = subset_dir
        self.tokenizer = tokenizer
        self.max_length = max_length

        # Use the same detailed prompt I already use for prompting_method=1
        self.prompt_text = load_prompt(1)  # 1 = detailed.txt
        print(f"[INFO] Loaded detailed prompt for training. First line:")
        print(f"       {self.prompt_text.splitlines()[0] if self.prompt_text else ''}")

    def __len__(self) -> int:
        return len(self.records)

    def __getitem__(self, idx: int) -> Dict[str, Any]:
        rec = self.records[idx]
        img_rel = rec["image_file"]
        caption = rec["caption"]

        img_path = self.subset_dir / img_rel

        pixel_values = load_image(str(img_path), max_num=12)

        question = f"<image>\n{self.prompt_text}"
        full_text = question + "\n" + caption

        enc = self.tokenizer(
            full_text,
            max_length=self.max_length,
            truncation=True,
            return_tensors="pt",
        )

        input_ids = enc["input_ids"].squeeze(0)
        attention_mask = enc["attention_mask"].squeeze(0)
        labels = input_ids.clone()

        return {
            "id": int(rec["id"]),
            "image_file": img_rel,
            "pixel_values": pixel_values,
            "input_ids": input_ids,
            "attention_mask": attention_mask,
            "labels": labels,
            "caption": caption,
        }


def make_collate_fn(pad_token_id: int):
    """
    Collate function that:
    - Pads text to max length in batch.
    - Concatenates all pixel patches and records num_patches_list.
    """

    def collate(batch: List[Dict[str, Any]]) -> Dict[str, Any]:
        # Text part: pad to max length
        input_ids_list = [b["input_ids"] for b in batch]
        attention_list = [b["attention_mask"] for b in batch]
        labels_list = [b["labels"] for b in batch]
        ids_list = [b["id"] for b in batch]
        image_files = [b["image_file"] for b in batch]
        captions = [b["caption"] for b in batch]

        max_len = max(x.size(0) for x in input_ids_list)

        batch_input_ids = []
        batch_attention = []
        batch_labels = []

```

```python
        for inp, attn, lab in zip(input_ids_list, attention_list, labels_list):
            pad_len = max_len - inp.size(0)

            if pad_len > 0:
                pad_ids = torch.full((pad_len,), pad_token_id, dtype=inp.dtype)
                pad_attn = torch.zeros(pad_len, dtype=attn.dtype)
                pad_lab = torch.full((pad_len,), -100, dtype=lab.dtype)  # ignore pad in loss

                inp = torch.cat([inp, pad_ids], dim=0)
                attn = torch.cat([attn, pad_attn], dim=0)
                lab = torch.cat([lab, pad_lab], dim=0)

            batch_input_ids.append(inp)
            batch_attention.append(attn)
            batch_labels.append(lab)

        batch_input_ids = torch.stack(batch_input_ids, dim=0)      # [B, T]
        batch_attention = torch.stack(batch_attention, dim=0)      # [B, T]
        batch_labels = torch.stack(batch_labels, dim=0)            # [B, T]

        # Image part: variable patches per sample
        pixel_values_list = [b["pixel_values"] for b in batch]
        num_patches_list = [pv.size(0) for pv in pixel_values_list]

        # Concatenate along patch dimension
        pixel_values_all = torch.cat(pixel_values_list, dim=0)

        return {
            "ids": ids_list,
            "image_files": image_files,
            "captions": captions,
            "pixel_values": pixel_values_all,
            "num_patches_list": num_patches_list,
            "input_ids": batch_input_ids,
            "attention_mask": batch_attention,
            "labels": batch_labels,
        }

    return collate

def build_mm_inputs_for_batch(
    tokenizer,
    captions,
    num_patches_list,
    prompt_text: str,
    num_image_token: int,
    max_length: int,
    device: torch.device,
):
    """
    Build input_ids / attention_mask / labels with the correct number of
    <IMG_CONTEXT> tokens per sample, so InternVL can inject visual features.
    """
    queries = []
    for cap, num_patches in zip(captions, num_patches_list):
        # How many visual tokens this sample needs
        n_vis_tokens = num_image_token * num_patches

        image_tokens = (
            IMG_START_TOKEN
            + IMG_CONTEXT_TOKEN * n_vis_tokens
            + IMG_END_TOKEN
        )

