VLM Reality Check — Antigravity Prompt Engineering Pipeline

Dataset Creation Only | Stages 1–5

> **How to use this document** Each stage is a deployable unit. Paste the prompt into Antigravity exactly as written. Do NOT proceed to the next stage until ALL test cases in the current stage pass. Treat each gate like a CI/CD merge check.

Pipeline Architecture

```
STAGE 1: Repo Scaffold & Config
         ↓ [GATE 1]
STAGE 2: Data Collection Pipeline
         ↓ [GATE 2]
STAGE 3: Vision Processing & Annotation
         ↓ [GATE 3]
STAGE 4: Adversarial Challenge Generator (9 Biases)
         ↓ [GATE 4]
STAGE 5: Multilingual Translation Pipeline
         ↓ [GATE 5]
         ✅ DATASET COMPLETE
         → data/challenges/challenges_multilingual.jsonl
         → 130K challenges × 5 languages = 650K instances
```

STAGE 1 — Repo Scaffold & Configuration

Prompt

```
You are a Staff ML Engineer at a top AI research lab.

Project: VLM Reality Check — a large-scale adversarial diagnostic benchmark
measuring 9 systematic biases in Vision-Language Models across 130K image-pair
challenges and 5 languages.

Task: Scaffold the complete project repository structure for this pipeline.
Follow production ML research conventions (similar to HuggingFace datasets repos).

Requirements:
- Monorepo layout with clear separation: data/, src/, configs/, scripts/, outputs/
- configs/biases.yaml: defines all 9 bias types with metadata:
    name, difficulty_split (easy/medium/hard summing to 1.0),
    ground_truth_method, target_challenge_count
  Biases: texture, counting, spatial_relations, physical_plausibility,
          temporal_reasoning, spurious_correlation, compositional_binding,
          text_in_image, scale_invariance
- configs/pipeline.yaml: global settings —
    target_dataset_size: 130000
```

```
    compound_challenge_count: 40000
    languages: [en, es, zh, hi, ar]
    random_seed: 42
    image_sources: [wikimedia, openimages, streetview, yfcc]
    difficulty_distribution: {easy: 0.30, medium: 0.50, hard: 0.20}
- configs/languages.yaml: EN, ES, ZH, HI, AR with Helsinki-NLP model IDs
- src/__init__.py with version = "0.1.0"
- Makefile with targets: collect, process, generate, translate, validate_dataset
- requirements.txt with pinned versions for:
    torch, transformers, ultralytics, opencv-python, astropy,
    pytesseract, Pillow, datasets, scipy, pandas, tqdm,
    requests, beautifulsoup4, imagehash
- .env.example with: STREET_VIEW_API_KEY, HF_TOKEN
- README.md with ASCII pipeline diagram and quickstart

Return the file tree first, then generate each config file in full.
Do not generate any src/ code yet — configs and scaffold only.
```

Rationale

Configs-first forces explicit decisions about taxonomy, scale, and target counts before any code is written. Errors here are cheap. Errors in Stage 4 are expensive.

GATE 1 — Test Cases

Test 1.1 — Bias taxonomy completeness

```python
import yaml

with open("configs/biases.yaml") as f:
    biases = yaml.safe_load(f)

required_biases = {
    'texture', 'counting', 'spatial_relations', 'physical_plausibility',
    'temporal_reasoning', 'spurious_correlation', 'compositional_binding',
    'text_in_image', 'scale_invariance'
}
assert set(biases.keys()) == required_biases, \
    f"Missing or extra biases: {set(biases.keys()) ^ required_biases}"

for name, meta in biases.items():
    splits = meta['difficulty_split']
    assert abs(sum(splits.values()) - 1.0) < 0.01, f"{name} splits don't sum to 1.0"
    assert 'ground_truth_method' in meta, f"{name} missing ground_truth_method"
    assert 'target_challenge_count' in meta, f"{name} missing target_challenge_count"

total = sum(b['target_challenge_count'] for b in biases.values())
assert 85000 <= total <= 95000, f"Single-bias targets should sum ~90K, got {total}"
print("GATE 1.1 PASSED")

PASS: Prints "GATE 1.1 PASSED"
FAIL: Any assert fails → fix the offending entry in configs/biases.yaml only
```

Test 1.2 — Pipeline config has all required keys

```python
import yaml

with open("configs/pipeline.yaml") as f:
    cfg = yaml.safe_load(f)

required_keys = {
    'target_dataset_size', 'compound_challenge_count', 'languages',
    'random_seed', 'image_sources', 'difficulty_distribution'
}
```

```
missing = required_keys - set(cfg.keys())
assert not missing, f"Missing pipeline config keys: {missing}"
assert cfg['target_dataset_size'] == 130000
assert cfg['compound_challenge_count'] == 40000
assert set(cfg['languages']) == {'en', 'es', 'zh', 'hi', 'ar'}
assert cfg['random_seed'] == 42
print("GATE 1.2 PASSED")

PASS: Prints "GATE 1.2 PASSED"
FAIL: Missing key → add to configs/pipeline.yaml
```

