SPEC-1-Watermark-Remover-Suite MVP

Background

You own the Watermark Remover Suite repo and want to evolve it from scaffolding into a minimally viable, reproducible watermark-removal tool that actually processes images (and optionally videos) end-to-end. The goal is to deliver:

- A reliable **CLI** that can auto-detect common watermarks (static corner logos and overlaid semi-transparent text) and inpaint them with a deterministic baseline.
- An optional **GUI** for manual mask editing and preview, wired to the same pipeline.
- Reproducible **samples**, **tests**, and **benchmarks** to back up claims.
- Clean **packaging** with pinned dependencies and optional model downloads (if advanced inpainting is enabled).

This MVP should emphasize correctness, reproducibility, and transparent limitations (e.g., complex embedded watermarks or dynamic moving overlays).

Requirements

MoSCoW Prioritization

Must Have - **Media types**: Images (**PNG/JPG**) **and** short videos (≤ **60s**, ≤ **1080p**, H.264 MP4 in/out). - **Primary inpainting**: Learning-based methods (**LaMa** and **Stable Diffusion inpainting**) as first-class options. - **Masking (auto)**: Static corner / semi-transparent overlaid text detection with configurable thresholds and morphological refinement. - **Pipeline**: End-to-end CLI flows: - `wmr image in.jpg --out out.jpg --method lama|sd --mask auto` - `wmr video in.mp4 --out out.mp4 --method lama|sd --mask static` - **Reproducibility**: Pinned environment (CPU & CUDA variants), deterministic seeds where applicable, model weight checksums. - **Samples & proofs**: `samples/` with 3–5 images and 1–2 clips, plus expected outputs and exact commands to reproduce. - **Tests**: Unit tests for mask IoU on synthetic overlays; regression tests comparing masked-region SSIM/LPIPS deltas. - **Docs**: Clear README usage, limitations, and legal notice about content licenses.

Should Have - **GUI**: Minimal PyQt/PySide preview app with manual brush/eraser for masks and before/after toggle. - **Model management**: Offline downloader with license text, sha256 checks, and cache directory. - **Config profiles**: Presets like `--profile corner-text`, `--profile tiled-logo`. - **Docker**: CPU-only image; a separate CUDA image if feasible.

Could Have - **Logo detection**: Template/classifier-based detection for common watermark logos. - **Temporal smoothing**: Optical-flow–guided consistency to reduce flicker in video outputs. - **Benchmarking**: Built-in `bench` command that emits SSIM/LPIPS and runtime, with CSV/JSON report. - **Batch mode**: Directory globbing with parallel workers and progress bar.

Won’t Have (MVP) - Robust support for **moving**/animated watermarks or those embedded via heavy compression artifacts. - 4K/long-form videos, streaming inputs, or live camera integration. - Mobile builds or a browser/WebAssembly front end. - Commercial dataset bundling or redistribution of third-party models without license compliance.

Method

Architecture Overview

- **CLI** (`wmr`): orchestrates jobs, parses flags, spawns workers.
- **Core**
- `wm_estimator`: builds global/slowly-varying watermark mask/template.
- `mask_propagator`: optical-flow–guided mask tracking per frame.
- `inpaint_lama` / `inpaint_sd`: learning-based inpainting backends.
- `temporal_guidance`: warps prev clean frames for stable conditioning.
- `seam_blender`: overlap-aware, flow-guided cross-fade between chunks.
- **Video IO**: ffmpeg frame extraction/remux; audio/subtitles passthrough.
- **QC**: warped-SSIM/LPIPS in masked region, re-run on failures.

```
@startuml
skinparam shadowing false
skinparam componentStyle rectangle
actor User
User --> CLI: wmr video input.mp4 ...
component CLI
component VideoIO
component WMEstimator
component MaskPropagator
component TemporalGuidance
component Inpaint(Lama/SD) as Inpaint
component SeamBlender
component QC

CLI --> VideoIO: extract frames
CLI --> WMEstimator: global/low-rank template
WMEstimator --> MaskPropagator: base mask
MaskPropagator --> CLI: per-frame masks
CLI --> TemporalGuidance: prev_clean, flow
TemporalGuidance --> Inpaint: guided init/cond
Inpaint --> SeamBlender: chunk outputs
SeamBlender --> QC: overlap frames
QC --> CLI: metrics / re-run flags
CLI --> VideoIO: mux frames+audio -> output
@enduml
```

Algorithms (concise)

A) Global Watermark Estimation - Input: frames `F[0..N-1]`, ROI ≈ bottom-right or auto-detected via high-contrast semi-transparent overlays. - For animated watermarks, compute **low-rank + sparse** decomposition on ROI stack: - `F_t = B_t + W_t` with `W_t` low-rank in time; solve via incremental PCP (Principal Component Pursuit) on luma or HSV-V. - Produce: base mask `M0`, per-frame alpha-like map `A_t` (optional), and confidence.

B) Mask Propagation - Compute optical flow `Flow(t-1→t)` (e.g., RAFT via torchscript or TV-L1 as CPU fallback). - `M_t = warp(M_{t-1}, Flow)`, then refine with morphological ops and threshold guided by `A_t`.

C) Chunked Processing with Overlap - Window `W=48`, Overlap `O=12`. Chunks: `[0,47], [36,83], …, [252,299]`. - **Warm-start**: for each chunk start `s`, warp last `K=8` cleaned frames from previous chunk into `[s, s+K-1]` as guidance. - **Seam blending**: for overlap frame index `i∈[0,O-1]`, output - `C_i = w_i * Prev_i + (1-w_i) * Curr_i`, where `w_i = 1 - i/(O-1)`; all terms flow-aligned.

D) Inpainting Backends - **LaMa**: feed `image, mask`; set `dilation=3–7` around `M_t`. Batch per chunk; enable tiling for >1080p. - **Stable Diffusion Inpaint**: conditioning with warped prev clean frame; set seed = `hash(video_id)+frame_idx`; steps 20–30; CFG 3–5; negative prompts for `text, logo, watermark` optional.

E) QC & Auto-Repair - Compute **warped-SSIM** between `C_t` and `warp(C_{t-1})` inside the dilated mask. If `< τ_stab` (e.g., 0.92) or edge energy spikes, re-run frame with (a) stronger dilation, (b) higher steps, or (c) heavier seam weight.