        # Simple instruction format: [image tokens] + prompt + caption
        text = (
            image_tokens
            + "\n"
            + prompt_text.strip()
            + "\n"
            + cap
        )
        queries.append(text)

    # Tokenize as a batch
    enc = tokenizer(
        queries,
        padding=True,
        truncation=True,
        max_length=max_length,
        return_tensors="pt",
    )
    input_ids = enc["input_ids"].to(device)
```

```python
    attention_mask = enc["attention_mask"].to(device)
    labels = input_ids.clone()

    return input_ids, attention_mask, labels


def main() -> None:
    parser = argparse.ArgumentParser(
        description=(
            "LoRA training setup: data loading, fixed-size split, dataset, "
            "and a single-batch DataLoader sanity check."
        )
    )
    parser.add_argument(
        "--subset-index",
        type=int,
        default=2,
        help="PixelProse subset index (uses data/pixelprose_subset{index}). Default: 2",
    )
    parser.add_argument(
        "--train-size",
        type=int,
        default=2000,
        help="Number of train samples to take from id >= 1000 pool. Default: 2000",
    )
    parser.add_argument(
        "--val-size",
        type=int,
        default=500,
        help="Number of val samples to take from id >= 1000 pool. Default: 500",
    )
    parser.add_argument(
        "--max-length",
        type=int,
        default=512,
        help="Max sequence length for tokenizer. Default: 512",
    )
    parser.add_argument(
        "--batch-size",
        type=int,
        default=4,
        help="Batch size for DataLoader sanity check. Default: 4",
    )
    parser.add_argument(
        "--num-epochs",
        type=int,
        default=1,
        help="Number of training epochs over the train set. Default: 1",
    )
    parser.add_argument(
        "--max-steps",
        type=int,
        default=50,
        help="Optional cap on training steps (batches) for quick runs. Default: 50",
    )
    parser.add_argument(
        "--cuda-device",
        type=int,
        default=0,
        help="Which CUDA device to run on (single-GPU). Default: 0",
    )
    parser.add_argument(
        "--resume-from",
        type=str,
        default=None,
        help="Path to an existing LoRA adapter to continue training from.",
    )
    parser.add_argument(
        "--output-dir",
        type=str,
        default=None,
        help="Directory to save the LoRA adapter. "
            "If not set, a default path based on subset index is used.",
    )

    args = parser.parse_args()

    subset_dir = Path(f"data/pixelprose_subset{args.subset_index}")
    meta_path = subset_dir / "metadata.jsonl"

    if not subset_dir.exists():
        raise FileNotFoundError(f"Subset dir not found: {subset_dir}")
    if not meta_path.exists():
```

```
        raise FileNotFoundError(f"Metadata not found: {meta_path}")

    print(f"[INFO] Using subset dir: {subset_dir}")
    print(f"[INFO] Reading metadata from: {meta_path}")

    records = load_metadata(meta_path)
    print(f"[INFO] Loaded {len(records)} records total")

    # Explicitly count how many are reserved for final eval
    reserved = [r for r in records if int(r["id"]) < MIN_TRAIN_ID]
    print(f"[INFO] Reserved for final PT/FT eval (id < {MIN_TRAIN_ID}): {len(reserved)}")

    train_records, val_records = split_train_val_by_counts(
        records,
        min_train_id=MIN_TRAIN_ID,
        train_size=args.train_size,
        val_size=args.val_size,
    )

    print(f"[INFO] Usable for LoRA (id >= {MIN_TRAIN_ID}): {len(train_records) + len(val_records)}")
    print(f"[INFO] Train records (requested {args.train_size}): {len(train_records)}")
    print(f"[INFO] Val records   (requested {args.val_size}):  {len(val_records)}")