Test 1.3 — Makefile targets resolve

```
make -n collect && make -n process && make -n generate && \
make -n translate && make -n validate_dataset
echo "GATE 1.3 PASSED"

PASS: All targets resolve without "No rule to make target" error
FAIL: Missing target → add to Makefile
```

STAGE 2 — Data Collection Pipeline

Prompt

```
You are a Staff ML Engineer. Stage 1 scaffold is complete.

Task: Build src/collection/ — the image scraping and download pipeline
targeting 500K images from 4 sources.

Architecture: 4 source collectors, 1 unified pipeline with deduplication.

Implement:

src/collection/__init__.py

src/collection/base_collector.py
  - Abstract class BaseCollector
  - ImageRecord = dataclass(image_id, url, local_path, source, metadata: dict, license)
  - Abstract method: fetch_batch(n: int) -> List[ImageRecord]
  - Built-in retry logic: 3 retries, exponential backoff (1s, 2s, 4s)
  - Rate limiting: configurable requests_per_second (default 2)
  - Deduplication: perceptual hash (pHash) stored in SQLite at data/raw/seen_hashes.db
    Threshold: hamming distance < 10 = duplicate, skip it

src/collection/wikimedia_collector.py
  - Wikimedia Commons API — free, no key required
  - Filter: CC-licensed only, min resolution 512×512
  - Categories: ["outdoor scenes", "urban streets", "animals", "everyday objects"]
  - Returns ImageRecord with license field populated

src/collection/openimages_collector.py
  - Open Images v7 CSV manifest (assume at data/raw/yfcc_metadata.csv)
  - Filter: has_bbox=True, min 50 images per object class
  - Returns stratified sample across object categories

src/collection/streetview_collector.py
  - Google Street View Static API (key from .env STREET_VIEW_API_KEY)
  - Global city grid: 100 major cities, 50 GPS points per city
```

```
  - Per location: 4 headings (0°, 90°, 180°, 270°) for spatial pair challenges
  - Graceful fallback: if API key missing → log warning and skip this source entirely

src/collection/yfcc_collector.py
  - YFCC100M metadata CSV at data/raw/yfcc_metadata.csv
  - Filter: has_gps=True, is_outdoor=True, CC license, year < 2020
  - Metadata dict must include: timestamp, lat, lon, camera_model

src/collection/pipeline.py
  - CollectionPipeline class:
    - Runs all 4 collectors in parallel via ThreadPoolExecutor (max_workers=4)
    - Deduplicates across sources using pHash
    - Saves images to data/raw/images/{source}/{image_id}.jpg
    - Saves manifest to data/raw/manifest.jsonl
      Each line: {image_id, source, local_path, url, license, metadata}
    - Checkpoint-resume: reads already-completed image_ids from manifest.jsonl
      on startup, skips re-downloading those IDs
    - Progress: tqdm bar showing total collected, per-source counts, duplicates skipped
    - Target: 500K images total

Use type hints everywhere. Docstrings on all public methods.
```

Rationale

pHash deduplication prevents the same image appearing as both image A and image B in a challenge pair, which would corrupt ground truth. Checkpoint-resume is mandatory at 500K scale — collection will not finish in one uninterrupted session.

GATE 2 — Test Cases

Test 2.1 — ImageRecord schema is correct

```
from src.collection.base_collector import ImageRecord
import dataclasses

fields = {f.name for f in dataclasses.fields(ImageRecord)}
required = {'image_id', 'url', 'local_path', 'source', 'metadata', 'license'}
assert required.issubset(fields), f"Missing fields: {required - fields}"
print("GATE 2.1 PASSED")

PASS: Prints "GATE 2.1 PASSED"
FAIL: Missing field → add to ImageRecord dataclass in base_collector.py
```

Test 2.2 — pHash deduplication catches near-identical images

```
import imagehash
from PIL import Image
import numpy as np

# Exact duplicate
img = Image.fromarray(np.random.randint(0, 255, (512,512,3), dtype=np.uint8))
h1 = imagehash.phash(img)
h2 = imagehash.phash(img)
assert (h1 - h2) < 10, "Exact duplicate not caught by pHash threshold"

# Clearly different image
img2 = Image.fromarray(np.random.randint(0, 255, (512,512,3), dtype=np.uint8))
h3 = imagehash.phash(img2)
# Different images should typically have distance > 10
# (probabilistic but will pass with overwhelming probability for random images)
print("GATE 2.2 PASSED")

PASS: No assertion error
FAIL: imagehash not installed → add to requirements.txt and reinstall
```

