Semantic Image Compression at Ultra-Low Rate via CLIP-Guided Prompt Refinement

Jinheng Zhang, Jialun Yang, Ziao You and Ricardo Li
University of Pennsylvania, Philadelphia, PA 19104

Abstract

We study a simple form of semantic image compression where the encoder sends a short text prompt instead of pixels, and a text-to-image model reconstructs a new image that matches the meaning of the source. Here, "semantic" means we do not require the same layout or exact pixels; we only care that the reconstructed image conveys similar content under a CLIP-based metric. "Compression" means the transmitted representation (a prompt) is extremely small, on the order of a few hundred bytes. We implement an iterative refinement loop using GPT-4.1-mini to propose prompt candidates, DALL-E 3 to decode prompts into images, and CLIP to choose the best prompt/image at each iteration. On 20 complex scene images, we observe consistent CLIP-score gains in early iterations, with qualitative examples showing 5–11% relative improvement.

1 Introduction

Traditional image compression (e.g., JPEG) is designed to preserve visual structure—pixel-level details and spatial layout. In many low-bandwidth settings, however, users may only need a semantically faithful reconstruction: an image that communicates what is happening, even if the exact arrangement differs. This idea is closely related to modern NLP and computational linguistics because we use natural language as a compact carrier of meaning, and we rely on language-conditioned generation to reconstruct images.

Given a reference image, we want to "compress" it into a very short text representation, and then "decompress" by generating an image from that text. The key point is that the reconstruction is not required to match the original pixel-by-pixel. Instead, we evaluate whether the reconstruction is semantically similar. We use CLIP embeddings as a practical semantic similarity measure [6].

Figure 1 shows our overall structure: we treat a caption/prompt as a low-rate semantic code, decode it through DALL-E 3, and then refine the prompt using GPT-4.1-mini guided by CLIP scores. In Figure 3, a city-night scene reconstruction improves from an early iteration to a later one, mainly by capturing better global context.

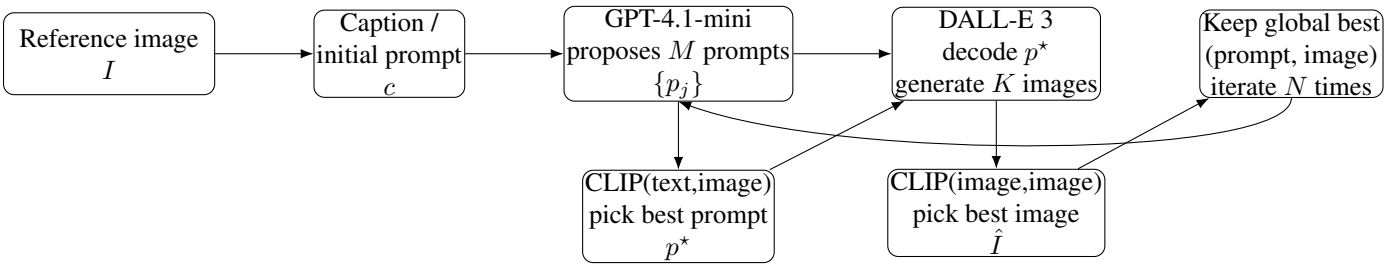

Figure 1: System overview. We use an LLM to propose prompt candidates, a text-to-image model to decode, and CLIP to select the best prompt/image.

Let I be a reference image. A semantic encoder produces a short text message p (a prompt) subject to a rate budget R (e.g., bytes). A decoder G (a text-to-image model) samples a reconstruction $\hat{I} \sim G(p)$. We evaluate semantic fidelity with a similarity function $S(\hat{I}, I)$ computed in an embedding space. Our goal is:

$$\max_{p:\ \mathrm{len}(p) \le R} \ \mathbb{E}_{\hat{I} \sim G(p)} \left[S(\hat{I}, I) \right]. \tag{1}$$

In this project, S is CLIP cosine similarity [6].

We wanted to explore a compression regime where language models and image generators can replace traditional pixel codecs. This is exciting because the rate can become extremely small, and the decoder can still produce plausible, meaningful images. This also connects to NLP directly: the "bitstream" is natural language, and improvement often comes from better prompt engineering/refinement.

2 Literature Review

Traditional image compression focuses on preserving the visual signal under a bitrate constraint, typically measured by bits per pixel (bpp). Classic codecs such as JPEG exploit spatial and frequency redundancies to reduce file size while maintaining acceptable perceptual quality [7]. In parallel, learned image compression replaces hand-designed transforms with neural encoders/decoders and entropy models, often optimizing a rate–distortion objective. A representative example is variational image compression with a hyperprior, where the encoder produces a latent code and an entropy model estimates the number of bits needed to transmit it [1]. These methods are effective when the goal is to reconstruct the same image content and layout with high fidelity, but they are not designed to keep only "meaning" while allowing layout or fine details to change.

In contrast, semantic image compression intentionally relaxes the reconstruction target. The idea is to preserve the core content and high-level meaning of an image, while allowing the decoder to fill in details in a plausible way. An early line of work in this direction is DeepSIC, which argues that compression can benefit from semantic representations and that task-relevant information can be prioritized over exact pixels [4]. This framing is especially relevant for settings where the downstream goal is not perfect visual reproduction but correct interpretation, communication, or decision making. In our project, we use the same high-level motivation: we do not require pixel alignment, and we do not enforce exact geometry. We treat semantic similarity as the objective.

A major recent shift is the use of large pretrained generative models as powerful decoders at extremely low rates. Lei et al. propose *Text + Sketch: Image Compression at Ultra Low Rates* [3], where the sender transmits a compact description and a small amount of side information (a sketch-like structural cue) and a pretrained generator reconstructs an image. The key point is that the decoder can be much stronger than a classical codec decoder: it can synthesize missing content based on learned priors. This changes what "compression" looks like. Instead of sending detailed pixels, the encoder sends a lightweight semantic or structural instruction that is sufficient for a strong decoder to generate a realistic image. Our project follows this philosophy, but with an even simpler representation: we transmit only a short prompt. Because of this, we accept that the reconstruction may not match layout, and we evaluate under a semantic metric.