Key Data Structures & APIs (sketch)

```
class ChunkJob(NamedTuple):
    start: int; end: int; overlap: int; seed_base: int

@dataclass
class FramePack:
    frames: List[np.ndarray]
    masks:  List[np.ndarray]
    guides: Optional[List[np.ndarray]]  # warped prev-clean

# CLI entry
wmr video in.mp4 --out out.mp4
  --method lama|sd --window 48 --overlap 12
  --wm-estimation lowrank,window=120
  --temporal-guidance flow,K=8
  --seam-blend flow_fade
  --qc warped_ssim>=0.92 --retry 2
```

Minimal File Layout (implementation anchors)

```
core/
  wm_estimation.py    # ROI detect + low-rank decomposition, mask/alpha
  flow.py             # RAFT/TV-L1 wrapper + warping utils
  mask.py             # propagation + refinement
  inpaint_lama.py     # LaMa runner (onnx/torch) with tiling
  inpaint_sd.py       # SD inpaint pipeline with conditioning
  temporal.py         # guidance, seam blending, stabilization
  pipeline.py         # chunk scheduler + orchestration
cli/
  wmr.py              # arg parsing, job dispatch, progress, mux
```

Expected Performance (1080p, 300 frames)

- **LaMa (GPU)**: ~6–12 fps per-frame baseline; with flow/guidance + overlap blending, effective **3–6 fps** end-to-end.
- **SD Inpaint (GPU)**: ~0.8–2 fps baseline; **0.5–1.2 fps** with temporal extras.
- CPU runs are possible but primarily for images/smaller batches.

Implementation

Defaults per your hardware

- **Optical flow**: **RAFT (Torch, GPU)** as primary; **TV-L1 (OpenCV, CPU)** fallback.
- **Inpainting**: **LaMa** default; **Stable Diffusion Inpaint** as opt-in via `--method sd`.

Tech stack & pins (initial recommendations)

- Python 3.11, PyTorch (CUDA 11.8/12.x build matching 3060 drivers), torchvision.
- RAFT weights (TorchScript or standard PyTorch ckpt) cached under `~/.wmr/models/`.
- LaMa: either **torch** or **onnxruntime-gpu** export; start with Torch for simplicity.
- OpenCV (>=4.8), NumPy, ffmpeg (system binary), LPIPS, scikit-image.

Repo structure (concrete)

```
watermark_remover/
  core/
    wm_estimation.py
    flow.py
    mask.py
    inpaint_lama.py
    inpaint_sd.py
    temporal.py
    pipeline.py
  cli/
```

```
    wmr.py
  models/
    download_models.py
    README.md
  qc/
    metrics.py  # warped-SSIM/LPIPS, edge energy
  samples/
    images/  videos/
  tests/
    test_mask.py  test_inpaint.py  test_temporal.py
pyproject.toml
Makefile
README.md
```

Makefile targets (runnable skeleton)

```
setup:  ## create venv, install deps, download models
	uv venv || python -m venv .venv
	. .venv/bin/activate && pip install -U pip
	. .venv/bin/activate && pip install -e .
	python -m watermark_remover.models.download_models --all

verify-sample: ## run on bundled samples
	wmr image samples/images/demo.jpg --out out/demo_lama.jpg --method lama --mask auto
	wmr video samples/videos/demo.mp4 --out out/demo_lama.mp4 --method lama --window 48 --overlap 12 --temporal-guidance flow,K=8 --seam-blend flow_fade

bench: ## emit JSON/CSV report
	wmr bench samples/videos/demo.mp4 --report out/bench.json

test:
	pytest -q
```

CLI behaviors (exact)

- **Image**: `wmr image in.jpg --out out.jpg --method lama --mask auto --dilate 5 --seed 1234`
- **Video**: `wmr video in.mp4 --out out.mp4 --method lama --window 48 --overlap 12 --temporal-guidance flow,K=8 --wm-estimation lowrank,window=120 --qc warped_ssim>=0.92 --retry 2`
- **Switch to SD**: add `--method sd --sd-steps 28 --sd-cfg 4 --sd-cond prev_warp`.

Module notes (what to implement)

- `core/flow.py`

- RAFT inference wrapper (`predict_flow(f_{t-1}, f_t)`) + `warp(image, flow)`; memory-aware batching.
- `core/wm_estimation.py`
- ROI detection (bottom-right heuristic + edge/alpha cues) → `M0`.
- Optional low-rank template over sliding window (`window=120`).
- `core/mask.py`
- `propagate(M_prev, Flow)` + morphology + confidence-threshold using `A_t` when available.
- `core/inpaint_lama.py`
- Tiled inference for >1080p; mask dilation; seed control where applicable.
- `core/inpaint_sd.py`
- Conditioning with warped previous clean frame; deterministic seed schedule; safety to clamp prompts.
- `core/temporal.py`
- Overlap scheduling; seam blending (`flow_fade`); regrain filter.
- `core/pipeline.py`
- Chunk generator; warm-start; retries on QC failure; progress events.
- `qc/metrics.py`
- Warped-SSIM in masked region; LPIPS optional; edge-energy spike detector.

Tests (fast, synthetic)

- **Mask IoU**: render synthetic white text overlay on random images; expect IoU $\geq$ 0.85.
- **Temporal stability**: generate moving background + synthetic watermark; expect $\geq$ 90% frames with warped-SSIM $\geq$ 0.92.
- **Regression**: lock seeds; compare outputs to goldens for samples; tolerance on PSNR/SSIM inside mask.

Samples & outputs

- Provide `samples/videos/demo.mp4` (5–10 s, 1080p) with a scripted watermark generator so you have a ground-truth clean reference for metrics.
- Commit **before/after** thumbnails and the exact commands in README.

Operational tips

- CUDA env probe at startup; refuse SD runs on CPU by default (`--allow-slow` to override).
- Cache flows and masks on disk per chunk to enable resume.
- One final **single-pass encode** for the whole video and **remux original audio/subs**.

Milestones

M0 – Scaffolding hygiene - PyProject/Makefile/CI skeleton; `wmr` CLI stub runs and prints config. - Model downloader caches RAFT/LaMa weights with checksums. - Acceptance: `make setup` completes; `wmr --help` lists planned flags.

M1 – Image MVP (LaMa) - Auto-mask for corner/semi-transparent text; LaMa inpaint with dilation. - Acceptance: `wmr image` transforms all sample images; masked-region SSIM ≥ baseline; visual before/after in README.

M2 – Video core (chunking + RAFT) - Frame extraction/mux; chunk window/overlap; RAFT flow; mask propagation. - Acceptance: `wmr video` produces an output for demo clip; no seam jumps on overlap frames by visual check.

M3 – Temporal guidance & seam blending - Warm-start with warped prev-clean; flow-aware cross-fade; soft regrain. - Acceptance: ≥ 80% of frames pass warped-SSIM ≥ 0.92 on demo; seam artifacts eliminated.

M4 – QC + auto-retry - Warped-SSIM/edge-energy gate; up to N retries with adjusted params. - Acceptance: demo run reports "clean frames: X/Total" and exceeds target threshold.