    #  show id ranges
    if train_records:
        print(
            f"[DEBUG] Train id range: "
            f"{min(int(r['id']) for r in train_records)}"
            f" .. {max(int(r['id']) for r in train_records)}"
        )
    if val_records:
        print(
            f"[DEBUG] Val id range:   "
            f"{min(int(r['id']) for r in val_records)}"
            f" .. {max(int(r['id']) for r in val_records)}"
        )

    # Load tokenizer
    print(f"[INFO] Loading tokenizer: {MODEL_ID}")
    tokenizer = AutoTokenizer.from_pretrained(
        MODEL_ID,
        trust_remote_code=True,
        use_fast=False,
    )

    pad_token_id = tokenizer.pad_token_id or tokenizer.eos_token_id
    print(f"[INFO] Using pad_token_id={pad_token_id}")

    # Build training dataset
    train_ds = PixelProseLoraDataset(
        train_records,
        subset_dir=subset_dir,
        tokenizer=tokenizer,
        max_length=args.max_length,
    )
    # Save prompt + max_length for multimodal input building later
    train_prompt_text = train_ds.prompt_text
    train_max_length = args.max_length

    print(f"[INFO] Built training dataset with {len(train_ds)} samples.")

    # DataLoader + single-batch sanity check
    collate_fn = make_collate_fn(pad_token_id)
    train_loader = DataLoader(
        train_ds,
        batch_size=args.batch_size,
        shuffle=True,
        num_workers=4,
        collate_fn=collate_fn,
        pin_memory=False,
    )

    print("[INFO] Fetching one batch from DataLoader for sanity check...")
    debug_batch = next(iter(train_loader))

    print("\n[INFO] Debug batch summary:")
    print(f"  ids:                 {debug_batch['ids']}")
    print(f"  num_patches_list:    {debug_batch['num_patches_list']}")
    print(f"  pixel_values shape:  {tuple(debug_batch['pixel_values'].shape)}  "
          "(sum_patches, 3, H, W)")
    print(f"  input_ids shape:     {tuple(debug_batch['input_ids'].shape)}      (B, T)")
    print(f"  attention_mask shape:{tuple(debug_batch['attention_mask'].shape)} (B, T)")
    print(f"  labels shape:        {tuple(debug_batch['labels'].shape)}         (B, T)")
```

```python
    print(f"  attention_mask sums: "
          f"{[int(x) for x in debug_batch['attention_mask'].sum(dim=1)]}")

    # -------------------------------------------------------
    # Load base InternVL model and wrap language_model with LoRA
    # -------------------------------------------------------
    device = torch.device(f"cuda:{args.cuda_device}" if torch.cuda.is_available() else "cpu")
    model_id = MODEL_ID

    print(f"[INFO] Loading base InternVL model for LoRA on {device}: {model_id}")
    model = AutoModel.from_pretrained(
        model_id,
        torch_dtype=torch.bfloat16,
        low_cpu_mem_usage=True,
        use_flash_attn=False,
        trust_remote_code=True,
        device_map=None,          # SINGLE GPU
    )

    # Build or resume LoRA adapter on the language_model (Qwen LM)
    if args.resume_from:
        print(f"[INFO] Resuming LoRA from adapter at: {args.resume_from}")
        lm = PeftModel.from_pretrained(
            model.language_model,
            args.resume_from,
            is_trainable=True,  # keep LoRA params trainable
        )
    else:
        print("[INFO] Creating new LoRA adapter for language_model...")
        lora_config = make_default_lora_config()
        lm = get_peft_model(model.language_model, lora_config)

    lm.print_trainable_parameters()
    model.language_model = lm  # plug LoRA LM back into InternVL

    model.to(device)
    model.train()

    # Tell InternVL which token is the image-context marker
    img_context_token_id = tokenizer.convert_tokens_to_ids(IMG_CONTEXT_TOKEN)
    model.img_context_token_id = img_context_token_id
    print(f"[INFO] IMG_CONTEXT_TOKEN id: {img_context_token_id}, "
          f"num_image_token: {model.num_image_token}")