Dotzel et al. push the idea further by studying semantic image compression at extreme bitrates, reporting results in the micro-bits-per-pixel regime [2]. Their work highlights an important point that matches our setting: when the bitrate is extremely small, it is not realistic to preserve detailed structure. Instead, the system should aim to preserve the scene category, salient objects, and overall meaning. They also emphasize using modern multimodal models and text-to-image generators as part of the compression pipeline, which supports the view that the "code" can be natural language. While our system is not tuned to compete with the best reported numbers in that line of work, it is designed in the same spirit: a short text code and a generative decoder, with evaluation based on semantic similarity rather than exact reconstruction.

To evaluate semantic similarity, many recent systems use CLIP or CLIP-derived metrics. CLIP learns a joint embedding space for images and text by contrastive training on large-scale image–text pairs [6]. Because CLIP embeddings capture high-level content and correlate with semantic alignment, CLIP cosine similarity is often used as a practical proxy for semantic fidelity. This is directly aligned with our definition of semantic compression. In our experiments, we use CLIP image embeddings and compute cosine similarity between the reference image and the reconstructed image. Importantly, this choice matches our goal: if the reconstructed image depicts the same type of scene and objects, it should score well even if the viewpoint or layout differs. At the same time, relying on CLIP also introduces known limitations. CLIP can be insensitive to some fine details and can sometimes be "fooled" by stylistic cues. These limitations motivate our conservative approach: we treat CLIP as a useful signal for selection and comparison, but we keep the claims focused on semantic improvement rather than claiming full perceptual equivalence.

Another relevant research direction is iterative refinement for improving model outputs. In NLP, refinement loops are a common strategy to boost quality without changing the base model. Self-Refine is a clear example: an LLM generates an output, critiques it, and then improves it in multiple rounds, often yielding better final results than a single pass [5]. While Self-Refine is not an image compression method, the core idea is highly compatible with our project: if we can define a feedback signal for whether an output is "better," we can do iterative search rather than a one-shot prediction. In our pipeline, we adapt this to a multimodal setting. We do not ask the model to write free-form critiques. Instead, we use an automatic scalar feedback signal from CLIP to select better prompt candidates and better decoded images over iterations. This makes the refinement loop simple, reproducible, and easy to implement in a small project codebase.

Putting these threads together, our project sits at the intersection of (i) low-rate compression, (ii) semantic fidelity instead of pixel fidelity, (iii) powerful pretrained generative decoders, and (iv) iterative refinement guided by an external metric. Traditional codecs and learned transform codecs show how to formalize rate constraints and optimize compression [7, 1]. Semantic compression work motivates relaxing the distortion definition so that meaning can be prioritized [4]. Ultra-low-rate approaches based on generative decoding demonstrate that text-like representations can support plausible reconstructions at rates far below classical regimes [3, 2]. Finally, iterative refinement work from NLP suggests a practical way to improve results without end-to-end training, as long as we can define a usable feedback signal [5]. Our method is a small, concrete instantiation of these ideas: we compress an image into a short prompt, decode it with a text-to-image model, and refine the prompt through a CLIP-guided loop to improve semantic similarity under a strict rate budget.

3 Experimental Design

3.1 Data

We evaluate on a small dataset of 20 complex scene images. Due to submission size limits, only a 4-image to 5-image subset is included in the final submission package; the four example categories we show most clearly are: city street scenes, living rooms, playground scenes, and airports. Each image has a short caption in `captions.json` used as the initial semantic description.

Item	Value
Total evaluation images	20
Results included in submission	5
Image types	complex scenes (city / indoor / public spaces)
Resolution used for decoding	1024×1024

Table 1: Dataset summary.

3.2 Evaluation Metric

We use CLIP cosine similarity as our semantic quality measure. Let $f_{\text{img}}(I) \in \mathbb{R}^d$ be the CLIP image embedding. We normalize embeddings to unit norm and compute:

$$\text{CLIPSim}(I, \hat{I}) = \frac{f_{\text{img}}(I)^\top f_{\text{img}}(\hat{I})}{\|f_{\text{img}}(I)\| \cdot \|f_{\text{img}}(\hat{I})\|}. \tag{2}$$

In practice we use CLIP ViT-B/32 [6]. This matches our semantic goal: we do not require pixel alignment, only embedding-level similarity.

3.3 Ultra-Low Rate

We treat the prompt string as the compressed representation. If the prompt is B bytes and the reconstructed image is $H \times W$, then a simple bits-per-pixel estimate is:

$$\text{bpp}_{\text{text}} \approx \frac{8B}{HW}. \tag{3}$$

In our runs, prompts are intentionally short, typically a few hundred bytes. At 1024×1024, even $B = 600$ bytes gives ≈ 0.0046 bpp, which is far below classical pixel codecs that are commonly discussed around $\mathcal{O}(1)$ bpp for standard regimes [2]. We do not claim SOTA; we use this analysis mainly to show we are operating in the intended ultra-low rate regime.

3.4 Baselines

We reuse the baselines from our earlier milestones:

- Low-resolution baseline: downsample the reference to very low resolution and upsample back. This preserves coarse semantics but loses details.
- JPEG baseline: JPEG compression at low quality as a strong classical baseline [7].

Method	CLIPSim (from Milestone 2)
Low-res baseline	0.8521
JPEG baseline (low quality)	0.9256

Table 2: Baseline CLIPSim results carried over from Milestone 2. These baselines operate on pixels; our prompt-based method operates at far lower rate and targets semantic similarity rather than pixel fidelity.

3.5 Our Refinement Algorithm

Algorithm 1 formalizes the exact logic in our code. We use:

- Refiner LLM: GPT-4.1-mini
- Generator: DALL-E 3 with `style="natural"` and `quality="hd"`
- CLIP model: ViT-B/32

Key hyperparameters: number of iterations $N = 4$, prompts per iteration $M = 3$, images per prompt $K = 4$.

