

Ford BnP QC System

Technical Documentation

Build & Price Asset Pack Quality Control System

Generated: February 20, 2026

Confidential — For Internal Use Only

Table of Contents

System Overview . . . 7

- What Is This Tool? . . . 7
- Modes of Operation . . . 7
- High-Level Pipeline . . . 7
- Component Architecture . . . 7
- Directory Structure . . . 7
- Technology Stack . . . 8

Data Model & Asset Pack Structure . . . 9

- Asset Pack Overview . . . 9
- linkingrecord.json Schema . . . 9
 - Top-Level Fields . . . 9
 - Header Fields . . . 10
 - Item Fields . . . 10
 - Record Fields . . . 10
 - Asset Fields . . . 10
- ViewType / ImageType Matrix . . . 10
- Pack Types: MEC vs BAU . . . 11
 - BAU (Business As Usual) . . . 11
 - MEC (Multi-Experience Configurator) . . . 11
- Allowed File Types . . . 12
- Asset Naming Conventions . . . 12
 - General Pattern . . . 12
 - Beltline Files . . . 12
 - MEC-Specific Files . . . 12
 - Layer Numbering . . . 12
- Feature Codes . . . 13

Architecture & Data Flow . . . 14

- Module Relationships . . . 14
- CLI Workflow . . . 14
 - CLI Usage . . . 14
 - CLI Output . . . 15
- Box Hotfolder Workflow . . . 15
 - Box Folder Structure . . . 15

| | |
|---|---------------|
| File Stamping . . . | 16 |
| Profile System . . . | 16 |
| How It Works . . . | 16 |
| __WORKING_DIR__ Substitution . . . | 16 |
| Dynamic Module Loading . . . | 16 |
| Configuration System . . . | 16 |
| utils/config.py — Singleton Pattern . . . | 16 |
| Loading Precedence . . . | 17 |
| Required Variables . . . | 17 |
| Optional Variables . . . | 17 |
| Error Handling Architecture . . . | 18 |
| Layer 1: qc_engine.py (CLI) . . . | 18 |
| Layer 2: qc_module.py (Library) . . . | 18 |
| Layer 3: ford_qc_box_hotfolder_process.py (Service) . . . | 18 |
| Retry & Resilience . . . | 18 |
| Exponential Backoff . . . | 18 |
| Graceful Shutdown . . . | 19 |
| Systemd Watchdog . . . | 19 |
| Service Restart . . . | 19 |
| QC Checks Reference . . . | 20 |
| 1. Archive Extraction & Verification . . . | 20 |
| What It Validates . . . | 20 |
| Header Validation Rules . . . | 20 |
| File Type Rules . . . | 20 |
| Pass/Fail . . . | 21 |
| 2. Color Chips Verification . . . | 21 |
| What It Validates . . . | 21 |
| Pass/Fail . . . | 21 |
| 3. Missing Images Check . . . | 21 |
| What It Validates . . . | 21 |
| Pass/Fail . . . | 21 |
| Output Details . . . | 21 |
| 4. MEC/BAU Special Requirements . . . | 22 |
| Pack Type Detection . . . | 22 |
| MEC Pack Validation . . . | 22 |
| BAU Pack Validation . . . | 22 |
| Pass/Fail . . . | 22 |
| 5. MEC Powertrain Validation . . . | 22 |
| What It Validates (MEC Only) . . . | 22 |
| Pass/Fail . . . | 23 |