M5 – Optional SD inpainting path - SD inpaint with conditioning; seeds per frame; safety clamps. - Acceptance: tough sample improves masked-region LPIPS vs. LaMa.

M6 – GUI (minimal) - PyQt/PySide: load, draw mask, preview, export; calls same pipeline. - Acceptance: user can refine mask and export a new result from GUI.

M7 – Benchmarks & docs - `wmr bench` emits JSON/CSV; README table with reproducible commands. - Acceptance: CI uploads bench artifact and image thumbnails.

M8 – Packaging & hardening - Docker images (CPU and CUDA); error messages; resume support; logging. - Acceptance: one-line docker run reproduces demo outputs.

Gathering Results

Metrics - **Mask quality (synthetic)**: IoU ≥ 0.85 on rendered overlays. - **Temporal stability (video)**: fraction of frames with masked-region **warped-SSIM ≥ 0.92**; report "clean frames / total". - **Perceptual quality**: masked-region **LPIPS** vs. previous cleaned frame ≤ 0.12 (lower is better). - **Artifact guard**: edge-energy spike detector on masked band; rate ≤ 2% of frames. - **Performance**: effective fps (frames / wall-clock) for LaMa and SD paths.

Datasets - **Synthetic**: scripted watermark generator that outputs (clean, watermarked) pairs for images and a short video → enables ground-truth metrics. - **Realistic samples**: 3 images + 2 clips under CC or your own footage; varied backgrounds and compression.

Reporting - CLI emits `run.json` with config, versions, metrics, and per-frame QC. - `bench/plot.ipynb` (optional) renders stability histograms and seam heatmaps. - README contains a compact table and thumbnails with exact commands.

Acceptance gates (MVP) - Image pipeline: passes mask IoU gate and produces visually acceptable inpaints on all samples. - Video pipeline: ≥ 85% "clean frames" on the demo clip, no visible seam pops, audio/subs preserved.

Post-release checks - Determinism: reruns with same seed schedule yield byte-identical masks and near-identical outputs (within codec tolerance). - Security/licensing: model hashes match; third-party licenses included; no unexpected network calls.

Starter Files (Drop-in)

> Copy these into your repo as-is to bootstrap the environment, QC metrics, and a synthetic watermark generator.

`pyproject.toml`

```
[build-system]
requires = ["setuptools>=68", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "watermark-remover-suite"
version = "0.1.0"
description = "Watermark Remover Suite — LaMa/SD inpainting with temporal
guidance"
authors = [{ name = "Blueibear" }]
readme = "README.md"
requires-python = ">=3.11"
dependencies = [
  "numpy>=1.26",
  "opencv-python>=4.8",
  "scikit-image>=0.23",
  "lpips>=0.1.4",
  "Pillow>=10.0",
  "tqdm>=4.66",
  "rich>=13.7",
  "ffmpeg-python>=0.2.0",
  "torch>=2.2; platform_system != 'Darwin'",
# pin to CUDA build manually in README
  "torchvision>=0.17; platform_system != 'Darwin'",
  "einops>=0.7",
]

[project.optional-dependencies]
# Stable Diffusion inpaint (optional)
sd = [
  "diffusers[torch]>=0.30",
  "transformers>=4.43",
  "accelerate>=0.33",
  "xformers>=0.0.27; platform_system == 'Linux'",
]
```

```toml
# ONNX LaMa path (optional alternative to Torch)
onx = ["onnxruntime-gpu>=1.18"]

gui = ["PySide6>=6.6"]

develop = ["pytest>=8.1", "black>=24.3", "ruff>=0.5", "mypy>=1.10"]

[project.scripts]
wmr = "watermark_remover.cli.wmr:main"

[tool.setuptools.packages.find]
where = ["."]
include = ["watermark_remover*"]

[tool.black]
line-length = 100

[tool.ruff]
line-length = 100
select = ["E", "F", "I"]
```

Makefile

```makefile
.PHONY: setup dev install download-models verify-sample bench test format lint
clean

VENV := .venv
PY := $(VENV)/bin/python
PIP := $(VENV)/bin/pip

setup:
    python -m venv $(VENV)
    $(PIP) install --upgrade pip
    $(PIP) install -e .[develop]

install:
    $(PIP) install -e .

format:
    $(VENV)/bin/black .
    $(VENV)/bin/ruff check --fix .

lint:
    $(VENV)/bin/ruff check .

 test:
```

```
    $(VENV)/bin/pytest -q

 download-models:
    $(PY) -m watermark_remover.models.download_models --all

 verify-sample:
    # image demo (placeholder; assumes pipeline exists)
    wmr image samples/images/demo.jpg --out out/demo_lama.jpg --method lama --
mask auto || true
    # video demo (placeholder)
    wmr video samples/videos/demo.mp4 --out out/demo_lama.mp4 --method lama --
window 48 --overlap 12 --temporal-guidance flow,K=8 --seam-blend flow_fade ||
true

 bench:
    wmr bench samples/videos/demo.mp4 --report out/bench.json || true

clean:
    rm -rf $(VENV) .pytest_cache .ruff_cache build dist *.egg-info out/**
```