Test 2.3 — Checkpoint-resume skips already-downloaded images

```
from src.collection.pipeline import CollectionPipeline
from unittest.mock import patch

pipeline = CollectionPipeline(config_path="configs/pipeline.yaml")

# Pre-write 3 image IDs to manifest as if already downloaded
import json
with open("data/raw/manifest.jsonl", "w") as f:
    for i in range(3):
        f.write(json.dumps({"image_id": f"img_{i:04d}", "source": "wikimedia",
                            "local_path": f"data/raw/images/img_{i:04d}.jpg",
                            "url": "http://example.com", "license": "CC", "metadata": {}}) + "\n")

with patch.object(pipeline, '_download_image', return_value=True) as mock_dl:
    pipeline.run(target_n=3, resume=True)
    assert mock_dl.call_count == 0, f"Expected 0 downloads, got {mock_dl.call_count}"

print("GATE 2.3 PASSED")

PASS: 0 redundant downloads triggered
FAIL: Re-downloading → fix checkpoint loading logic in pipeline.py __init__
```

STAGE 3 — Vision Processing & Annotation Pipeline

Prompt

```
You are a Staff ML Engineer and Computer Vision researcher.
Stage 2 is complete. Images are at data/raw/images/, manifest at data/raw/manifest.jsonl.

Task: Build src/processing/ — the automated annotation pipeline that extracts
structured metadata from each image. This metadata is the ground truth source
for all 9 bias challenge generators in Stage 4.

Critical: each processor must be independently importable and runnable.
Use result caching (SQLite) so re-runs don't reprocess existing images.

Implement:

src/processing/__init__.py

src/processing/base_processor.py
  - Abstract ProcessorBase:
    - Abstract method: process(image_path: str) -> dict
    - Cache layer: SQLite at data/processed/annotation_cache.db
      Key: image_id + processor_name. Returns cached dict if exists.
    - Error handling: on exception, return {"error": str(e), "skipped": True}
      Never crash the pipeline on a single bad image.

src/processing/object_detector.py   ← Used by: counting, spurious, scale
  - YOLOv8x model via ultralytics
  - Returns: {detections: List[{label, bbox, confidence, area_fraction}]}
  - Only include detections with confidence > 0.7
  - area_fraction = bbox_area / (image_width × image_height)
  - Batch inference: process images in batches of 32

src/processing/depth_estimator.py   ← Used by: spatial_relations, scale
```

```
  - MiDaS DPT-Large via transformers
  - Returns: {left_depth_mean, right_depth_mean, top_depth_mean, bottom_depth_mean}
    (average depth in each image quadrant)
  - Normalize depth map to [0, 1] before computing quadrant means

src/processing/shadow_detector.py   ← Used by: physical_plausibility
  - Two steps:
    Step A — Shadow detection:
      OpenCV HSV thresholding (S < 50, V < 80) + contour analysis
      Returns: shadow_angle_detected (degrees from vertical, None if no shadow found)
    Step B — Expected shadow angle:
      Astropy: given EXIF GPS (lat, lon) + EXIF DateTimeOriginal →
      compute sun azimuth and elevation via AltAz frame
      Shadow angle = sun_azimuth + 180° (shadow opposite sun)
      Returns: shadow_angle_expected (degrees), sun_elevation (degrees)
    Step C — Plausibility check:
      angle_delta = |shadow_angle_detected − shadow_angle_expected|
      is_physically_plausible = (angle_delta < 25°) if both angles available else None
  - Fallback: if EXIF GPS or timestamp missing → return all fields as None, skippable=True

src/processing/ocr_extractor.py     ← Used by: text_in_image
  - Tesseract via pytesseract
  - Returns: {
      has_text: bool,
      text_blocks: List[{text, bbox, confidence}],
      text_is_location_relevant: bool  ← True if any block matches street/road/place patterns
    }
  - Filter: only blocks with confidence > 60 and len(text.strip()) > 2

src/processing/texture_analyzer.py  ← Used by: texture_bias
  - Returns: {
      edge_density: float,              ← Canny edge pixel fraction
      dominant_texture_freq: float,     ← Peak Gabor filter response frequency
      silhouette_extractable: bool      ← True if edge_density > 0.05 and largest
                                           contour area > 10% of image
    }

src/processing/temporal_extractor.py ← Used by: temporal_reasoning
  - Parse EXIF DateTimeOriginal
  - Returns: {
      timestamp: str (ISO format, None if missing),
      time_of_day: str (dawn/morning/noon/afternoon/dusk/night),
      season: str (spring/summer/autumn/winter),
      construction_score: float  ← fraction of YOLO detections that are
                                   machinery/scaffolding/construction labels
    }

src/processing/annotation_pipeline.py
  - AnnotationPipeline: runs all 7 processors on each image, merges dicts
  - Output: data/processed/annotations.jsonl
    Each line: {image_id, detections, depth, shadow, ocr, texture, temporal}
  - Skip if image_id already annotated (cache check)
  - GPU batching for YOLO and MiDaS (batch_size=32)
  - tqdm progress bar with ETA
  - On completion: print summary — total processed, skipped (cached), errored
```

Rationale

The shadow detector (Astropy + OpenCV) is the key innovation that makes physical plausibility ground truth scientifically automated rather than manually labeled. The `skippable=True` fallback is critical — images without EXIF GPS must be cleanly excluded from physical_plausibility challenges, not cause pipeline crashes.