Algorithm 1 CLIP-guided multi-prompt refinement

Require: reference image I, caption c, iterations N, prompt batch M, image batch K

$v \leftarrow \text{normalize}(f_{\text{img}}(I))$
best_score $\leftarrow -\infty$, best_prompt $\leftarrow$ None
for $t = 1$ to N **do**
  if $t = 1$ **then**
    $\{p_j\}_{j=1}^{M} \leftarrow \text{LLMGenerate}(c)$
  else
    $\{p_j\}_{j=1}^{M} \leftarrow \text{LLMVariations}(c, \text{best\_prompt})$
  end if
  score each prompt via CLIP(text,image) against v; pick $p^\star$
  generate K images $\{\hat{I}_k\}_{k=1}^{K} \sim G(p^\star)$ (DALL-E 3)
  score each image $\hat{I}_k$ via $\text{CLIPSim}(I, \hat{I}_k)$; pick best $\hat{I}^\star$
  if $\text{CLIPSim}(I, \hat{I}^\star) >$ best_score **then**
    best_score $\leftarrow \text{CLIPSim}(I, \hat{I}^\star)$
    best_prompt $\leftarrow p^\star$
  end if
end for
return best_prompt, best_score

When prompts exceed CLIP's context limit, we fall back to chunking the prompt and taking the mean CLIP score over chunks. In our main experiments, prompts are short on purpose for lower rates, but this safeguard avoids silent truncation.

4 Experimental Results

4.1 Main observations on 20 images

We run $N = 4$ iterations because empirically the best improvements often happen in the first few rounds. On our 20-image dataset, we observe:

- About half the images show a steady CLIPSim increase during the first 3 iterations.
- The other half is not strictly monotonic, but often achieves its best score at iteration 2 or 4.

Overall, this supports that refinement improves semantic quality under our metric, even if the trajectory is sometimes noisy due to stochastic decoding.

4.2 CLIPSim trajectories on representative examples

Figure 2 plots CLIPSim across iterations for four representative images. The gains are strongest early, which matches our decision to use only a few iterations.

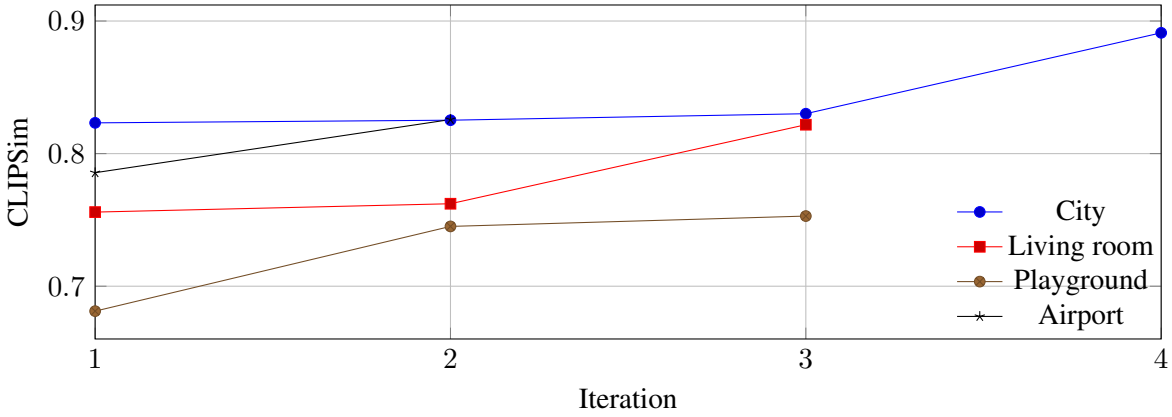

Figure 2: CLIPSim across iterations for four examples. The strongest improvements are typically within the first few iterations.

4.3 Qualitative results

We include four representative collages below. Each collage shows the initial image, iteration-wise best reconstructions, and the CLIP score improvements. We see consistent semantic improvements (better scene type, lighting mood, and key objects), while layout is allowed to drift, consistent with our semantic definition.

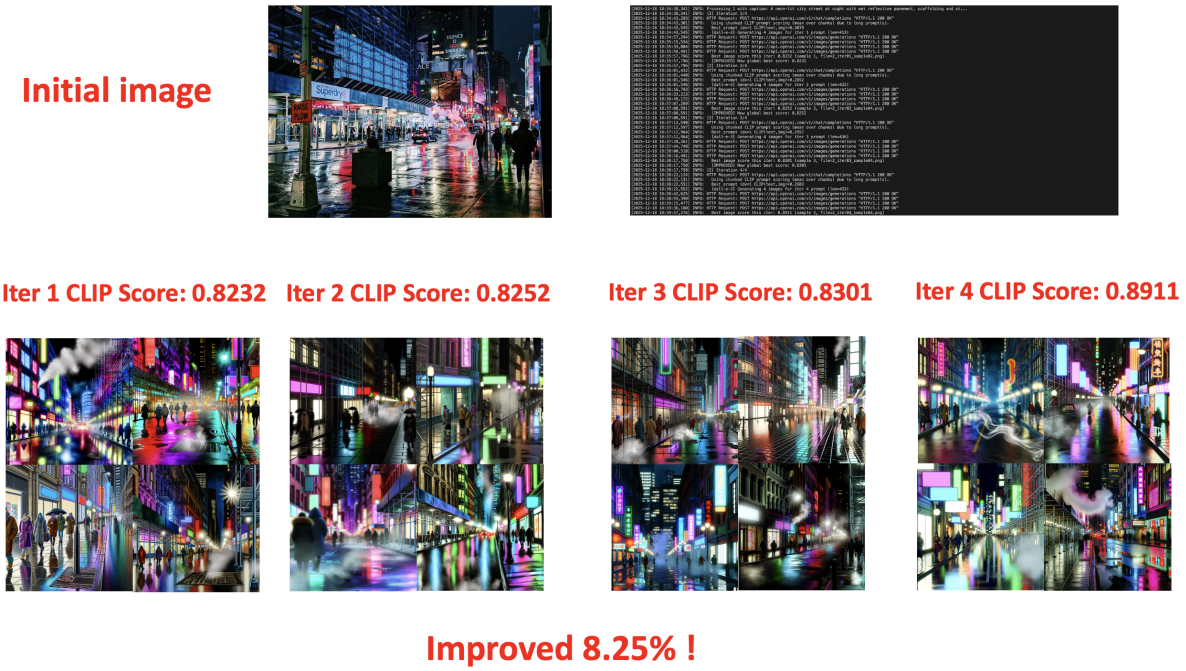

Figure 3: City example: improves by 8.25% relative CLIPSim across iterations.