`watermark_remover/qc/metrics.py`

```python
from __future__ import annotations
import json
from dataclasses import dataclass
from pathlib import Path
from typing import Optional, Tuple

import cv2
import lpips  # type: ignore
import numpy as np
from skimage.metrics import structural_similarity as ssim

# ---- Utilities ----

def _to_gray(img: np.ndarray) -> np.ndarray:
    if img.ndim == 2:
        return img
    return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)


def _ensure_uint8(img: np.ndarray) -> np.ndarray:
    if img.dtype == np.uint8:
        return img
    img = np.clip(img, 0, 1) if img.max() <= 1.0 else np.clip(img, 0, 255)
    return (img * 255).astype(np.uint8) if img.max() <= 1.0 else
```

```
img.astype(np.uint8)


def _farneback_flow(prev: np.ndarray, curr: np.ndarray) -> np.ndarray:
    prev_g = _to_gray(_ensure_uint8(prev))
    curr_g = _to_gray(_ensure_uint8(curr))
    flow = cv2.calcOpticalFlowFarneback(prev_g, curr_g, None, 0.5, 3, 15, 3, 5,
1.2, 0)
    return flow  # HxWx2 (dx, dy)


def _warp(img: np.ndarray, flow: np.ndarray) -> np.ndarray:
    h, w = flow.shape[:2]
    grid_x, grid_y = np.meshgrid(np.arange(w), np.arange(h))
    map_x = (grid_x + flow[..., 0]).astype(np.float32)
    map_y = (grid_y + flow[..., 1]).astype(np.float32)
    return cv2.remap(img, map_x, map_y, interpolation=cv2.INTER_LINEAR,
borderMode=cv2.BORDER_REPLICATE)


def _to_lpips_tensor(bgr: np.ndarray) -> 'torch.Tensor':  # type: ignore
    import torch

    rgb = cv2.cvtColor(_ensure_uint8(bgr),
cv2.COLOR_BGR2RGB).astype(np.float32) / 255.0
    t = torch.from_numpy(rgb).permute(2, 0, 1).unsqueeze(0)  # 1x3xHxW
    t = t * 2 - 1  # [0,1] -> [-1,1]
    return t

# ---- Metrics ----

def masked_ssim_warped(prev: np.ndarray, curr: np.ndarray, mask: np.ndarray,
flow: Optional[np.ndarray] = None) -> float:
    """Compute SSIM on masked region after warping prev -> curr.
    mask: uint8 (0 or 255) same HxW; dilated mask recommended.
    """
    if flow is None:
        flow = _farneback_flow(prev, curr)
    prev_w = _warp(prev, flow)
    m = (mask > 0).astype(np.uint8)
    prev_g = _to_gray(prev_w)
    curr_g = _to_gray(curr)
    # Focus SSIM inside mask by zeroing outside and using gaussian_weights
    prev_roi = cv2.bitwise_and(prev_g, prev_g, mask=m)
    curr_roi = cv2.bitwise_and(curr_g, curr_g, mask=m)
    # Add small epsilon to avoid constant-black
    if np.count_nonzero(m) < 64:
        return 1.0
```

```python
    val = ssim(prev_roi, curr_roi, gaussian_weights=True,
use_sample_covariance=False)
    return float(val)


def masked_lpips(prev_clean: np.ndarray, curr_clean: np.ndarray, mask:
np.ndarray) -> float:
    import torch

    loss_fn = lpips.LPIPS(net='vgg')  # heavy but accurate
    with torch.no_grad():
        t0 = _to_lpips_tensor(prev_clean)
        t1 = _to_lpips_tensor(curr_clean)
        d = loss_fn(t0, t1)  # 1x1
        val = float(d.item())
    # Roughly weight by masked area proportion to emphasize region
    area = float(np.count_nonzero(mask)) / mask.size
    return val * max(0.25, min(1.0, area / 0.3))

@dataclass
class QCResult:
    warped_ssim: float
    lpips_val: Optional[float]


def qc_pair(prev_frame: np.ndarray, curr_frame: np.ndarray, mask: np.ndarray,
prev_clean: Optional[np.ndarray] = None) -> QCResult:
    ws = masked_ssim_warped(prev_frame, curr_frame, mask)
    lp = None
    if prev_clean is not None:
        lp = masked_lpips(prev_clean, curr_frame, mask)
    return QCResult(warped_ssim=ws, lpips_val=lp)

# ---- Tiny CLI to process a folder of frames ----

def _imread(p: Path) -> np.ndarray:
    img = cv2.imread(str(p), cv2.IMREAD_COLOR)
    if img is None:
        raise FileNotFoundError(p)
    return img


def main():
    import argparse

    ap = argparse.ArgumentParser(description="QC metrics for inpainted video
frames")
    ap.add_argument("--frames", type=Path, required=True, help="Folder with
```

```
cleaned frames (000001.png ...)")
    ap.add_argument("--orig", type=Path, required=True, help="Folder with
original frames for flow reference")
    ap.add_argument("--masks", type=Path, required=True, help="Folder with
masks")
    ap.add_argument("--out", type=Path, default=Path("out/qc.json"))
    ap.add_argument("--threshold", type=float, default=0.92)
    args = ap.parse_args()

    frames = sorted(args.frames.glob("*.png"))
    origs = sorted(args.orig.glob("*.png"))
    masks = sorted(args.masks.glob("*.png"))
    assert len(frames) == len(origs) == len(masks) and len(frames) > 1

    clean_count = 0
    vals = []
    for i in range(1, len(frames)):
        prev = _imread(origs[i-1])
        curr = _imread(origs[i])
        mask = _to_gray(_imread(masks[i]))
        curr_clean = _imread(frames[i])
        res = qc_pair(prev, curr, mask, prev_clean=_imread(frames[i-1]))
        ok = res.warped_ssim >= args.threshold
        clean_count += int(ok)
        vals.append({"i": i, "warped_ssim": res.warped_ssim, "ok": ok})

    report = {
        "total": len(frames),
        "clean_frames": clean_count,
        "threshold": args.threshold,
        "pass_rate": clean_count / len(frames),
        "per_frame": vals,
    }
    args.out.parent.mkdir(parents=True, exist_ok=True)
    with open(args.out, "w") as f:
        json.dump(report, f, indent=2)
    print(json.dumps(report, indent=2))

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

samples/synth_watermark.py

```
from __future__ import annotations
import math
from pathlib import Path
```

```
from typing import Tuple
import cv2
import numpy as np

# --- Simple synthetic generator ---
# Produces a 10s 1080p@30fps video with a moving background and an animated
bottom-right watermark.
# Also saves per-frame PNGs and a clean reference video.

H, W, FPS, SECS = 1080, 1920, 30, 10
N = FPS * SECS


def moving_background(t: int) -> np.ndarray:
    # scrolling gradient + noise texture
    x = (np.linspace(0, 1, W)[None, :] + (t / 200.0)) % 1.0
    y = (np.linspace(0, 1, H)[:, None] + (t / 300.0)) % 1.0
    base = (0.6 * x + 0.4 * y)
    tex = cv2.GaussianBlur(np.random.rand(H, W), (0, 0), 1.5)
    img = np.clip(0.6 * base + 0.4 * tex, 0, 1)
    img = (cv2.cvtColor((img * 255).astype(np.uint8), cv2.COLOR_GRAY2BGR))
    return img


def render_watermark(frame: np.ndarray, t: int) -> np.ndarray:
    # Pulsing alpha + slight scale wobble
    alpha = 0.25 + 0.15 * math.sin(2 * math.pi * t / 45.0)
    scale = 1.0 + 0.03 * math.sin(2 * math.pi * t / 60.0)
    text = "DEMO WATERMARK"
    overlay = frame.copy()
    # Text size
    font = cv2.FONT_HERSHEY_SIMPLEX
    font_scale = 1.5 * scale
    thickness = 3
    (tw, th), _ = cv2.getTextSize(text, font, font_scale, thickness)
    margin = 24
    x = W - tw - margin
    y = H - margin
    cv2.putText(overlay, text, (x, y), font, font_scale, (255, 255, 255),
thickness, cv2.LINE_AA)
    # Simple glow
    glow = cv2.GaussianBlur(overlay, (0, 0), 3)
    blended = cv2.addWeighted(frame, 1.0, glow, alpha, 0)
    return blended