    # Optimizer over LoRA params only
    optimizer = torch.optim.AdamW(
        [p for p in model.language_model.parameters() if p.requires_grad],
        lr=1e-4,
        weight_decay=0.01,
    )

    if device.type == "cuda":
        autocast_ctx = torch.cuda.amp.autocast(dtype=torch.bfloat16)
    else:
        autocast_ctx = contextlib.nullcontext()

    global_step = 0
    print(f"[INFO] Starting training for {args.num_epochs} epoch(s), "
          f"max_steps={args.max_steps}...")
    train_start_time = time.time()

    for epoch in range(args.num_epochs):
        print(f"\n[INFO] Epoch {epoch + 1}/{args.num_epochs}")
        for step, batch in enumerate(train_loader):
            if global_step >= args.max_steps:
                print(f"[INFO] Reached max_steps={args.max_steps}, stopping training.")
                break

            model.train()

            pixel_values = batch["pixel_values"].to(device, dtype=torch.bfloat16)
            num_patches_list = batch["num_patches_list"]
            captions = batch["captions"]

            input_ids, attention_mask, labels = build_mm_inputs_for_batch(
                tokenizer=tokenizer,
                captions=captions,
                num_patches_list=num_patches_list,
                prompt_text=train_prompt_text,
                num_image_token=model.num_image_token,
                max_length=train_max_length,
                device=device,
            )
```

```python

            image_flags = torch.ones(
                pixel_values.size(0), 1,
                dtype=torch.long,
                device=device,
            )

            optimizer.zero_grad(set_to_none=True)

            with autocast_ctx:
                outputs = model(
                    pixel_values=pixel_values,
                    input_ids=input_ids,
                    attention_mask=attention_mask,
                    labels=labels,
                    image_flags=image_flags,
                )
                loss = outputs.loss

            loss_value = float(loss.item())
            loss.backward()

            # Optional grad norm monitor
            if (global_step % 10) == 0:
                total_norm = 0.0
                count = 0
                for p in model.language_model.parameters():
                    if p.requires_grad and p.grad is not None:
                        param_norm = p.grad.data.norm(2).item()
                        total_norm += param_norm ** 2
                        count += 1
                if count > 0:
                    total_norm = total_norm ** 0.5
                print(f"[STEP {global_step}] loss={loss_value:.4f}, "
                      f"grad_norm={total_norm:.4f}")

            optimizer.step()
            global_step += 1

        if global_step >= args.max_steps:
            break

    print(f"\n[INFO] Training finished at global_step={global_step}.")

    # ------------------------------
    # Timing summary
    # ------------------------------
    train_end_time = time.time()
    elapsed = train_end_time - train_start_time
    mins = elapsed / 60
    hrs = mins / 60
    print("[INFO] ------------------------------------------")
    print(f"[INFO] Training elapsed time: {elapsed:.2f} seconds")
    print(f"[INFO] = {mins:.2f} minutes")
    print(f"[INFO] = {hrs:.2f} hours")
    print("[INFO] ------------------------------------------")

    # -------------------------------------------------------
    # Save LoRA adapter (language_model only)
    # -------------------------------------------------------

    # Choose output directory
    if args.output_dir is not None:
        out_dir = Path(args.output_dir)
    else:
        # Default path if none is provided
        out_dir = Path(
            f"checkpoints/internvl3_5_2b_lora_pixelprose/subset{args.subset_index}_r32_a64"
        )

    out_dir.mkdir(parents=True, exist_ok=True)

    print(f"[INFO] Saving LoRA adapter to: {out_dir}")
    model.language_model.save_pretrained(out_dir)
    print("[INFO] Done. You can now load this adapter in eval_captions.py via PeftModel.from_pretrained(base_model,
    ↪  <adapter_path>).")