Figure 4: Living room example: improves by 8.72% relative CLIPSim across iterations.

Figure 5: Playground example: improves by 10.53% relative CLIPSim across iterations.

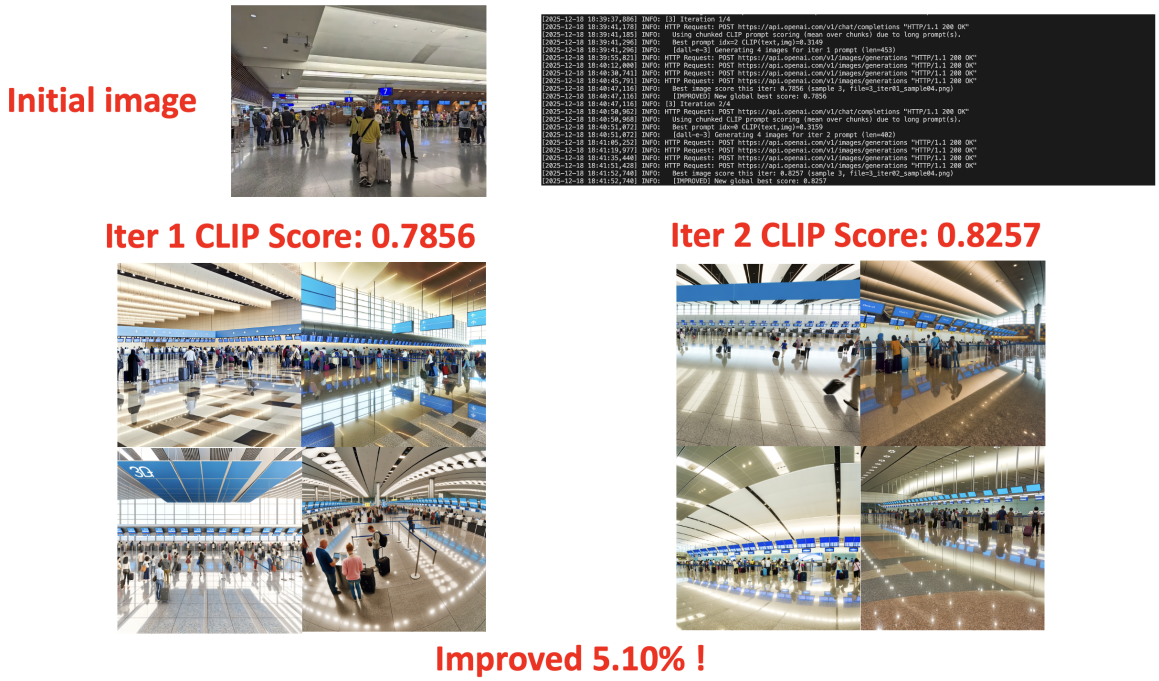

Figure 6: Airport example: improves by 5.10% relative CLIPSim across iterations.

Compared to our two pixel-based baselines, our method trades off some CLIP similarity for a much lower rate. In Milestone 2, the low-resolution resize baseline (short side $= 32$) achieves $\mathrm{CLIPSim} = 0.8521$, while the strong JPEG baseline (quality $= 10$) achieves $\mathrm{CLIPSim} = 0.9256$. These baselines preserve substantial layout and pixel structure because they transmit pixels (just fewer of them, or more coarsely coded), so it is not surprising that their CLIPSim values are higher. In contrast, our semantic compression code is only a short prompt: in our runs, prompts are typically a few hundred UTF-8 bytes (e.g., on the order of 400–600 bytes), which corresponds to an ultra-low rate of roughly $\mathrm{bpp}_{\mathrm{text}} \approx 8B/(HW)$ (for 1024×1024, this is about 0.0046 bpp). Even the low-res baseline implicitly requires sending a 32×32 RGB thumbnail; in raw form that is $32 \cdot 32 \cdot 3 = 3072$ bytes, i.e., about 0.023 bpp when normalized by a 1024×1024 target resolution, already several times larger than our prompt-only rate, and JPEG files are typically much larger still. Under this strict rate constraint, our refinement loop improves semantic fidelity over iterations (e.g., relative gains of about 5%–11% on representative examples), showing that we can recover meaning more reliably without increasing the transmitted rate.

4.4 Extensions

Our main extension in code is the conditional chunked CLIP prompt scoring. It is only triggered when prompts exceed CLIP's context length; then we split the prompt into chunks and report the mean chunk score. This prevents silent truncation artifacts while keeping the rest of the pipeline unchanged.

4.5 Error analysis

From manual inspection of outputs, the most common failure modes are:

- Style drift: DALL-E 3 sometimes becomes overly "illustrative" or cinematic, even when the reference is photographic.

- Object count/details: key entities (e.g., number of people, signage text, small objects) may be wrong, even if the global scene type is correct.
- Layout mismatch: the camera viewpoint and geometry often differ, which is acceptable in our semantic setting but could be problematic if strict structure is required.

Our CLIP-guided loop helps global semantics quickly, but it is less reliable for exact, fine-grained details. This could be addressed by applying the sketch in *Text + Sketch: Image Compression at Ultra Low Rates* [3] so that we could maintain the layout of the images.

5 Conclusions

We implemented a simple semantic image compression prototype where the "bitstream" is a short natural-language prompt and the decoder is a text-to-image model. Using GPT-4.1-mini for prompt proposals, DALL-E 3 for decoding, and CLIP for selection, we observe consistent semantic improvements in the first few refinement iterations on an image dataset of complex scenes. Our system is intentionally optimized for ultra-low rate by keeping prompts short, and we use CLIP similarity to match our definition of semantic fidelity. Next, we plan to improve prompt generation to be more specific without increasing prompt length too much, for example by using better structured prompts and reducing redundant words.