def make_videos(outdir: Path):
    outdir.mkdir(parents=True, exist_ok=True)
```

```
    clean_dir = outdir / "clean_frames"
    mark_dir = outdir / "wm_frames"
    clean_dir.mkdir(exist_ok=True)
    mark_dir.mkdir(exist_ok=True)

    vw_clean = cv2.VideoWriter(str(outdir / "clean.mp4"),
cv2.VideoWriter_fourcc(*"mp4v"), FPS, (W, H))
    vw_mark = cv2.VideoWriter(str(outdir / "watermarked.mp4"),
cv2.VideoWriter_fourcc(*"mp4v"), FPS, (W, H))

    for t in range(N):
        f = moving_background(t)
        f_mark = render_watermark(f.copy(), t)
        vw_clean.write(f)
        vw_mark.write(f_mark)
        cv2.imwrite(str(clean_dir / f"{t:06d}.png"), f)
        cv2.imwrite(str(mark_dir / f"{t:06d}.png"), f_mark)

    vw_clean.release()
    vw_mark.release()
    print(f"Wrote {N} frames and two videos to {outdir}")

if __name__ == "__main__":
    make_videos(Path("samples/generated"))
```

Minimal CLI & Pipeline Stubs (runnable today)

> These run end-to-end with a simple **auto-mask + OpenCV Telea** inpaint so you can test the wiring while LaMa/SD and RAFT are being integrated. Methods `lama` / `sd` currently raise `NotImplementedError` but keep the CLI surface consistent.

`watermark_remover/cli/wmr.py`

```
from __future__ import annotations
import argparse
from pathlib import Path
import sys

from watermark_remover.core.pipeline import process_image, process_video


def _add_common_image_args(p: argparse.ArgumentParser):
    p.add_argument("input", type=Path)
    p.add_argument("--out", type=Path, required=True)
    p.add_argument("--mask", choices=["auto", "manual"], default="auto")
    p.add_argument("--dilate", type=int, default=5)
```

```
    p.add_argument("--method", choices=["telea", "lama", "sd", "noop"],
default="telea")
    p.add_argument("--seed", type=int, default=1234)


def _add_common_video_args(p: argparse.ArgumentParser):
    _add_common_image_args(p)
    p.add_argument("--window", type=int, default=48)
    p.add_argument("--overlap", type=int, default=12)
    p.add_argument("--temporal-guidance", default="none")
    p.add_argument("--wm-estimation", default="none")
    p.add_argument("--seam-blend", default="none")
    p.add_argument("--qc", default=None, help="e.g., warped_ssim>=0.92
(stubbed)")
    p.add_argument("--retry", type=int, default=0)


def main(argv=None):
    ap = argparse.ArgumentParser(prog="wmr", description="Watermark Remover
Suite (stub)")
    sp = ap.add_subparsers(dest="cmd", required=True)

    p_img = sp.add_parser("image", help="Process a single image")
    _add_common_image_args(p_img)

    p_vid = sp.add_parser("video", help="Process a video (stubbed sequential)")
    _add_common_video_args(p_vid)

    args = ap.parse_args(argv)

    if args.cmd == "image":
        process_image(args.input, args.out, method=args.method,
mask_mode=args.mask, dilate=args.dilate, seed=args.seed)
        return 0
    elif args.cmd == "video":
        process_video(args.input, args.out, method=args.method,
mask_mode=args.mask, dilate=args.dilate, seed=args.seed,
                      window=args.window, overlap=args.overlap)
        return 0
    else:
        ap.print_help()
        return 1


if __name__ == "__main__":
    sys.exit(main())
```

watermark_remover/core/pipeline.py

```python
from __future__ import annotations
from dataclasses import dataclass
from pathlib import Path
from typing import Literal

import cv2
import numpy as np
from tqdm import tqdm

Method = Literal["telea", "lama", "sd", "noop"]


def _imread(p: Path) -> np.ndarray:
    img = cv2.imread(str(p), cv2.IMREAD_COLOR)
    if img is None:
        raise FileNotFoundError(p)
    return img


def _imwrite(p: Path, img: np.ndarray) -> None:
    p.parent.mkdir(parents=True, exist_ok=True)
    cv2.imwrite(str(p), img)


def _auto_mask_bottom_right(img: np.ndarray, dilate: int = 5) -> np.ndarray:
    h, w = img.shape[:2]
    roi = img[int(h*0.75):, int(w*0.65):].copy()  # bottom-right quadrant-ish
    gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
    # Emphasize bright/semi-transparent text
    thr = max(180, int(gray.mean() + 0.6 * gray.std()))
    _, m = cv2.threshold(gray, thr, 255, cv2.THRESH_BINARY)
    # Edge boost (helps on thin text)
    edges = cv2.Canny(gray, 50, 150)
    m = cv2.bitwise_or(m, edges)
    # Place back into full-size mask
    mask = np.zeros((h, w), dtype=np.uint8)
    mask[int(h*0.75):, int(w*0.65):] = m
    if dilate > 0:
        k = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (dilate, dilate))
        mask = cv2.dilate(mask, k)
    return mask


def _inpaint(img: np.ndarray, mask: np.ndarray, method: Method, seed: int) ->
np.ndarray:
```

```python
    if method == "noop":
        return img
    if method == "telea":
        return cv2.inpaint(img, mask, 3, cv2.INPAINT_TELEA)
    if method == "lama":
        raise NotImplementedError("LaMa backend not wired yet — placeholder stub")
    if method == "sd":
        raise NotImplementedError("Stable Diffusion inpaint not wired yet — placeholder stub")
    raise ValueError(method)


def process_image(path_in: Path, path_out: Path, *, method: Method, mask_mode: str, dilate: int, seed: int) -> None:
    img = _imread(path_in)
    if mask_mode == "auto":
        mask = _auto_mask_bottom_right(img, dilate=dilate)
    else:
        # For now, manual not implemented; fall back to auto
        mask = _auto_mask_bottom_right(img, dilate=dilate)
    out = _inpaint(img, mask, method, seed)
    _imwrite(path_out, out)


def process_video(path_in: Path, path_out: Path, *, method: Method, mask_mode: str, dilate: int, seed: int, window: int, overlap: int) -> None:
    cap = cv2.VideoCapture(str(path_in))
    if not cap.isOpened():
        raise FileNotFoundError(path_in)
    w = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    h = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    fps = cap.get(cv2.CAP_PROP_FPS) or 30.0
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    out = cv2.VideoWriter(str(path_out), fourcc, fps, (w, h))

    pbar = tqdm(total=int(cap.get(cv2.CAP_PROP_FRAME_COUNT)) or None, desc="wmr-video")
    ok, frame = cap.read()
    prev_mask = None
    while ok:
        if mask_mode == "auto":
            mask = _auto_mask_bottom_right(frame, dilate=dilate)
        else:
            mask = _auto_mask_bottom_right(frame, dilate=dilate)
        cleaned = _inpaint(frame, mask, method, seed)
        out.write(cleaned)
        pbar.update(1)
```