if __name__ == "__main__":
    main()

```

E compute_metrics.py

```python
#!/usr/bin/env python
# scripts/compute_metrics.py
#
# Compute automatic metrics for one or more eval logs.
# Currently: BLEU only (via sacrebleu).
# Design: to add a new metric (e.g., METEOR), you:
#   1) Implement compute_meteor(preds, refs)
#   2) Register it in METRIC_FNS = {"bleu": compute_bleu, "meteor": compute_meteor, ...}
#
# Usage:
#   python -m scripts.compute_metrics \
#       --logs logs/eval_subset0_PT_N.jsonl \
#              logs/eval_subset0_PT_D.jsonl \
#              logs/eval_subset0_PT_D_FS.jsonl \
#       --out-csv metrics/metrics_subset0_PT.csv

import argparse
import csv
import json
from pathlib import Path
from typing import List, Dict, Any, Callable

import sacrebleu
import nltk
nltk.data.path.insert(0, ".venv/nltk_data")
from nltk.translate.meteor_score import meteor_score
from nltk.tokenize import word_tokenize
import evaluate



# ---------------------------------------------------------------------
# Metric functions (all share the same signature)
# ---------------------------------------------------------------------

def compute_bleu(preds: List[str], refs: List[str]) -> float:
    """
    Corpus BLEU using sacrebleu.
    preds: list of model predictions (strings)
    refs:  list of reference captions (strings)
    """
    if not preds:
        return 0.0
    bleu = sacrebleu.corpus_bleu(preds, [refs])
    return float(bleu.score)

def compute_meteor(preds: List[str], refs: List[str]) -> float:
    """
    Corpus METEOR as the average of sentence-level METEOR scores.
    Returns score on 0-100 scale for consistency with BLEU.
    """
    if not preds:
        return 0.0

    scores = []
    for hyp, ref in zip(preds, refs):
        hyp = hyp or ""
        ref = ref or ""

        # Tokenize both hypothesis and reference
        hyp_tokens = word_tokenize(hyp)
        ref_tokens = word_tokenize(ref)

        # meteor_score expects token lists:
        # references: List[List[str]], hypothesis: List[str]
        scores.append(meteor_score([ref_tokens], hyp_tokens))

    avg_score = sum(scores) / len(scores) if scores else 0.0
    return float(avg_score * 100.0)

# Global handle for CIDEr metric (lazy-loaded)
_CIDER_METRIC = None

def get_cider_metric():
    global _CIDER_METRIC
    if _CIDER_METRIC is None:
        # Uses the Kamichanw/CIDEr metric on the Hub
        _CIDER_METRIC = evaluate.load("Kamichanw/CIDEr")
    return _CIDER_METRIC

def compute_cider(preds: List[str], refs: List[str]) -> float:
```

```
    """
    Corpus CIDEr using the Hugging Face evaluate implementation.

    preds: list of hypothesis captions (strings)
    refs:  list of reference captions (strings)

    We have exactly one reference per prediction, so we wrap each ref
    in a singleton list, as the metric expects List[List[str]].
    """
    if not preds:
        return 0.0

    metric = get_cider_metric()

    # Metric expects:
    #   predictions: List[str]
    #   references: List[List[str]]  (list of reference captions per prediction)
    references_wrapped = [[r or ""] for r in refs]
    predictions = [p or "" for p in preds]

    result = metric.compute(predictions=predictions, references=references_wrapped)
    # The metric returns something like {"CIDEr": value}
    score = float(result.get("CIDEr", 0.0))

    return score
# Global handle for BERTScore metric (lazy-loaded)
_BERTSCORE_METRIC = None

def get_bertscore_metric():
    global _BERTSCORE_METRIC
    if _BERTSCORE_METRIC is None:
        # This uses the 'bert-score' metric from Hugging Face evaluate
        _BERTSCORE_METRIC = evaluate.load("bertscore")
    return _BERTSCORE_METRIC

def compute_bertscore(preds: List[str], refs: List[str]) -> float:
    """
    Corpus BERTScore (F1), averaged over all examples.
    Returns score on 0-100 scale for consistency with BLEU/METEOR/CIDEr.
    """
    if not preds:
        return 0.0

    metric = get_bertscore_metric()