GATE 3 — Test Cases

Test 3.1 — Shadow plausibility is deterministically correct

```python
from src.processing.shadow_detector import ShadowDetector

detector = ShadowDetector()
# Known case: NYC (40.71°N, 74.01°W), summer solstice noon
# Sun is due south, elevation ~72° → expected shadow points north (~180° azimuth)
result = detector.compute_expected_shadow(
    lat=40.7128, lon=-74.0060,
    timestamp="2024-06-21T12:00:00"
)
assert result['shadow_angle_expected'] is not None
assert 160 < result['shadow_angle_expected'] < 200, \
    f"Expected shadow ~180° (north), got {result['shadow_angle_expected']}"
assert result['sun_elevation'] > 60, "Sun should be high at NYC noon in June"
print("GATE 3.1 PASSED")

PASS: Prints "GATE 3.1 PASSED"
FAIL: Wrong angle → check Astropy AltAz coordinate frame and timezone handling
```

Test 3.2 — Object detector returns correct schema

```python
from src.processing.object_detector import ObjectDetector
import os

detector = ObjectDetector()
# Use any real image from data/raw/images/ or a test fixture
test_img = "tests/fixtures/sample.jpg"  # place any outdoor JPEG here
result = detector.process(test_img)

assert 'detections' in result
assert isinstance(result['detections'], list)
for det in result['detections']:
    assert 'label' in det and 'confidence' in det and 'bbox' in det and 'area_fraction' in det
    assert det['confidence'] > 0.7, "Low-confidence detections should be filtered"
    assert 0 < det['area_fraction'] <= 1.0
print("GATE 3.2 PASSED")

PASS: Prints "GATE 3.2 PASSED"
FAIL: Schema mismatch → fix return dict structure in object_detector.py
```

Test 3.3 — Annotation pipeline output is complete and cacheable

```python
import json, os
from src.processing.annotation_pipeline import AnnotationPipeline

# Run on 5 fixture images
pipeline = AnnotationPipeline()
pipeline.run(image_ids=["test_001","test_002","test_003","test_004","test_005"],
             images_dir="tests/fixtures/")

required_keys = {'image_id', 'detections', 'depth', 'shadow', 'ocr', 'texture', 'temporal'}
with open("data/processed/annotations.jsonl") as f:
    records = [json.loads(l) for l in f]

assert len(records) == 5
for rec in records:
    missing = required_keys - set(rec.keys())
    assert not missing, f"Missing keys {missing} in record {rec.get('image_id')}"

# Run again — should use cache, not reprocess
import time
t0 = time.time()
pipeline.run(image_ids=["test_001","test_002","test_003","test_004","test_005"],
```

```
            images_dir="tests/fixtures/")
elapsed = time.time() - t0
assert elapsed < 2.0, f"Cache not working — reprocessing took {elapsed:.1f}s"
print("GATE 3.3 PASSED")

PASS: Prints "GATE 3.3 PASSED"
FAIL: Missing keys → add null fallback in annotation_pipeline.py merge step
      Cache not working → fix cache key lookup in base_processor.py
```

STAGE 4 — Adversarial Challenge Generator (9 Biases)