```
        ok, frame = cap.read()
    pbar.close()
    cap.release()
    out.release()
```

`watermark_remover/models/download_models.py`

```
from __future__ import annotations
from pathlib import Path

# Placeholder downloader. For now it just creates the model cache tree and
prints guidance.

CACHE = Path.home() / ".wmr" / "models"

MODELS = {
    "raft": {
        "url": "<add public RAFT weights URL or instructions>",
        "sha256": "<fill>",
    },
    "lama": {
        "url": "<add LaMa weights URL or instructions>",
        "sha256": "<fill>",
    },
}


def main():
    CACHE.mkdir(parents=True, exist_ok=True)
    print(f"Model cache at {CACHE}")
    for name, meta in MODELS.items():
        print(f"- {name}: please download from {meta['url']} and place weights
here; set sha256 later")

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

LaMa Integration (Torch) — Drop-in Backend

This wires `--method lama` using a lightweight PyTorch LaMa checkpoint with simple tiling for 1080p+. It assumes you'll place weights at `~/.wmr/models/lama/` (see `download_models.py`).

watermark_remover/core/inpaint_lama.py

```
from __future__ import annotations
from dataclasses import dataclass
from pathlib import Path
from typing import Optional

import cv2
import numpy as np
import torch
import torch.nn.functional as F

# Minimal LaMa-like interface (expects a torchscript or standard torch model)

@dataclass
class LaMaConfig:
    device: str = "cuda"
    tile: int = 768       # process large images in tiles
    overlap: int = 64
    fp16: bool = True
    ckpt: Path = Path.home() / ".wmr" / "models" / "lama" / "lama.ckpt"


class LaMaInpainter:
    def __init__(self, cfg: Optional[LaMaConfig] = None):
        self.cfg = cfg or LaMaConfig()
        self.device = torch.device(self.cfg.device if torch.cuda.is_available()
else "cpu")
        self.model = self._load_model(self.cfg.ckpt, self.device)
        self.model.eval()
        if self.cfg.fp16 and self.device.type == "cuda":
            self.model.half()

    def _load_model(self, ckpt: Path, device: torch.device):
        if not ckpt.exists():
            raise FileNotFoundError(f"LaMa checkpoint not found: {ckpt}")
        # Expect a torchscript file for simplicity; swap with real LaMa loader
as needed
        model = torch.jit.load(str(ckpt), map_location=device)
        return model

    @torch.inference_mode()
    def inpaint(self, img_bgr: np.ndarray, mask: np.ndarray, dilation: int = 5,
seed: int = 1234) -> np.ndarray:
        # Normalize inputs
        img = img_bgr.copy()
        if dilation > 0:
```

```
            k = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (dilation,
dilation))
            mask = cv2.dilate(mask, k)
        mask = (mask > 0).astype(np.uint8)

        # Convert to tensor in range [-1,1]
        rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB).astype(np.float32) / 255.0
        t_img = torch.from_numpy(rgb).permute(2, 0, 1).unsqueeze(0)  # 1x3xH xW
        t_mask = torch.from_numpy(mask).unsqueeze(0).unsqueeze(0).float()
        if self.device.type == "cuda" and self.cfg.fp16:
            t_img = t_img.half()
        t_img = t_img.to(self.device)
        t_mask = t_mask.to(self.device)

        H, W = img.shape[:2]
        tile = self.cfg.tile
        if max(H, W) <= tile:
            out = self._run(t_img, t_mask)
        else:
            out = self._run_tiled(t_img, t_mask, tile=tile,
overlap=self.cfg.overlap)

        out = out.squeeze(0).permute(1, 2, 0).clamp(0, 1).float().cpu().numpy()
        out = (out * 255.0 + 0.5).astype(np.uint8)
        out = cv2.cvtColor(out, cv2.COLOR_RGB2BGR)
        return out

    def _run(self, img: torch.Tensor, mask: torch.Tensor) -> torch.Tensor:
        # Expected LaMa signature: model(img, mask) -> inpainted_img in [0,1]
        res = self.model(img, mask)
        return res

    def _run_tiled(self, img: torch.Tensor, mask: torch.Tensor, *, tile: int,
overlap: int) -> torch.Tensor:
        _, _, H, W = img.shape
        stride = tile - overlap
        acc = torch.zeros_like(img, dtype=torch.float16 if img.dtype ==
torch.float16 else torch.float32)
        weight = torch.zeros((1, 1, H, W), device=img.device,
dtype=torch.float32)

        for y in range(0, H, stride):
            for x in range(0, W, stride):
                y0, x0 = y, x
                y1, x1 = min(y + tile, H), min(x + tile, W)
                patch_img = img[:, :, y0:y1, x0:x1]
                patch_mask = mask[:, :, y0:y1, x0:x1]
                out = self._run(patch_img, patch_mask)
```

```
                acc[:, :, y0:y1, x0:x1] += out.to(acc.dtype)
                weight[:, :, y0:y1, x0:x1] += 1.0

        acc = acc / weight.clamp_min(1e-6)
        return acc
```

Wire LaMa into `pipeline.py`

```
# add near top
try:
    from watermark_remover.core.inpaint_lama import LaMaInpainter, LaMaConfig
    _LAMA_AVAILABLE = True
except Exception:
    _LAMA_AVAILABLE = False
    LaMaInpainter = None  # type: ignore

# inside _inpaint(...)
    if method == "lama":
        if not _LAMA_AVAILABLE:
            raise RuntimeError("LaMa not available. Ensure weights exist and
torch can load the model.")
        global _LAMA
        if "_LAMA" not in globals() or _LAMA is None:
            _LAMA = LaMaInpainter()
        return _LAMA.inpaint(img, mask, dilation=3, seed=seed)
```

Notes

- This expects a **TorchScript LaMa checkpoint** callable as `model(img, mask) -> out` in `[0,1]`. If you have the original LaMa repo weights, either export to TorchScript or adapt `_run()` to match their forward API.
- Tiling keeps VRAM steady on 1080p/1440p frames. Adjust `tile/overlap` based on your 3060's headroom.
- Once RAFT is wired, the same `LaMaInpainter` is used per frame; temporal guidance and seam blending remain in `temporal.py`.

LaMa Backend Wiring (first integration)

This adds a real LaMa path using **ONNX Runtime (GPU or CPU)** with simple tiling. Swap to a Torch checkpoint later if you prefer; the interface stays the same.