    # BERTScore expects:
    #   predictions: List[str]
    #   references:  List[str]
    result = metric.compute(
        predictions=[p or "" for p in preds],
        references=[r or "" for r in refs],
        lang="en",
        model_type="bert-base-uncased",  # lighter, good enough for this project
        rescale_with_baseline=True,
    )

    # result["f1"] is a list of per-example scores in [0, 1]
    f1_scores = result["f1"]
    avg_f1 = sum(f1_scores) / len(f1_scores)

    return float(avg_f1 * 100.0)

# Registry of metric name -> function
METRIC_FNS: Dict[str, Callable[[List[str], List[str]], float]] = {
    "bleu": compute_bleu,
    "meteor": compute_meteor,
    "cider": compute_cider,
    "bertscore": compute_bertscore,
}

# --------------------------------------------------------------------
# Helpers to load logs and compute basic stats
# --------------------------------------------------------------------

def load_records(path: Path) -> List[Dict[str, Any]]:
    assert path.exists(), f"Log file not found: {path}"
    recs: List[Dict[str, Any]] = []
    with path.open("r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
            recs.append(json.loads(line))
```

```python
    return recs


def avg_len(texts: List[str]) -> float:
    if not texts:
        return 0.0
    return sum(len(t.split()) for t in texts) / len(texts)


def compute_all_metrics(preds: List[str], refs: List[str]) -> Dict[str, float]:
    """
    Run all registered metrics in METRIC_FNS and return a dict
    {metric_name: value}.
    """
    results: Dict[str, float] = {}
    for name, fn in METRIC_FNS.items():
        results[name] = fn(preds, refs)
    return results


def summarize(path: Path) -> Dict[str, Any]:
    recs = load_records(path)

    if not recs:
        base = {
            "log": str(path),
            "prompting_method": "",
            "model_config": "",
            "num_records": 0,
            "avg_gt_len": 0.0,
            "avg_pred_len": 0.0,
        }
        # Fill metrics with zeros so CSV headers remain consistent
        for m in METRIC_FNS.keys():
            base[m] = 0.0
        return base

    gts = [r["gt_caption"] for r in recs]
    preds = [r["prediction"] for r in recs]

    avg_gt = avg_len(gts)
    avg_pred = avg_len(preds)
    metric_vals = compute_all_metrics(preds, gts)

    tag_pm = recs[0].get("prompting_method", "")
    tag_cfg = recs[0].get("model_config", "")

    row: Dict[str, Any] = {
        "log": str(path),
        "prompting_method": tag_pm,
        "model_config": tag_cfg,
        "num_records": len(recs),
        "avg_gt_len": avg_gt,
        "avg_pred_len": avg_pred,
    }
    row.update(metric_vals)
    return row


# ---------------------------------------------------------------------
# CSV + pretty table
# ---------------------------------------------------------------------

def write_csv(rows: List[Dict[str, Any]], out_csv: Path) -> None:
    out_csv.parent.mkdir(parents=True, exist_ok=True)

    base_fields = [
        "log",
        "prompting_method",
        "model_config",
        "num_records",
        "avg_gt_len",
        "avg_pred_len",
    ]
    metric_fields = list(METRIC_FNS.keys())
    fieldnames = base_fields + metric_fields

    with out_csv.open("w", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        writer.writeheader()
        for r in rows:
            writer.writerow({k: r.get(k, "") for k in fieldnames})

```

```python
    print(f"\n[INFO] Wrote metrics CSV to: {out_csv}")


def print_table(rows: List[Dict[str, Any]]) -> None:
    """
    Pretty console output showing all registered metrics dynamically.
    """
    metric_names = list(METRIC_FNS.keys())  # e.g. ["bleu", "meteor"]

    print("\n=== Metric Summary ===\n")

    # Build dynamic header
    header_parts = [
        f"{'log':45}",
        f"{'PM':>2}",
        f"{'CFG':>3}",
        f"{'#':>5}",
        f"{'avg_gt':>8}",
        f"{'avg_pred':>9}",
    ]