Prompt

```
You are a Staff ML Research Engineer.
Stages 1–3 complete. Annotations at data/processed/annotations.jsonl.

Task: Build src/generators/ — the adversarial challenge generation engine.
This is the scientific core of the benchmark.

CRITICAL CONSTRAINT: Each challenge must vary EXACTLY ONE factor between
Image A and Image B. Challenges that vary multiple factors are confounded
and scientifically invalid. Reject them — do not patch them.

Implement:

src/generators/base_generator.py
  - Challenge = dataclass(
        challenge_id: str,
        bias_type: str,
        difficulty: str,           # easy / medium / hard
        image_a_id: str,
        image_b_id: str,           # same as image_a_id for single-image transforms
        question_template: str,    # uses {variable} slots
        correct_answer: str,
        distractor_answers: List[str],
        ground_truth_method: str,  # e.g. "yolo_count", "astropy_shadow", "ocr_text"
        confound_check_passed: bool,
        metadata: dict
    )
  - Abstract ChallengeGenerator:
    - Abstract: generate_challenge(annotations: List[dict]) -> Challenge | None
    - validate_single_factor_isolation(challenge: Challenge) -> bool
      Checks: only the bias-relevant attribute differs between images.
      Returns False → generator must return None (rejected).

src/generators/texture_generator.py
  - Pairs: real photo of object vs OpenCV-generated silhouette of the SAME image
    (Canny edges + filled contour = silhouette, same object, texture removed)
  - Easy: high-edge-density object (dog, car) — shape is clear
  - Hard: low-edge-density, ambiguous shape (cat vs dog silhouette)
  - Question: "Do both images show the same type of object? Answer Yes or No."
  - Correct answer: always Yes (same object, texture removed)
  - Bias signal: model answers No when texture cues removed

src/generators/counting_generator.py
  - Pairs two images with YOLO-verified counts N vs M, same object category
  - Require |N − M| >= 2
  - Easy: |N − M| >= 4
```

```
  - Medium: |N − M| = 2 or 3
  - Hard: N=5, M=6 (both above subitizing limit of 4)
  - Question: "Which image has more {object_category}? Answer A or B."
  - Correct answer: deterministic from stored YOLO counts

src/generators/spatial_generator.py
  - Takes one image, creates adversarial version via:
    Horizontal flip → left/right inverted
    Vertical flip → above/below inverted
  - Question: "Are these the same scene from the same viewpoint? Answer Yes or No."
  - Correct answer: always No
  - Confound check: verify only spatial orientation changed (same pixel content, flipped)

src/generators/physics_generator.py
  - Source images: only those where shadow.is_physically_plausible = True
    and shadow.shadow_angle_detected is not None
  - Creates adversarial copy: rotate detected shadow region by 120−150°
    using OpenCV affine transform (shadow pixels only, not full image)
  - Easy: shadow points directly opposite correct direction (180° off)
  - Medium: shadow 60° off expected angle
  - Hard: shadow 30° off (subtly wrong, still plausible-looking)
  - Original image: Question "Is the lighting physically possible? Yes or No." → Yes
  - Modified image: same question → No
  - Confound check: only shadow angle changed, no other pixel regions modified

src/generators/temporal_generator.py
  - Groups 3 images by temporal sequence:
    Option A: dawn → noon → dusk (time_of_day progression, same scene type)
    Option B: construction_score low → medium → high (same GPS cluster)
  - Question: "Order these images chronologically. Answer as A,B,C or A,C,B etc."
  - Correct answer: the correct temporal ordering string
  - Hard: 3 images all within 2 hours (subtle lighting change)

src/generators/spurious_generator.py
  - Pairs image of entity in TYPICAL context vs ATYPICAL context
    Example: cow on grass (typical) vs cow on beach (atypical)
  - Same YOLO label, different background context (determined from scene metadata)
  - Question: "What is the main {category} in this image? Name it."
  - Correct answer: same object label for BOTH images
  - Bias signal: model gives different answers based on background

src/generators/compositional_generator.py
  - Selects images with 2+ YOLO-detected objects each having distinct dominant colors
    (computed via color histogram per bbox)
  - Generates correct description and 3 foil descriptions (swapped attribute bindings)
  - Question: "Which description correctly matches the image?"
  - Correct answer: the non-swapped description
  - Hard: objects with similar colors (dark blue vs dark green)

src/generators/text_image_generator.py
  - Pairs two images where ocr.text_is_location_relevant = True
  - OCR-extracted text differs between image A and image B
  - Question: "Are these the same location? Use only the visible text to decide.
               Answer Yes or No."
  - Correct answer: No (different text = different location)
  - Confound check: images must be visually similar except for text content

src/generators/scale_generator.py
  - Takes one image with a clearly detected object (area_fraction 0.1−0.4)
  - Creates zoomed version: crop to 1/zoom_factor of image centered on object bbox
    Zoom factor: randomly chosen from [5x, 8x, 10x, 15x]
  - Question: "Do these show the same type of object? Answer Yes or No." → Yes
  - Second question: "Are these objects the same real-world size? Answer Yes or No." → No
  - This split isolates recognition bias from scale bias
```

```
src/generators/compound_generator.py
  - Combines 2 single-bias generators sequentially on same image pair
  - Supported combinations (must be visually coherent):
    spatial + physics, counting + spurious, texture + scale, compositional + text
  - Records: bias_types_combined: ["spatial_relations", "physical_plausibility"]
  - difficulty: always "hard"
  - metadata: includes expected_failure_mode per bias component

src/generators/pipeline.py
  - GenerationPipeline:
    - Loads annotations.jsonl
    - Runs each generator to hit per-bias target counts from configs/biases.yaml
    - Compound generator runs last (depends on single-bias images being validated)
    - Total target: 90K single-bias + 40K compound = 130K challenges
    - Saves to data/challenges/challenges.jsonl
    - Rejects any challenge where confound_check_passed = False (log rejection)
    - On completion prints: generated per bias, rejected per bias, total
```

Rationale

The `confound_check_passed` rejection (not patching) is the scientific boundary condition of this benchmark. The compound generator must run last because compound challenges combine already-validated single-bias image pairs — using unvalidated pairs would silently contaminate the compound set. Scale invariance is split into two sub-questions deliberately — this is the only way to separate object recognition from size estimation.