`watermark_remover/core/inpaint_lama.py`

```
from __future__ import annotations
from pathlib import Path
```

```
from typing import Tuple
import cv2
import numpy as np

try:
    import onnxruntime as ort  # pip install onnxruntime-gpu or onnxruntime
except Exception as e:  # pragma: no cover
    ort = None  # handled at runtime


class LaMaONNX:
    """Minimal LaMa ONNX runner with square-tiling for large images.
    Expects an ONNX that takes (image, mask) in NCHW float32 in [0,1].
    """

    def __init__(self, onnx_path: Path, device: str = "auto"):
        if ort is None:
            raise RuntimeError("onnxruntime not installed. pip install
onnxruntime-gpu or onnxruntime")
        providers = [
            ("CUDAExecutionProvider", {"arena_extend_strategy":
"kNextPowerOfTwo"}),
            "CPUExecutionProvider",
        ]
        if device == "cpu":
            providers = ["CPUExecutionProvider"]
        self.sess = ort.InferenceSession(str(onnx_path), providers=providers)
        self.input_names = [i.name for i in self.sess.get_inputs()]
        self.output_names = [o.name for o in self.sess.get_outputs()]
        # Try to infer expected tile size from model or default to 512
        self.tile = 512

    @staticmethod
    def _prep(img_bgr: np.ndarray, mask_u8: np.ndarray) -> Tuple[np.ndarray,
np.ndarray]:
        img = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB).astype(np.float32) /
255.0
        if mask_u8.ndim == 3:
            mask = cv2.cvtColor(mask_u8, cv2.COLOR_BGR2GRAY)
        else:
            mask = mask_u8
        mask = (mask > 0).astype(np.float32)
        return img, mask

    def _run(self, img_rgb: np.ndarray, mask: np.ndarray) -> np.ndarray:
        # NCHW
        inp_img = np.transpose(img_rgb, (2, 0, 1))[None, ...].astype(np.float32)
        inp_msk = mask[None, None, ...].astype(np.float32)
```

```
        feeds = {self.input_names[0]: inp_img, self.input_names[1]: inp_msk}
        out = self.sess.run(self.output_names, feeds)[0]
        out_rgb = np.transpose(out[0], (1, 2, 0))
        out_bgr = cv2.cvtColor(np.clip(out_rgb, 0, 1), cv2.COLOR_RGB2BGR)
        out_u8 = (out_bgr * 255.0 + 0.5).astype(np.uint8)
        return out_u8

    def inpaint(self, img_bgr: np.ndarray, mask_u8: np.ndarray) -> np.ndarray:
        h, w = img_bgr.shape[:2]
        if max(h, w) <= self.tile:
            img, m = self._prep(img_bgr, mask_u8)
            return self._run(img, m)
        # Tiled path with 32px overlap
        overlap = 32
        tile = self.tile
        out = np.zeros_like(img_bgr)
        weight = np.zeros((h, w, 1), dtype=np.float32)
        for y in range(0, h, tile - overlap):
            for x in range(0, w, tile - overlap):
                y0, y1 = y, min(y + tile, h)
                x0, x1 = x, min(x + tile, w)
                img = img_bgr[y0:y1, x0:x1]
                msk = mask_u8[y0:y1, x0:x1]
                patch = self.inpaint(img, msk) if max(img.shape[:2]) > tile else
self._run(*self._prep(img, msk))
                wy = np.linspace(0, 1, y1 - y0)[:, None]
                wx = np.linspace(0, 1, x1 - x0)[None, :]
                wpatch = (wy * wx)[..., None].astype(np.float32)
                out[y0:y1, x0:x1] = (out[y0:y1, x0:x1] * weight[y0:y1, x0:x1] +
patch * wpatch) / (
                    np.maximum(1e-6, weight[y0:y1, x0:x1] + wpatch)
                )
                weight[y0:y1, x0:x1] += wpatch
        return out


def inpaint_lama(img_bgr: np.ndarray, mask_u8: np.ndarray, model_path: Path,
device: str = "auto") -> np.ndarray:
    runner = LaMaONNX(model_path, device=device)
    return runner.inpaint(img_bgr, mask_u8)
```

`watermark_remover/models/README.md`

```
# Models

Place weights under `~/.wmr/models/` by default.
```

```
- **LaMa (ONNX)**: export or download a LaMa inpainting ONNX with 2 inputs:
`(image[NCHW float32 0..1], mask[N1HW float32])` and one output `image[NCHW
0..1]`.
- **RAFT (optional for flow)**: keep as `.pth` or TorchScript.

You can override the model path via CLI flags (to be added) or env vars later.
```

Pipeline hook (enable `--method lama` now)

Open `watermark_remover/core/pipeline.py` and modify `_inpaint`:

```
from pathlib import Path
from .inpaint_lama import inpaint_lama as lama_run

# ... inside _inpaint(...):
    if method == "lama":
        model = Path.home() / ".wmr" / "models" / "lama.onnx"
        if not model.exists():
            raise FileNotFoundError(f"LaMa ONNX not found at {model}. See
models/README.md")
        return lama_run(img, mask, model_path=model, device="auto")
```

> With this, `wmr image … --method lama` becomes functional as soon as you drop a `lama.onnx` in `~/.wmr/models/`.

RAFT Flow + Overlap Seams (video robustness)

> This adds a **flow module** with RAFT (best quality on 3060) and a **temporal module** for chunking + flow-aware seam blending. It runs even without RAFT via a Farneback fallback, so you can test immediately.

`watermark_remover/core/flow.py`

```
from __future__ import annotations
from dataclasses import dataclass
from pathlib import Path
from typing import Optional
import cv2
import numpy as np

@dataclass
class FlowConfig:
    backend: str = "auto"  # auto|raft|tvl1|farneback
```

```
    raft_weights: Optional[Path] = None  # ~/.wmr/models/raft.pth by default
    device: str = "cuda"

class FlowEstimator:
    def __init__(self, cfg: FlowConfig = FlowConfig()):
        self.cfg = cfg
        self._raft = None
        if cfg.backend in ("auto", "raft"):
            try:
                import torch
                from .raft_stub import RAFT
                w = cfg.raft_weights or (Path.home() / ".wmr" / "models" /
"raft.pth")
                if w.exists():
                    self._raft = RAFT.load_from_checkpoint(w, device=cfg.device)
                elif cfg.backend == "raft":
                    raise FileNotFoundError(f"RAFT weights not found at {w}")
            except Exception:
                self._raft = None

    @staticmethod
    def _to_gray_u8(img: np.ndarray) -> np.ndarray:
        if img.ndim == 2:
            g = img
        else:
            g = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        if g.dtype != np.uint8:
            g = np.clip(g, 0, 255).astype(np.uint8)
        return g

    def flow(self, prev: np.ndarray, curr: np.ndarray) -> np.ndarray:
        if self._raft is not None:
            return self._raft.flow(prev, curr)
        if self.cfg.backend in ("tvl1",):
            tvl1 = cv2.optflow.DualTVL1OpticalFlow_create()  # type: ignore
            return tvl1.calc(self._to_gray_u8(prev), self._to_gray_u8(curr),
None)
        # Farneback fallback
        return cv2.calcOpticalFlowFarneback(
            self._to_gray_u8(prev), self._to_gray_u8(curr), None, 0.5, 3, 15, 3,
5, 1.2, 0
        )

    @staticmethod
    def warp(img: np.ndarray, flow: np.ndarray) -> np.ndarray:
        h, w = flow.shape[:2]
        grid_x, grid_y = np.meshgrid(np.arange(w), np.arange(h))
        map_x = (grid_x + flow[..., 0]).astype(np.float32)
```

```
        map_y = (grid_y + flow[..., 1]).astype(np.float32)
        return cv2.remap(img, map_x, map_y, interpolation=cv2.INTER_LINEAR,
borderMode=cv2.BORDER_REPLICATE)
```