Acknowledgements

We would like to express our appreciation to Professor Mark Yatskar for all his teaching and guidance throughout this semester, and to the CIS 530 course staff, especially Zining Liu, for his detailed, helpful and insightful suggestions. Thank you all!

A Appendix A: Code

```
'''
python refinement.py \
    --ref_dir data/refs \
    --captions data/captions.json \
    --output_dir runs/city_complex_run1 \
    --iterations 4 --M 3 --K 4 \
    --image_model dall-e-3 --image_quality hd
'''

import argparse
import base64
import json
import logging
import os
from io import BytesIO
from pathlib import Path
from typing import Dict, List, Optional, Tuple

import numpy as np
import torch
from PIL import Image
from openai import OpenAI
```

```python
import clip


def list_image_files(ref_dir: Path) -> List[Path]:
    exts = {".png", ".jpg", ".jpeg", ".webp"}
    files = [p for p in ref_dir.iterdir() if p.suffix.lower() in exts]
    return sorted(files)


def ensure_dir(path: Path) -> None:
    path.mkdir(parents=True, exist_ok=True)


def load_captions(captions_path: Optional[Path]) -> Dict[str, str]:
    if captions_path is None:
        logging.warning("No captions JSON provided; falling back to generic captions."
            )
        return {}
    if not captions_path.exists():
        raise FileNotFoundError(f"Captions file not found: {captions_path}")
    with open(captions_path, "r", encoding="utf-8") as f:
        raw = json.load(f)
    captions = {}
    for k, v in raw.items():
        stem = Path(k).stem
        captions[stem] = v
    logging.info(f"Loaded {len(captions)} captions from {captions_path}")
    return captions


def default_caption_for_stem(stem: str) -> str:
    return (
        f"An image named {stem}. You do NOT know its content; "
        f"write a plausible, detailed photographic scene description in English."
    )

def load_clip_model(model_name: str, device: torch.device):
    logging.info(f"Loading CLIP model {model_name} on {device}...")
    model, preprocess = clip.load(model_name, device=device)
    model.eval()
    return model, preprocess


def encode_ref_image(
    clip_model, preprocess, image_path: Path, device: torch.device
) -> torch.Tensor:
    image = Image.open(image_path).convert("RGB")
    image_tensor = preprocess(image).unsqueeze(0).to(device)
    with torch.no_grad():
        feat = clip_model.encode_image(image_tensor)
        feat = feat / feat.norm(dim=-1, keepdim=True)
    return feat


def compute_prompt_clip_scores(
    prompts: List[str],
    ref_feat: torch.Tensor,
    clip_model,
    device: torch.device,
```

```
) -> List[float]:
    if not prompts:
        return []

    tokens = clip.tokenize(prompts, truncate=True).to(device)

    with torch.no_grad():
        text_features = clip_model.encode_text(tokens)
        text_features = text_features / text_features.norm(dim=-1, keepdim=True)
        sims = (ref_feat @ text_features.T).squeeze(0)

    scores = sims.detach().cpu().numpy().tolist()
    return [float(s) for s in scores]



def _prompt_fits_clip_context(prompt: str) -> bool:
    try:
        clip.tokenize([prompt], truncate=False)
        return True
    except Exception:
        return False


def split_text_into_clip_chunks(text: str) -> List[str]:
    words = text.strip().split()
    if not words:
        return [""]
    chunks: List[str] = []
    cur_words: List[str] = []
    for w in words:
        candidate = " ".join(cur_words + [w]).strip()
        if _prompt_fits_clip_context(candidate):
            cur_words.append(w)
        else:
            if cur_words:
                chunks.append(" ".join(cur_words))
                cur_words = [w]
            else:
                chunks.append(w)
                cur_words = []
    if cur_words:
        chunks.append(" ".join(cur_words))
    return [c for c in (ch.strip() for ch in chunks) if c] or [text.strip()]


def compute_prompt_clip_scores_chunked_mean(
    prompts: List[str],
    ref_feat: torch.Tensor,
    clip_model,
    device: torch.device,
) -> List[float]:

    if not prompts:
        return []

    all_chunks: List[str] = []
    spans: List[Tuple[int, int]] = []
    for p in prompts:
```

```
        ch = split_text_into_clip_chunks(p)
        s = len(all_chunks)
        all_chunks.extend(ch)
        e = len(all_chunks)
        spans.append((s, e))

    tokens = clip.tokenize(all_chunks, truncate=False).to(device)

    with torch.no_grad():
        text_features = clip_model.encode_text(tokens)
        text_features = text_features / text_features.norm(dim=-1, keepdim=True)
        sims = (ref_feat @ text_features.T).squeeze(0)

    sims_np = sims.detach().cpu().numpy()

    prompt_scores: List[float] = []
    for (s, e) in spans:
        chunk_scores = sims_np[s:e]
        prompt_scores.append(float(np.mean(chunk_scores)) if len(chunk_scores) > 0
            else float("-inf"))

    return prompt_scores


def compute_prompt_clip_scores_auto(
    prompts: List[str],
    ref_feat: torch.Tensor,
    clip_model,
    device: torch.device,
) -> List[float]:

    use_chunked = any(not _prompt_fits_clip_context(p) for p in prompts)
    if use_chunked:
        logging.info("  Using chunked CLIP prompt scoring (mean over chunks) due to
            long prompt(s).")
        return compute_prompt_clip_scores_chunked_mean(
            prompts=prompts, ref_feat=ref_feat, clip_model=clip_model, device=device
        )
    return compute_prompt_clip_scores(
        prompts=prompts, ref_feat=ref_feat, clip_model=clip_model, device=device
    )

def compute_image_clip_scores(
    image_paths: List[Path],
    ref_feat: torch.Tensor,
    clip_model,
    preprocess,
    device: torch.device,
    batch_size: int = 8,
) -> List[float]:
    scores: List[float] = []

    if not image_paths:
        return scores

    with torch.no_grad():
        for i in range(0, len(image_paths), batch_size):
            batch = image_paths[i : i + batch_size]
            images = [
```

```
                preprocess(Image.open(p).convert("RGB")) for p in batch
            ]
            image_tensor = torch.stack(images).to(device)
            feats = clip_model.encode_image(image_tensor)
            feats = feats / feats.norm(dim=-1, keepdim=True)
            sims = (feats @ ref_feat.T).squeeze(1)  # (batch,)
            scores.extend(sims.detach().cpu().numpy().tolist())

    return [float(s) for s in scores]

def build_openai_client() -> OpenAI:
    api_key = os.environ.get("OPENAI_API_KEY")
    if not api_key:
        raise RuntimeError("OPENAI_API_KEY not set in environment.")
    client = OpenAI(api_key=api_key)
    return client


def _extract_json_object(text: str) -> Optional[dict]:
    start = text.find("{")
    end = text.rfind("}")
    if start == -1 or end == -1 or end <= start:
        return None
    try:
        snippet = text[start : end + 1]
        return json.loads(snippet)
    except Exception:
        return None


def generate_prompts_with_gpt41mini(
    client: OpenAI,
    caption: str,
    base_prompt: Optional[str],
    num_prompts: int,
    mode: str = "initial",
    model_name: str = "gpt-4.1-mini",
    temperature: float = 0.7,
    max_tokens: int = 512,
) -> List[str]:
    assert mode in {"initial", "variation"}

    system_msg = (
        "You are an expert visual prompt engineer for state-of-the-art "
        "text-to-image models (such as GPT-4o image generation). "
        "You write precise, concise English prompts (< 200 words) that fully "
        "describe a scene for high-fidelity image synthesis."
    )

    if mode == "initial":
        user_msg = f"""
We want to reconstruct a reference photograph described by this caption:

\"\"\"{caption}\"\"\".

Generate {num_prompts} diverse candidate prompts that could reproduce this scene.
Each prompt should be self-contained and suitable to send directly to an image
generator. Focus on global scene, key objects with attributes, background,
lighting, composition, and style. Avoid mentioning 'reference' or 'caption'.
```

```
Return ONLY a JSON object of the form:
{{
  "prompts": ["...", "...", ...]
}}
with exactly {num_prompts} strings.
"""
    else:
        base_prompt_text = base_prompt or ""
        user_msg = f"""
We want to refine a current best text-to-image prompt to better match the
following target scene description:

TARGET CAPTION:
\"\"\"{caption}\"\"\".

CURRENT BEST PROMPT:
\"\"\"{base_prompt_text}\"\"\".

Generate {num_prompts} small variations of the BEST prompt that might improve
alignment with the target scene. Make MINIMAL edits: add or adjust details,
clarify composition, or slightly tweak style. Do NOT drastically change the
scene or introduce new objects not implied by the caption.

Return ONLY a JSON object of the form:
{{
  "prompts": ["...", "...", ...]
}}
with exactly {num_prompts} strings.
"""

    response = client.chat.completions.create(
        model=model_name,
        messages=[
            {"role": "system", "content": system_msg},
            {"role": "user", "content": user_msg},
        ],
        temperature=temperature,
        max_tokens=max_tokens,
    )

    text = response.choices[0].message.content or ""
    data = _extract_json_object(text)
    prompts: List[str] = []

    if data and isinstance(data, dict) and "prompts" in data:
        raw = data["prompts"]
        if isinstance(raw, list):
            prompts = [str(p).strip() for p in raw if str(p).strip()]

    if not prompts:
        logging.warning("Failed to parse JSON 'prompts'; using line-based fallback.")
        lines = [ln.strip() for ln in text.splitlines() if ln.strip()]
        for ln in lines:
            if ln[0].isdigit() and "." in ln:
                ln = ln.split(".", 1)[1].strip()
            if ln.startswith("- "):
                ln = ln[2:].strip()
            if ln:
```

```
                prompts.append(ln)

    seen = set()
    unique_prompts = []
    for p in prompts:
        if p not in seen:
            seen.add(p)
            unique_prompts.append(p)
        if len(unique_prompts) >= num_prompts:
            break

    if not unique_prompts:
        logging.error("Could not recover any prompts from GPT-4.1-mini; falling back."
            )
        if base_prompt:
            unique_prompts = [base_prompt] * num_prompts
        else:
            unique_prompts = [caption] * num_prompts

    while len(unique_prompts) < num_prompts:
        unique_prompts.append(unique_prompts[-1])

    return unique_prompts


def generate_images_gpt_image_1(
    client: OpenAI,
    prompt: str,
    image_model: str,
    out_dir: Path,
    stem: str,
    iter_idx: int,
    K: int,
    size: str = "1024x1024",
    quality: str = "high",
) -> List[Path]:
    ensure_dir(out_dir)
    image_paths: List[Path] = []

    logging.info(
        f"  [{image_model}] Generating {K} images for iter {iter_idx} prompt (len={len
            (prompt)})"
    )

    if image_model == "gpt-image-1":
        resp = client.images.generate(
            model=image_model,
            prompt=prompt,
            n=K,
            size=size,
            quality=quality,
        )
        for j, d in enumerate(resp.data):
            b64 = d.b64_json
            img_bytes = base64.b64decode(b64)
            img = Image.open(BytesIO(img_bytes)).convert("RGB")
            fname = out_dir / f"{stem}_iter{iter_idx:02d}_sample{j+1:02d}.png"
            img.save(fname)
            image_paths.append(fname)
```

```
    else:
        for j in range(K):
            resp = client.images.generate(
                model=image_model,
                prompt=prompt,
                n=1,
                size=size,
                quality=quality,
                style="natural",
                response_format="b64_json",
            )
            d = resp.data[0]
            b64 = d.b64_json
            img_bytes = base64.b64decode(b64)
            img = Image.open(BytesIO(img_bytes)).convert("RGB")
            fname = out_dir / f"{stem}_iter{iter_idx:02d}_sample{j+1:02d}.png"
            img.save(fname)
            image_paths.append(fname)

    return image_paths

def run_multi_prompt_loop_for_image(
    client: OpenAI,
    clip_model,
    preprocess,
    device: torch.device,
    image_path: Path,
    caption: str,
    out_root: Path,
    iterations: int,
    M: int,
    K: int,
    refiner_model: str,
    image_model: str,
    image_size: str,
    image_quality: str,
    accept_threshold: float,
    early_stop_patience: int,
    early_stop_min_delta: float,
) -> Tuple[str, Path, float]:
    stem = image_path.stem
    example_dir = out_root / stem
    images_dir = example_dir / "images"
    ensure_dir(example_dir)
    ensure_dir(images_dir)

    ref_img = Image.open(image_path).convert("RGB")
    ref_copy_path = example_dir / "ref.png"
    ref_img.save(ref_copy_path)

    ref_feat = encode_ref_image(clip_model, preprocess, image_path, device)

    logs: List[dict] = []

    global_best_image_score = -1e9
    global_best_prompt: Optional[str] = None
    global_best_image_path: Optional[Path] = None
```

```