GATE 4 — Test Cases

Test 4.1 — Confounded challenge is rejected, not accepted

```python
from src.generators.spatial_generator import SpatialGenerator

gen = SpatialGenerator()
# Mock annotation pair where BOTH spatial orientation AND object count differ
# (would be confounded - testing two things at once)
confounded_annotations = [
    {"image_id": "img_a", "detections": {"detections": [{"label":"cat"}]*3},
     "depth": {"left_depth_mean": 0.3}},
    {"image_id": "img_b", "detections": {"detections": [{"label":"cat"}]*7},  # count also differs
     "depth": {"left_depth_mean": 0.7}},
]
# Patch generate to use these annotations, then check confound check fails
result = gen.generate_challenge(confounded_annotations)
if result is not None:
    assert result.confound_check_passed == False, \
        "Confounded challenge must have confound_check_passed=False"
print("GATE 4.1 PASSED")

PASS: result is None OR result.confound_check_passed == False
FAIL: Confounded challenge accepted as valid → fix validate_single_factor_isolation()
```

Test 4.2 — Counting generator assigns difficulty and answer correctly

```python
from src.generators.counting_generator import CountingGenerator

gen = CountingGenerator()
# 3 cats vs 7 cats → easy (|diff|=4), answer=B
ann_3 = {"image_id":"img_a", "detections":{"detections":[{"label":"cat","confidence":0.9}]*3}}
ann_7 = {"image_id":"img_b", "detections":{"detections":[{"label":"cat","confidence":0.9}]*7}}
challenge = gen.generate_challenge([ann_3, ann_7])
assert challenge is not None
assert challenge.correct_answer == "B", "B has more cats"
assert challenge.difficulty == "easy", "|3-7|=4 should be easy"

# 5 cats vs 6 cats → hard (both >4, subitizing zone)
```

```
ann_5 = {"image_id":"img_c", "detections":{"detections":[{"label":"cat","confidence":0.9}]*5}}
ann_6 = {"image_id":"img_d", "detections":{"detections":[{"label":"cat","confidence":0.9}]*6}}
challenge_hard = gen.generate_challenge([ann_5, ann_6])
assert challenge_hard.difficulty == "hard", "|5-6|=1 and both >4 should be hard"
print("GATE 4.2 PASSED")

PASS: Prints "GATE 4.2 PASSED"
FAIL: Wrong answer or difficulty → fix ground truth and difficulty assignment logic
```

Test 4.3 — Final dataset distribution matches target

```
import json
from collections import Counter

challenges = [json.loads(l) for l in open("data/challenges/challenges.jsonl")]
total = len(challenges)
assert total >= 125_000, f"Only {total} challenges generated (target 130K)"

bias_counts = Counter(c['bias_type'] for c in challenges)
difficulty_counts = Counter(c['difficulty'] for c in challenges)

# No single bias > 15% of total
for bias, count in bias_counts.items():
    frac = count / total
    assert frac <= 0.15, f"{bias} is {frac:.1%} of dataset — too dominant"

# Difficulty distribution within ±5% of target
easy_frac   = difficulty_counts['easy']   / total
medium_frac = difficulty_counts['medium'] / total
hard_frac   = difficulty_counts['hard']   / total
assert 0.25 <= easy_frac   <= 0.35, f"Easy {easy_frac:.2%} outside [25%, 35%]"
assert 0.45 <= medium_frac <= 0.55, f"Medium {medium_frac:.2%} outside [45%, 55%]"
assert 0.15 <= hard_frac   <= 0.25, f"Hard {hard_frac:.2%} outside [15%, 25%]"

# Confirm compound challenges exist
compound = [c for c in challenges if c['bias_type'] == 'compound']
assert len(compound) >= 35_000, f"Only {len(compound)} compound challenges (target 40K)"
print("GATE 4.3 PASSED")

PASS: Prints "GATE 4.3 PASSED"
FAIL: Distribution off → adjust target_challenge_count in configs/biases.yaml
      and re-run generation pipeline
```

