🧭 Unity Migration Plan – Image Processing Pipeline

1. 🔧 Project Setup

1.1 Install Prerequisites

Before starting the migration:

- **Unity Editor** (preferably latest LTS, e.g., 2022.3+)
- **Visual Studio 2022** or **JetBrains Rider** (C# IDE with Unity support)
- **.NET SDK** (v6 or v8 depending on Unity version)
- **Git** (for version control)
- **Python (optional)** for verifying original outputs during testing

Optional dependencies depending on features:

- **OpenCV for Unity** (Asset Store) — for masking, convex hulls, and geometric ops.
- **MediaPipe Unity Plugin** (from GitHub) — for body pose detection and landmark extraction.
- **ImageSharp or Magick.NET** — for fallback image encoding if MozJPEG unavailable.
- **MozJPEG binary** (`cjpeg.exe`) — placed in `Assets/StreamingAssets` for direct access.

1.2 Unity Project Structure

Create a new Unity 3D project and organize folders like this:

```
Assets/
 ├── Scripts/
 │     ├── Core/
 │     ├── Pipeline/
 │     ├── Utilities/
 │     ├── Tests/
 ├── Plugins/           → (for native DLLs like mozjpeg.dll)
 ├── StreamingAssets/   → (for external cjpeg.exe + configs)
 ├── InputImages/       → (for image data samples)
 ├── Output/            → (for processed results)
 ├── Resources/
```

This mirrors your Python folder structure: `input → temp → output`.

1.3 Unity Environment Configuration

- In **Project Settings → Player → API Compatibility Level**, choose **.NET 4.x** or **.NET 6**.
- In **Build Settings**, set target platform (Windows, macOS, or Linux).
- Ensure read/write permissions for `StreamingAssets` and `Output` folders.
- Add `Newtonsoft.Json` via Unity Package Manager for JSON parsing.

---

2. 🧩 Functional Migration Overview

You'll migrate functionality stage-by-stage, mirroring your Python pipeline. Each stage's purpose and Unity migration plan is detailed below.

2.1 Stage 0 – Configuration Loading

Python behavior: Reads `config.json` → loads parameters controlling toggles, file paths, and quality.

Unity migration plan:

- Create a `Config` class with equivalent fields.

- Load JSON using `JsonUtility` or `Newtonsoft.Json`.
- Use `StreamingAssets/config.json` for easy cross-platform access.
- Validate paths on load and create output directories automatically.

Testing:

- Print config values in the Unity console on startup.
- Confirm directory creation and flag toggles match Python behavior.

2.2 Stage 1 – Sample Creation

Python behavior: Randomly copies a subset of JPGs from input directory to a temporary folder (`TEMP_DIR`).

Unity migration plan:

- Use `System.IO.Directory` and `File.Copy` to select random samples.
- Implement optional sampling toggle (`TOGGLE_SAMPLE_CREATOR`).
- Store the copied images inside `/Temp/` in Unity project root or in `/Application.persistentDataPath/Temp/`.

Testing:

- Run with small datasets.
- Verify random selection and correct file counts in Unity console logs.

2.3 Stage 2 – Convex Hull Masking (Pose-Based)

Python behavior: Uses **MediaPipe Pose** to detect human landmarks → computes convex hull → masks background → optionally crops image.

Unity migration plan:

- Integrate **MediaPipe Unity Plugin**:
  - Use Pose Graph to get 2D body landmarks.
  - Retrieve landmark coordinates in image space.

- Compute convex hull using:
  - OpenCV for Unity's `CvInvoke.ConvexHull()` , or
  - Custom convex hull algorithm in C# (Graham scan or Quickhull).
- Create a binary mask (`Texture2D`) → fill hull region white → apply to original texture.
- Black out pixels outside the hull.
- Implement optional cropping (`TOGGLE_CONVEX_HULL_CROP`) by finding bounding box of non-zero mask area.

Testing:

- Load a few input images.
- Visually verify background masking and cropping.
- Compare side-by-side with Python output.

2.4 Stage 3 – Grayscale Conversion

Python behavior: Converts BGR to grayscale using OpenCV.

Unity migration plan:

- Modify pixel data via `Texture2D.GetPixels32()` .
- Compute luminance for each pixel → assign equal R, G, B values.
- Save as new `Texture2D` .

Testing:

- Display processed texture in a Unity UI Image component.
- Confirm grayscale visually and numerically by checking pixel RGB equality.

2.5 Stage 4 – Downscaling

Python behavior: Resizes images using OpenCV's `cv2.resize` with `INTER_AREA`.

Unity migration plan:

- Use `TextureScale` (available from Unity's docs) or `RenderTexture` / `Graphics.Blit()` for resizing.
- Apply `SCALE_FACTOR` from config.
- Maintain aspect ratio and validate output dimensions.

Testing:

- Log before/after dimensions.
- Compare with Python output resolutions.