| | |
|---|----|
| 6. Lifestyle & Inventory Validation . . . | 23 |
| What It Validates . . . | 23 |
| Pass/Fail . . . | 23 |
| 7. Image Resolution Check . . . | 23 |
| BAU Resolution Requirements . . . | 23 |
| MEC Resolution Differences . . . | 24 |
| MEC Override Logic . . . | 24 |
| Pass/Fail . . . | 24 |
| 8. Image Format Validation . . . | 25 |
| Format Requirements . . . | 25 |
| Base Asset Special Rules . . . | 25 |
| AVIF Validation . . . | 25 |
| Pass/Fail . . . | 25 |
| 9. File Size Validation . . . | 26 |
| Size Limits . . . | 26 |
| Three-Tier Validation . . . | 26 |
| Pass/Fail . . . | 26 |
| 10. Image Linking Check . . . | 27 |
| What It Validates . . . | 27 |
| Pass/Fail . . . | 27 |
| 11. Layer Depth Validation . . . | 27 |
| What It Validates . . . | 27 |
| Filename Layer Extraction . . . | 27 |
| Pass/Fail . . . | 27 |
| 12. Series Permutations Check . . . | 28 |
| What It Validates . . . | 28 |
| Feature Code Extraction . . . | 28 |
| Pass/Fail . . . | 28 |
| 13. Beltline Validation . . . | 28 |
| What It Validates . . . | 28 |
| Additional Validations . . . | 28 |
| Pass/Fail . . . | 29 |
| 14. Extra Carousel Validation . . . | 29 |
| What It Validates . . . | 29 |
| Coverage Thresholds . . . | 29 |
| Pass/Fail . . . | 29 |
| Check Execution Order . . . | 29 |
| Skipped Types Handling . . . | 30 |

Reports & Output . . . 31

| | |
|-------------------------------|----|
| Report Types . . . | 31 |
| JSON Results Structure . . . | 31 |
| Top-Level Schema . . . | 31 |
| Per-Check Result Schema . . . | 31 |
| HTML QC Report . . . | 32 |
| Generation . . . | 32 |
| Output Filename . . . | 32 |
| Report Layout . . . | 32 |
| Check Name Display . . . | 33 |
| Detail Rendering . . . | 34 |
| HTML Error Report . . . | 34 |
| When It's Generated . . . | 34 |
| Generation . . . | 34 |
| Output Filename . . . | 35 |
| Error Types . . . | 35 |
| Error Report Layout . . . | 35 |
| Fallback . . . | 35 |
| Report Destinations . . . | 35 |

Deployment & Operations . . . 37

| | |
|--|----|
| Local Development Setup . . . | 37 |
| Prerequisites . . . | 37 |
| Installation . . . | 37 |
| Running QC Locally (CLI) . . . | 37 |
| Testing Individual Checks . . . | 37 |
| Creating a Test Profile . . . | 37 |
| Environment Configuration . . . | 38 |
| Configuration Files . . . | 38 |
| Setting Up a New Environment . . . | 38 |
| Key Variables to Configure . . . | 38 |
| Box JWT Configuration . . . | 38 |
| Setting Up Box JWT . . . | 38 |
| Production Deployment . . . | 39 |
| Service Architecture . . . | 39 |
| Installing the Service . . . | 39 |
| Service File Breakdown . . . | 39 |
| Service Management . . . | 40 |
| Common Commands . . . | 40 |
| Viewing Logs . . . | 40 |
| Running Manually (for debugging) . . . | 40 |

Multi-Environment Operation . . . 41

Troubleshooting . . . 41

Service Won't Start . . . 41

Box Authentication Failures . . . 41

QC Checks Failing Unexpectedly . . . 41

Service Stuck / Not Processing . . . 41

Developer Guide: Extending the System . . . 42

The Check Module Contract . . . 42

Required Interface . . . 42

Status Values . . . 42

Error Result Format . . . 42

Adding a New Check: Step by Step . . . 43

1. Create the Module . . . 43

2. Add to the QC Profile . . . 43

3. Test the Check . . . 43

4. Add HTML Report Rendering (if needed) . . . 44

Using Check Helpers . . . 44

Recording Skipped Types . . . 44

Common Patterns . . . 44

Pack Type Detection . . . 44

Deduplication by Filename . . . 45

Iterating the Linking Record . . . 45

Writing Tests . . . 45

Test Structure . . . 45

Testing Tips . . . 46

Common Pitfalls . . . 46

File Path Handling . . . 46

AVIF Case Sensitivity . . . 47

Check Ordering . . . 47

Error Handling . . . 47

The HTML Reporter . . . 47

System Overview

What Is This Tool?