STAGE 5 — Multilingual Translation Pipeline

Prompt

```
You are a Staff NLP Engineer.
Stage 4 complete. Challenges at data/challenges/challenges.jsonl.

Task: Build src/translation/ — translate all 130K challenge questions into
4 additional languages (ES, ZH, HI, AR) using Helsinki-NLP models locally.

Critical design: template-first translation, NOT sentence-by-sentence.
Translate the question TEMPLATE (with {variable} slots preserved), then
slot-fill variables after. This prevents translation drift and variable corruption.

Implement:
```

```
src/translation/__init__.py

src/translation/question_templates.py
  - Define 3-5 question templates PER BIAS TYPE in English
  - Variables in {} are preserved through translation unchanged
  - Example for counting:
      TEMPLATES['counting'] = [
        "Which image has more {object_category}? Answer A or B.",
        "Image A shows {count_a} {object_category}. Image B shows {count_b}.
         Which has more? Answer A or B.",
      ]
  - Templates for all 9 biases + compound. Total: ~50 templates.
  - Each template must have: template_id, bias_type, variables: List[str],
    valid_answers: List[str]

src/translation/helsinki_translator.py
  - Loads Helsinki-NLP/opus-mt-{src}-{tgt} models from HuggingFace:
      EN→ES: Helsinki-NLP/opus-mt-en-es
      EN→ZH: Helsinki-NLP/opus-mt-en-zh
      EN→HI: Helsinki-NLP/opus-mt-en-hi
      EN→AR: Helsinki-NLP/opus-mt-en-ar
  - Template translation method:
      1. Replace {variable_name} with [SLOT_0], [SLOT_1], etc.
      2. Translate the de-slotted template string
      3. Restore [SLOT_N] back to {variable_name}
  - Batch size: 32 templates per forward pass
  - Cache: if (template_id, target_lang) already in SQLite cache → return cached
  - Returns: translated_template string with {} slots intact

src/translation/script_validator.py
  - Validates translated text has correct script characters:
    ES: basic Latin - assert no [SLOT] tokens remain
    ZH: assert contains CJK unicode range (U+4E00-U+9FFF)
    HI: assert contains Devanagari range (U+0900-U+097F)
    AR: assert contains Arabic range (U+0600-U+06FF)
  - For AR (RTL): prepend Unicode RLM marker \u200F for correct rendering
  - For ZH: strip extra spaces between characters

src/translation/translation_pipeline.py
  - Loads challenges.jsonl
  - For each challenge:
      1. Select appropriate template by bias_type
      2. Translate template for all 4 non-English languages (cached)
      3. Slot-fill {variables} from challenge metadata
      4. Validate script per language via script_validator
  - Output: data/challenges/challenges_multilingual.jsonl
    Each record is the original challenge dict + new field:
    "questions": {"en": "...", "es": "...", "zh": "...", "hi": "...", "ar": "..."}
  - Log: cache hit rate, per-language template count, any failed validations

src/translation/human_validation_sampler.py
  - Randomly samples (seed=42) 100 challenges per language = 500 total
  - Output: data/validation/human_validation_sample.csv
    Columns: challenge_id, bias_type, language, en_question, translated_question,
             is_fluent, is_accurate, notes
    (is_fluent / is_accurate / notes left blank for human annotators to fill)
```

Rationale

Template-first translation is mandatory here. Free-form translation of complete sentences like "Which image has more cats?" will sometimes translate "cats" to a localized equivalent, breaking the variable slot that should have been "{object_category}". The slot-masking approach guarantees object names, answer choices (A/B), and numbers survive translation intact. Arabic RTL requires the unicode RLM marker or the question renders garbled in most interfaces.

GATE 5 — Test Cases

Test 5.1 — Variable slots survive all 4 translations

```
from src.translation.helsinki_translator import HelsinkiTranslator

translator = HelsinkiTranslator()
template = "Which image has more {object_category}? Answer A or B."

for lang in ['es', 'zh', 'hi', 'ar']:
    result = translator.translate_template(template, target_lang=lang)
    assert '{object_category}' in result, \
        f"Slot '{'{object_category}'}' lost in {lang} translation: '{result}'"
    assert '[SLOT' not in result, \
        f"SLOT token not restored in {lang}: '{result}'"

print("GATE 5.1 PASSED")

PASS: All 4 languages preserve {object_category} and restore SLOT tokens
FAIL: Slot eaten → fix masking/restoring logic in helsinki_translator.py
```

Test 5.2 — Script validation catches wrong-script outputs

```
from src.translation.script_validator import ScriptValidator

validator = ScriptValidator()

# Correct scripts should pass
assert validator.validate("Hola mundo", lang="es") == True
assert validator.validate("你好世界", lang="zh") == True
assert validator.validate("नमस्ते दुनिया", lang="hi") == True
assert validator.validate("مرحبا بالعالم", lang="ar") == True

# Wrong script should fail
assert validator.validate("Hello world", lang="zh") == False, \
    "English text should fail Chinese validation"
assert validator.validate("Hello world", lang="ar") == False, \
    "English text should fail Arabic validation"

print("GATE 5.2 PASSED")

PASS: Prints "GATE 5.2 PASSED"
FAIL: Validation too loose → tighten unicode range checks in script_validator.py
```