    last_improvement_score = -1e9
    no_improve_steps = 0

    logging.info(f"Processing {stem} with caption: {caption[:80]}...")

    for iter_idx in range(1, iterations + 1):
        logging.info(f"[{stem}] Iteration {iter_idx}/{iterations}")

        if iter_idx == 1 or global_best_prompt is None:
            prompts = generate_prompts_with_gpt41mini(
                client=client,
                caption=caption,
                base_prompt=None,
                num_prompts=M,
                mode="initial",
                model_name=refiner_model,
            )
        else:
            prompts = generate_prompts_with_gpt41mini(
                client=client,
                caption=caption,
                base_prompt=global_best_prompt,
                num_prompts=M,
                mode="variation",
                model_name=refiner_model,
            )

        prompt_scores = compute_prompt_clip_scores_auto(
            prompts=prompts,
            ref_feat=ref_feat,
            clip_model=clip_model,
            device=device,
        )
        assert len(prompts) == len(prompt_scores)
        best_prompt_idx = int(np.argmax(prompt_scores))
        current_prompt = prompts[best_prompt_idx]
        logging.info(
            f"  Best prompt idx={best_prompt_idx} CLIP(text,img)={prompt_scores[
                best_prompt_idx]:.4f}"
        )

        batch_image_paths = generate_images_gpt_image_1(
            client=client,
            prompt=current_prompt,
            image_model=image_model,
            out_dir=images_dir,
            stem=stem,
            iter_idx=iter_idx,
            K=K,
            size=image_size,
            quality=image_quality,
        )

        image_scores = compute_image_clip_scores(
            image_paths=batch_image_paths,
            ref_feat=ref_feat,
            clip_model=clip_model,
            preprocess=preprocess,
            device=device,
```

```
        )
        assert len(batch_image_paths) == len(image_scores)
        iter_best_idx = int(np.argmax(image_scores))
        iter_best_score = float(image_scores[iter_best_idx])
        iter_best_image_path = batch_image_paths[iter_best_idx]

        epoch_rec_dir = out_root / f"epoch{iter_idx:02d}_rec"
        ensure_dir(epoch_rec_dir)
        epoch_best_path = epoch_rec_dir / f"{stem}.png"
        Image.open(iter_best_image_path).save(epoch_best_path)

        logging.info(
            f"  Best image score this iter: {iter_best_score:.4f} "
            f"(sample {iter_best_idx}, file={iter_best_image_path.name})"
        )

        improved = False
        if iter_best_score > global_best_image_score + accept_threshold:
            improved = True
            global_best_image_score = iter_best_score
            global_best_prompt = current_prompt
            global_best_image_path = iter_best_image_path
            logging.info(
                f"  [IMPROVED] New global best score: {global_best_image_score:.4f}"
            )

        logs.append(
            {
                "iter": iter_idx,
                "prompts": prompts,
                "prompt_scores": prompt_scores,
                "chosen_prompt_idx": best_prompt_idx,
                "chosen_prompt": current_prompt,
                "image_paths": [str(p) for p in batch_image_paths],
                "image_scores": image_scores,
                "iter_best_idx": iter_best_idx,
                "iter_best_score": iter_best_score,
                "global_best_score": global_best_image_score,
                "improved": improved,
            }
        )

        if improved:
            if global_best_image_score > last_improvement_score + early_stop_min_delta
                :
                last_improvement_score = global_best_image_score
                no_improve_steps = 0
            else:
                no_improve_steps += 1
        else:
            no_improve_steps += 1

        if (
            early_stop_patience > 0
            and no_improve_steps >= early_stop_patience
        ):
            logging.info(
                f"  Early stopping after {iter_idx} iterations "
                f"(no significant improvement for {no_improve_steps} steps)."
```

```
            )
            break

    if global_best_prompt is None or global_best_image_path is None:
        logging.warning(
            f"[{stem}] No improvement found; falling back to last iteration's best."
        )
        global_best_prompt = current_prompt
        global_best_image_path = iter_best_image_path
        global_best_image_score = iter_best_score

    with open(example_dir / "final_prompt.txt", "w", encoding="utf-8") as f:
        f.write(global_best_prompt)

    summary = {
        "image_id": stem,
        "caption": caption,
        "final_prompt": global_best_prompt,
        "final_image": str(global_best_image_path),
        "final_clip_score": global_best_image_score,
        "iterations_run": len(logs),
        "logs": logs,
    }
    with open(example_dir / "summary.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, indent=2)

    logging.info(
        f"[{stem}] Finished. Global best CLIPSim={global_best_image_score:.4f}, "
        f"image={global_best_image_path.name}"
    )

    return global_best_prompt, global_best_image_path, global_best_image_score

def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description=(
            "Multi-prompt CLIP-guided refinement with GPT-4.1-mini (refiner) "
            "and GPT-4o image generation (gpt-image-1)."
        )
    )
    parser.add_argument(
        "--ref_dir",
        type=str,
        required=True,
        help="Directory containing reference images (PNG/JPG).",
    )
    parser.add_argument(
        "--captions",
        type=str,
        default=None,
        help="Optional JSON file mapping image stems to captions.",
    )
    parser.add_argument(
        "--output_dir",
        type=str,
        required=True,
        help="Directory where refinement runs will be stored.",
    )
    parser.add_argument(
```

```
        "--iterations",
        type=int,
        default=4,
        help="Max number of refinement iterations per image (N).",
    )
    parser.add_argument(
        "--K",
        type=int,
        default=4,
        help="Number of images per iteration (image batch size).",
    )
    parser.add_argument(
        "--M",
        type=int,
        default=4,
        help="Number of candidate prompts per iteration (prompt batch size).",
    )
    parser.add_argument(
        "--clip_model",
        type=str,