    # Add each metric with fixed width
    for m in metric_names:
        header_parts.append(f"{m:>10}")

    header_line = "  ".join(header_parts)
    print(header_line)
    print("-" * len(header_line))

    # Table rows
    for r in rows:
        base_parts = [
            f"{Path(r['log']).name:45}",
            f"{str(r['prompting_method']):>2}",
            f"{str(r['model_config']):>3}",
            f"{r['num_records']:5d}",
            f"{r['avg_gt_len']:8.2f}",
            f"{r['avg_pred_len']:9.2f}",
        ]

        metric_parts = [f"{r[m]:10.2f}" for m in metric_names]
        line = "  ".join(base_parts + metric_parts)
        print(line)


# ----------------------------------------------------------------------
# Main
# ----------------------------------------------------------------------

def main():
    parser = argparse.ArgumentParser(
        description="Compute metrics and caption length statistics for eval logs."
    )
    parser.add_argument(
        "--logs",
        nargs="+",
        required=True,
        help="Paths to eval logs (JSONL).",
    )
    parser.add_argument(
        "--out-csv",
        type=str,
        default=None,
        help="Optional path to write metrics CSV (e.g., metrics/metrics_subset0_PT.csv).",
    )
    args = parser.parse_args()

    rows: List[Dict[str, Any]] = [summarize(Path(lp)) for lp in args.logs]

    print_table(rows)

    if args.out_csv is not None:
        write_csv(rows, Path(args.out_csv))


if __name__ == "__main__":
    main()
```

F test_internvl_caption.py

```python
# src/test_internvl_caption.py
# load_image is used in other files
import math
from pathlib import Path

import torch
import torchvision.transforms as T
from PIL import Image
from torchvision.transforms.functional import InterpolationMode
from transformers import AutoModel, AutoTokenizer

IMAGENET_MEAN = (0.485, 0.456, 0.406)
IMAGENET_STD = (0.229, 0.224, 0.225)


def build_transform(input_size: int):
    mean, std = IMAGENET_MEAN, IMAGENET_STD
    return T.Compose(
        [
            T.Lambda(lambda img: img.convert("RGB") if img.mode != "RGB" else img),
            T.Resize((input_size, input_size), interpolation=InterpolationMode.BICUBIC),
            T.ToTensor(),
            T.Normalize(mean=mean, std=std),
        ]
    )


def find_closest_aspect_ratio(aspect_ratio, target_ratios, width, height, image_size):
    best_ratio_diff = float("inf")
    best_ratio = (1, 1)
    area = width * height
    for ratio in target_ratios:
        target_aspect_ratio = ratio[0] / ratio[1]
        ratio_diff = abs(aspect_ratio - target_aspect_ratio)
        if ratio_diff < best_ratio_diff:
            best_ratio_diff = ratio_diff
            best_ratio = ratio
        elif ratio_diff == best_ratio_diff:
            if area > 0.5 * image_size * image_size * ratio[0] * ratio[1]:
                best_ratio = ratio
    return best_ratio


def dynamic_preprocess(image, min_num=1, max_num=12, image_size=448, use_thumbnail=False):
    orig_width, orig_height = image.size
    aspect_ratio = orig_width / orig_height

    target_ratios = set(
        (i, j)
        for n in range(min_num, max_num + 1)
        for i in range(1, n + 1)
        for j in range(1, n + 1)
        if i * j <= max_num and i * j >= min_num
    )
    target_ratios = sorted(target_ratios, key=lambda x: x[0] * x[1])

    target_aspect_ratio = find_closest_aspect_ratio(
        aspect_ratio, target_ratios, orig_width, orig_height, image_size
    )

    target_width = image_size * target_aspect_ratio[0]
    target_height = image_size * target_aspect_ratio[1]
    blocks = target_aspect_ratio[0] * target_aspect_ratio[1]

    resized_img = image.resize((target_width, target_height))
    processed_images = []
    for i in range(blocks):
        box = (
            (i % (target_width // image_size)) * image_size,
            (i // (target_width // image_size)) * image_size,
            ((i % (target_width // image_size)) + 1) * image_size,
            ((i // (target_width // image_size)) + 1) * image_size,
        )
        split_img = resized_img.crop(box)
        processed_images.append(split_img)

    assert len(processed_images) == blocks

    if use_thumbnail and len(processed_images) != 1:
        thumbnail_img = image.resize((image_size, image_size))
        processed_images.append(thumbnail_img)
```