Test 5.3 — Final multilingual output is complete

```
import json

records = [json.loads(l) for l in open("data/challenges/challenges_multilingual.jsonl")]
langs = ['en', 'es', 'zh', 'hi', 'ar']

issues = []
for rec in records[:2000]:  # spot-check first 2K
    for lang in langs:
        q = rec.get('questions', {}).get(lang, '')
        if not q or len(q.strip()) < 5:
            issues.append(f"challenge {rec['challenge_id']} lang {lang} is empty/short")

assert len(issues) == 0, f"{len(issues)} translation issues:\n" + "\n".join(issues[:5])

# Verify total record count
assert len(records) >= 125_000, f"Only {len(records)} records (expected 130K)"
print(f"GATE 5.3 PASSED — {len(records):,} multilingual challenges generated")
```

```
PASS: Prints "GATE 5.3 PASSED — X multilingual challenges generated"
FAIL: Empty translations → check which bias_type × language combination is failing
      and verify template coverage in question_templates.py
```

✅ Dataset Complete — Final Smoke Test

Run this after all 5 gates pass

```python
"""
Dataset completeness smoke test.
Verifies the final deliverable before handing off to evaluation.
"""
import json
from collections import Counter
from pathlib import Path

print("Running final dataset smoke test...")

# 1. Check all files exist
assert Path("data/raw/manifest.jsonl").exists(), "manifest.jsonl missing"
assert Path("data/processed/annotations.jsonl").exists(), "annotations.jsonl missing"
assert Path("data/challenges/challenges.jsonl").exists(), "challenges.jsonl missing"
assert Path("data/challenges/challenges_multilingual.jsonl").exists(), "multilingual missing"
assert Path("data/validation/human_validation_sample.csv").exists(), "validation sample missing"

# 2. Check image count
manifest = [json.loads(l) for l in open("data/raw/manifest.jsonl")]
assert len(manifest) >= 400_000, f"Only {len(manifest):,} images collected (target 500K)"

# 3. Check challenge count and structure
challenges = [json.loads(l) for l in open("data/challenges/challenges_multilingual.jsonl")]
assert len(challenges) >= 125_000, f"Only {len(challenges):,} challenges (target 130K)"

required_challenge_keys = {
    'challenge_id', 'bias_type', 'difficulty',
    'image_a_id', 'image_b_id', 'correct_answer',
    'ground_truth_method', 'confound_check_passed', 'questions'
}
for c in challenges[:100]:  # spot-check
    missing = required_challenge_keys - set(c.keys())
    assert not missing, f"Challenge {c['challenge_id']} missing keys: {missing}"

# 4. Check language coverage
for c in challenges[:100]:
    for lang in ['en', 'es', 'zh', 'hi', 'ar']:
        q = c['questions'].get(lang, '')
        assert q and len(q) > 4, f"Missing {lang} in challenge {c['challenge_id']}"

# 5. Check no confounded challenges slipped through
confounded = [c for c in challenges if not c.get('confound_check_passed', True)]
assert len(confounded) == 0, f"{len(confounded)} confounded challenges in dataset!"

# 6. Summary
bias_dist = Counter(c['bias_type'] for c in challenges)
diff_dist  = Counter(c['difficulty'] for c in challenges)
print("\n" + "="*60)
print("DATASET SMOKE TEST PASSED")
print("="*60)
```

```python
print(f"Images collected:      {len(manifest):>10,}")
print(f"Total challenges:      {len(challenges):>10,}")
print(f"Languages per challenge: 5 (EN, ES, ZH, HI, AR)")
print(f"Total question instances: {len(challenges)*5:>8,}")
print(f"\nBias distribution:")
for bias, count in sorted(bias_dist.items()):
    print(f"  {bias:<28} {count:>7,}  ({count/len(challenges):.1%})")
print(f"\nDifficulty distribution:")
for diff, count in sorted(diff_dist.items()):
    print(f"  {diff:<10} {count:>7,}  ({count/len(challenges):.1%})")
print("\nDataset ready for evaluation pipeline.")
```

Summary

Stage	Deliverable	Key File
1	Repo + configs	`configs/biases.yaml` , `configs/pipeline.yaml`
2	500K images	`data/raw/manifest.jsonl`
3	Vision annotations	`data/processed/annotations.jsonl`
4	130K challenges	`data/challenges/challenges.jsonl`
5	650K multilingual instances	`data/challenges/challenges_multilingual.jsonl`

5 stages · 15 test gates · 1 integration test