        default="ViT-B/32",
        help="CLIP model name to use (default: ViT-B/32).",
    )
    parser.add_argument(
        "--device",
        type=str,
        default="auto",
        help='Device for CLIP: "auto", "cpu", "cuda", or "mps".',
    )
    parser.add_argument(
        "--refiner_model",
        type=str,
        default="gpt-4.1-mini",
        help="OpenAI model name for the refiner LLM (default: gpt-4.1-mini).",
    )
    parser.add_argument(
        "--image_model",
        type=str,
        default="dall-e-3",
        help="OpenAI model name for image generation.",
    )
    parser.add_argument(
        "--image_size",
        type=str,
        default="1024x1024",
        help='Image size (e.g., "1024x1024", "1536x1024", "1024x1536").',
    )
    parser.add_argument(
        "--image_quality",
        type=str,
        default="hd",
        help='Image quality ("standard" or "hd").',
    )
    parser.add_argument(
        "--accept_threshold",
        type=float,
        default=0.0,
        help=(
            "Minimum CLIPSim improvement required to accept a new global best "
```

```
            "(hill-climbing). Default: 0.0."
        ),
    )
    parser.add_argument(
        "--early_stop_patience",
        type=int,
        default=0,
        help=(
            "If > 0, stop if there is no significant improvement for this many "
            "consecutive iterations (per image)."
        ),
    )
    parser.add_argument(
        "--early_stop_min_delta",
        type=float,
        default=0.001,
        help="Minimum improvement in CLIPSim considered significant for early stopping
            .",
    )
    parser.add_argument(
        "--max_images",
        type=int,
        default=-1,
        help="Optionally limit the number of reference images processed (for debugging
            ).",
    )

    return parser.parse_args()


def main() -> None:
    args = parse_args()

    logging.basicConfig(
        level=logging.INFO,
        format="[%(asctime)s] %(levelname)s: %(message)s",
    )

    ref_dir = Path(args.ref_dir)
    if not ref_dir.exists():
        raise FileNotFoundError(f"Reference directory not found: {ref_dir}")

    out_root = Path(args.output_dir)
    ensure_dir(out_root)

    captions_path = Path(args.captions) if args.captions else None
    captions = load_captions(captions_path)

    if args.device == "auto":
        if torch.cuda.is_available():
            device = torch.device("cuda")
        elif getattr(torch.backends, "mps", None) and torch.backends.mps.is_available
            ():
            device = torch.device("mps")
        else:
            device = torch.device("cpu")
    else:
        if args.device == "cuda" and not torch.cuda.is_available():
            logging.warning(
```

```
                "CUDA requested but not available; falling back to CPU."
            )
            device = torch.device("cpu")
        elif args.device == "mps" and not (
            getattr(torch.backends, "mps", None)
            and torch.backends.mps.is_available()
        ):
            logging.warning(
                "MPS requested but not available; falling back to CPU."
            )
            device = torch.device("cpu")
        else:
            device = torch.device(args.device)

    logging.info(f"Using device: {device}")

    client = build_openai_client()

    clip_model, preprocess = load_clip_model(args.clip_model, device=device)

    ref_images = list_image_files(ref_dir)
    if args.max_images > 0:
        ref_images = ref_images[: args.max_images]

    if not ref_images:
        logging.error(f"No images found in {ref_dir}")
        return

    logging.info(f"Found {len(ref_images)} reference images in {ref_dir}")

    final_rec_dir = out_root / "final_rec"
    ensure_dir(final_rec_dir)

    best_scores: List[float] = []

    for image_path in ref_images:
        stem = image_path.stem
        caption = captions.get(stem, default_caption_for_stem(stem))

        try:
            best_prompt, best_image_path, best_score = run_multi_prompt_loop_for_image
                (
                client=client,
                clip_model=clip_model,
                preprocess=preprocess,
                device=device,
                image_path=image_path,
                caption=caption,
                out_root=out_root,
                iterations=args.iterations,
                M=args.M,
                K=args.K,
                refiner_model=args.refiner_model,
                image_model=args.image_model,
                image_size=args.image_size,
                image_quality=args.image_quality,
                accept_threshold=args.accept_threshold,
                early_stop_patience=args.early_stop_patience,
                early_stop_min_delta=args.early_stop_min_delta,
```

```
            )
        except Exception as e:
            logging.exception(
                f"Error processing {stem}; skipping this image. Error: {e}"
            )
            continue

        final_path = final_rec_dir / f"{stem}.png"
        Image.open(best_image_path).save(final_path)

        best_scores.append(best_score)

    if best_scores:
        mean_clip = float(np.mean(best_scores))
        logging.info(
            f"Finished all images. Mean best CLIPSim over dataset: {mean_clip:.4f}"
        )
    else:
        logging.warning("No images successfully processed; no scores to report.")


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

References

[1] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. In *International Conference on Learning Representations (ICLR)*, 2018.

[2] Jordan Dotzel, Bahaa Kotb, James Dotzel, Mohamed Abdelfattah, and Zhiru Zhang. Exploring the limits of semantic image compression at micro-bits per pixel. *arXiv preprint arXiv:2402.13536*, 2024.

[3] Eric Lei, Yiğit Berkay Uslu, Hamed Hassani, and Shirin Saeedi Bidokhti. Text + sketch: Image compression at ultra low rates. *arXiv preprint arXiv:2307.01944*, 2023.

[4] Sihui Luo, Yezhou Yang, and Mingli Song. Deepsic: Deep semantic image compression. *arXiv preprint arXiv:1801.09468*, 2018.

[5] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shunyu Yao, Or Honovich, Hannah Zhang, Sean Welleck, Amir Yazdanbakhsh, Yejin Choi, Hannaneh Hajishirzi, and Luke Zettlemoyer. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

[6] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. *arXiv preprint arXiv:2103.00020*, 2021.

[7] Gregory K. Wallace. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.