> Note: `raft_stub.RAFT` is a tiny wrapper you'll fill when you add actual RAFT weights/code (TorchScript or PyTorch). Until then, Farneback works.

`watermark_remover/core/temporal.py`

```
from __future__ import annotations
from typing import Iterator, List, Tuple
import numpy as np

from .flow import FlowEstimator


def make_chunks(n_frames: int, window: int, overlap: int) -> List[Tuple[int,
int]]:
    assert window > overlap >= 0
    chunks = []
    s = 0
    while s < n_frames:
        e = min(n_frames - 1, s + window - 1)
        chunks.append((s, e))
        if e == n_frames - 1:
            break
        s = e - overlap + 1
    return chunks


def blend_overlap(prev_clean: np.ndarray, curr_clean: np.ndarray,
flow_prev_to_curr: np.ndarray, alpha: float) -> np.ndarray:
    # alpha in [0,1]: weight of prev chunk
    from .flow import FlowEstimator
    warped_prev = FlowEstimator.warp(prev_clean, flow_prev_to_curr)
    return (warped_prev.astype(np.float32) * alpha +
curr_clean.astype(np.float32) * (1.0 - alpha)).astype(np.uint8)
```

Pipeline upgrade: chunking + seam blending

Open `watermark_remover/core/pipeline.py` and replace `process_video` with the version below. This keeps your existing `_auto_mask_bottom_right` and `_inpaint` functions.

```
from .temporal import make_chunks, blend_overlap
from .flow import FlowEstimator, FlowConfig
```

```
def process_video(path_in: Path, path_out: Path, *, method: Method, mask_mode: 
str, dilate: int, seed: int, window: int, overlap: int) -> None:
    cap = cv2.VideoCapture(str(path_in))
    if not cap.isOpened():
        raise FileNotFoundError(path_in)
    w = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    h = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    fps = cap.get(cv2.CAP_PROP_FPS) or 30.0
    total = int(cap.get(cv2.CAP_PROP_FRAME_COUNT)) or 0

    frames = []
    while True:
        ok, f = cap.read()
        if not ok:
            break
        frames.append(f)
    cap.release()

    if total == 0:
        total = len(frames)
    writer = cv2.VideoWriter(str(path_out), cv2.VideoWriter_fourcc(*"mp4v"),
fps, (w, h))

    # Precompute masks
    masks = [(_auto_mask_bottom_right(fr, dilate=dilate) if mask_mode == "auto"
else _auto_mask_bottom_right(fr, dilate=dilate)) for fr in frames]

    # Flow estimator (RAFT if available)
    flow_est = FlowEstimator()

    chunks = make_chunks(len(frames), window, overlap)
    prev_chunk_clean = None

    for ci, (s, e) in enumerate(tqdm(chunks, desc="wmr-video")):
        # Inpaint current chunk
        curr_clean = []
        for i in range(s, e + 1):
            cleaned = _inpaint(frames[i], masks[i], method, seed)
            curr_clean.append(cleaned)

        if ci == 0:
            # First chunk: write all except the trailing overlap (kept for 
blending)
            last_keep = e - overlap
            for fr in curr_clean[: max(0, last_keep - s + 1)]:
                writer.write(fr)
```

```
        else:
            # Blend the overlap with previous chunk
            # prev_chunk_clean holds the last 'overlap' frames of previous chunk
            for j in range(overlap):
                idx_prev = len(prev_chunk_clean) - overlap + j
                prev_fr = prev_chunk_clean[idx_prev]
                curr_fr = curr_clean[j]
                # Flow from original prev frame to curr frame (use original
frames for flow)
                flow = flow_est.flow(frames[s - overlap + j], frames[s + j])
                alpha = 1.0 - (j / max(1, overlap - 1))
                blended = blend_overlap(prev_fr, curr_fr, flow, alpha)
                writer.write(blended)
            # Write the middle of the chunk (non-overlap)
            mid_start = overlap
            mid_end = (e - s + 1) - overlap
            for fr in curr_clean[mid_start:mid_end]:
                writer.write(fr)

        # Keep trailing overlap of current chunk for next blend
        prev_chunk_clean = curr_clean

    # Flush tail: write the final chunk's trailing overlap as-is
    if prev_chunk_clean is not None:
        tail = prev_chunk_clean[-overlap:] if overlap > 0 else []
        for fr in tail:
            writer.write(fr)

    writer.release()
```

This is intentionally simple: it reads all frames, processes each chunk, **blends only at chunk seams** using flow, and writes the rest directly. You can later add warm-start guidance and QC retries.

RAFT stub wrapper (placeholder)

Create `watermark_remover/core/raft_stub.py` so `FlowEstimator` can import it now and you can replace internals later.

```
from __future__ import annotations
from dataclasses import dataclass
from pathlib import Path
import numpy as np

@dataclass
class RAFT:
```

```
    device: str = "cuda"

    @classmethod
    def load_from_checkpoint(cls, ckpt: Path, device: str = "cuda") -> "RAFT":
        # TODO: replace with real RAFT loading (torch), this is a placeholder
        return cls(device=device)

    def flow(self, prev: np.ndarray, curr: np.ndarray) -> np.ndarray:
        # TODO: call real RAFT model. For now, raise to trigger fallback in
FlowEstimator
        raise
RuntimeError("RAFT not implemented yet — install real RAFT or rely on fallback")
```

Usage (works today via fallback)

```
wmr video samples/generated/watermarked.mp4
  --out out/test_lama_temporal.mp4
  --method lama --window 48 --overlap 12
```

When you drop real RAFT weights + loader into `raft_stub.py`, the pipeline will automatically use it and produce smoother seams.