2.6 Stage 5 – JPEG Saving (MozJPEG Integration)

Python behavior: Saves processed image using MozJPEG CLI (`cjpeg.exe`); falls back to Pillow.

Unity migration plan:

- Implement two-tier encoder strategy:

  Primary: Call MozJPEG binary using `System.Diagnostics.Process` (exactly as Python does).

  Fallback: Use `Texture2D.EncodeToJPG(quality)` or ImageSharp's encoder.

- Temporary PNG written to disk → passed to `cjpeg.exe` for conversion.
- Keep `MOZJPEG_PATH` configurable in `config.json`.

Testing:

- Compare output file sizes and visual quality against original Python results.
- Validate fallback path by temporarily removing `cjpeg.exe`.

2.7 Stage 6 – Batch Testing & Reporting

Python behavior: Runs multiple configurations, executes pipeline, measures compression, generates CSV, charts, and PDF reports.

Unity migration plan:

- Create an **Editor tool or MonoBehaviour** that loops over test configs.
- Measure folder sizes via `DirectoryInfo.GetFiles().Sum(f.Length)`.
- Write results to CSV in `/Output/Reports/`.
- Optional: Visualize results inside Unity UI (bar charts via UI Toolkit) or export CSV for Python/Excel visualization.

Testing:

- Run small test batches.
- Confirm file size calculations and CSV formatting.
- Optionally cross-check compression ratios with the Python `compression_report.csv`.

---

3. ⚙️ Cross-Feature Considerations

3.1 File Management

- Replace Python's relative paths with Unity's safe paths (`Application.dataPath`, `Application.persistentDataPath`).
- Ensure proper read/write permissions across OS builds.

3.2 Performance Optimization

- Use **Coroutines** or **Async Tasks** for long operations (prevent Editor freezes).
- Batch process images sequentially or in worker threads.
- Optionally leverage Unity's **Job System** or **Compute Shaders** for large-scale image sets.

3.3 Error Handling & Logging

- Wrap all file and encoding steps in `try/catch`.
- Mirror Python's "MozJPEG failed, fallback to Pillow" pattern.
- Log progress with Unity's `Debug.Log` and on-screen UI.

3.4 Debug & Visualization

- Create a small Unity Editor window to:
  - Display config toggles.
  - Load and preview sample images.
  - Run pipeline interactively for single images.

---

4. 🧠 Testing & Validation Plan

4.1 Phase 1 – Environment Validation

- Confirm Unity can read and write images.
- Load a test `config.json` and verify directory creation.

4.2 Phase 2 – Incremental Functional Tests

Test each module independently:

1. Sample creation → verify random file copy.
2. Grayscale → visually confirm.
3. Downscale → verify resolution.
4. Masking → check background removal visually.
5. Saving → compare JPEG file size and quality.

4.3 Phase 3 – Pipeline Integration

- Chain all stages end-to-end.

- Compare output folder content against Python pipeline output.
- Confirm runtime stability and correctness of compression logs.

4.4 Phase 4 – Performance Testing

- Measure total processing time across 20–100 images.
- Profile CPU/memory usage in Unity's Profiler.

4.5 Phase 5 – Regression Testing

- Save baseline results and rerun after any changes.
- Automate comparison (e.g., by pixel difference threshold).

---

5. 🚀 Deployment Strategy

- Package the pipeline as a Unity Editor Tool:
  - Custom menu: **Tools → Image Pipeline → Run Config**.
  - Drag-and-drop `config.json` to execute.
- Build standalone version if needed (e.g., command-line batch processor).
- Document usage similar to your current README (steps, toggles, config keys).

---

6. 📊 Migration Roadmap (Recommended Order)

Step	Component	Priority	Description
1	Unity environment setup	🔵 High	Create base project, import dependencies
2	Config loader	🔵 High	Foundation for all toggles
3	Grayscale conversion	🟢 Easy	Verify texture processing works
4	Downscale	🟢 Easy	Validate scaling and saving logic

Step	Component	Priority	Description
5	MozJPEG saving	🔵 High	Maintain compression parity
6	Sample creation	🟡 Medium	Add file management utilities
7	Convex hull masking	🔴 Hard	Integrate MediaPipe plugin, pose detection
8	Batch testing/reporting	🟡 Medium	Add CSV generation
9	Editor GUI	🟢 Optional	User interface for convenience
10	Optimization & polish	🔵 Final	Improve performance, clean logs

---

7. ✅ End Goal

After full migration, your Unity pipeline will:

- Read `config.json` with toggle control (same schema as Python).
- Process images (masking, grayscale, downscaling) with optional sampling.
- Save final results using **MozJPEG** for compression quality parity.
- Optionally batch test configurations and generate reports.
- Run fully within Unity (Editor or built application).

You'll maintain **feature parity** with the Python version while gaining real-time visualization, native image preview, and cross-platform portability.