```python

    return processed_images


def load_image(image_file: str, input_size=448, max_num=12):
    image = Image.open(image_file).convert("RGB")
    transform = build_transform(input_size=input_size)
    images = dynamic_preprocess(
        image, image_size=input_size, use_thumbnail=True, max_num=max_num
    )
    pixel_values = [transform(img) for img in images]
    pixel_values = torch.stack(pixel_values)
    return pixel_values


def main():
    # Put any local JPG/PNG here
    img_path = Path("data/test.jpg")
    assert img_path.exists(), f"Place a test image at {img_path}"

    model_id = "OpenGVLab/InternVL3_5-2B-Instruct"

    print("Loading model on GPUs...")
    model = AutoModel.from_pretrained(
        model_id,
        torch_dtype=torch.bfloat16,
        use_flash_attn=False,
        trust_remote_code=True,
        device_map="auto",
    ).eval()

    tokenizer = AutoTokenizer.from_pretrained(
        model_id,
        trust_remote_code=True,
        use_fast=False,
    )

    pixel_values = load_image(str(img_path), max_num=12).to(torch.bfloat16).cuda()

    generation_config = dict(max_new_tokens=256, do_sample=False)

    # Neutral prompt (baseline)
    question_neutral = "<image>\nProvide a factual caption for this image."
    response_neutral = model.chat(
        tokenizer, pixel_values, question_neutral, generation_config
    )
    print("\n[Neutral caption]")
    print(response_neutral)

    # Detailed prompt (what you care about)
    question_detailed = (
        "<image>\nProvide a detailed, comprehensive caption describing all key "
        "objects, attributes, actions, and context."
    )
    response_detailed = model.chat(
        tokenizer, pixel_values, question_detailed, generation_config
    )
    print("\n[Detailed caption]")
    print(response_detailed)


if __name__ == "__main__":
    main()

```

G check_exemplar_leakage.py

```python
import argparse
import json
from pathlib import Path


def load_ids_from_log(path: Path, field: str = "id"):
    assert path.exists(), f"Log file not found: {path}"
    ids = set()
    with path.open("r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
```

```python
            rec = json.loads(line)
            ids.add(int(rec[field]))
    return ids


def main():
    parser = argparse.ArgumentParser(
        description="Check whether eval log contains exemplar IDs."
    )
    parser.add_argument(
        "--eval-log",
        required=True,
        help="Path to eval_*.jsonl file (e.g., logs/eval_subset0_PT_D_FS.jsonl).",
    )
    parser.add_argument(
        "--exemplar-log",
        required=True,
        help=(
            "Path to *_exemplars.jsonl file "
            "(e.g., logs/eval_subset0_PT_D_FS_exemplars.jsonl)."
        ),
    )
    args = parser.parse_args()

    eval_log_path = Path(args.eval_log)
    exemplar_log_path = Path(args.exemplar_log)

    eval_ids = load_ids_from_log(eval_log_path, field="id")
    exemplar_ids = load_ids_from_log(exemplar_log_path, field="id")

    print(f"[INFO] Exemplar IDs      : {sorted(exemplar_ids)}")
    print(f"[INFO] Num exemplar IDs  : {len(exemplar_ids)}")
    print(f"[INFO] Num evaluated IDs : {len(eval_ids)}")

    overlap = exemplar_ids & eval_ids
    if overlap:
        print(f"[WARN] Overlap found! Evaluated exemplar IDs: {sorted(overlap)}")
    else:
        print("[OK] No exemplar IDs appear in the eval log.")


if __name__ == "__main__":
    main()
```