The Ford BnP (Build & Price) QC system is an automated quality control pipeline for Ford vehicle configurator asset packs. These asset packs are ZIP files containing:

- A `linkingrecord.json` manifest that describes every image in the pack, its type, viewing angle, features, and layer information
- **Hundreds of images** (JPEG, PNG, AVIF) — exterior/interior renders, color chips, carousel thumbnails, beltline animations, and more

The QC system extracts the ZIP, parses the manifest, and runs **14 sequential checks** that validate everything from image resolutions and file formats to business logic like MEC/BAU compliance and series permutation coverage. It produces an HTML report summarizing all findings.

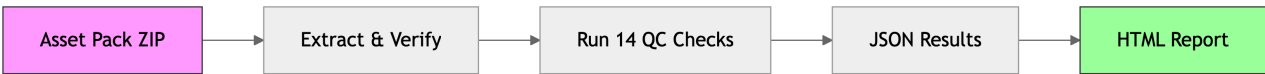
Modes of Operation

The system runs in two distinct modes:

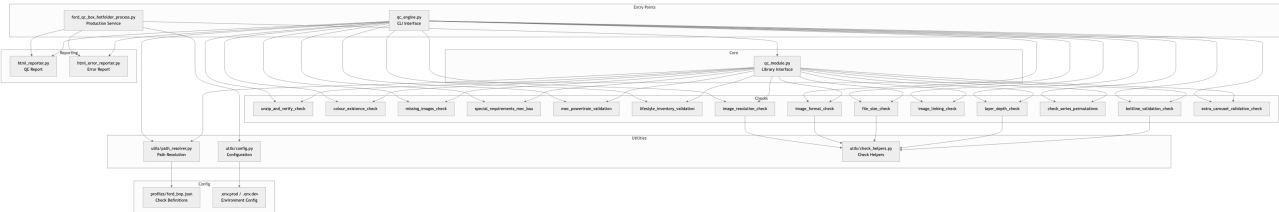
| Mode | Entry Point | Use Case |
|---------------|---|--|
| CLI | <code>qc_engine.py</code> | Local development, ad-hoc testing, manual QC runs |
| Box Hotfolder | <code>ford_qc_box_hotfolder_process.py</code> | Production — monitors a Box folder, auto-processes new uploads |

Both modes execute the same QC checks via the same profile. The difference is how files arrive and where reports go.

High-Level Pipeline



Component Architecture



Directory Structure

```
Ford_BnP_QC/
|-- checks/                                # QC check modules + reporters
|   |-- __init__.py
|   |-- unzip_and_verify_check.py          # Check 1: ZIP extraction & manifest validation
|   |-- colour_existence_check.py          # Check 2: Color chip file presence
|   |-- missing_images_check.py            # Check 3: Image file existence
|   |-- special_requirements_mec_bau.py     # Check 4: MEC/BAU rules
|   |-- mec_powertrain_validation.py        # Check 5: MEC powertrain requirements
|   |-- lifestyle_inventory_validation.py    # Check 6: Lifestyle/inventory rules
|   |-- image_resolution_check.py          # Check 7: Image dimensions
|   |-- image_format_check.py              # Check 8: Image format validation
|   |-- file_size_check.py                 # Check 9: File size limits
|   |-- image_linking_check.py              # Check 10: Orphan image detection
|   |-- layer_depth_check.py               # Check 11: Layer/filename consistency
|   |-- check_series_permutations.py        # Check 12: Series combination coverage
|   |-- beltline_validation_check.py        # Check 13: Beltline frame validation
|   |-- extra_carousel_validation_check.py  # Check 14: Extra carousel WERS coverage
|   |-- html_reporter.py                   # Generates HTML QC reports
|   |-- html_error_reporter.py              # Generates HTML error reports
|-- utils/
|   |-- config.py                           # Centralized environment configuration
|   |-- path_resolver.py                     # Working directory resolution
|   |-- check_helpers.py                     # Shared check utilities
|-- profiles/
|   |-- ford_bnp.json                       # Production QC profile (14 checks)
|-- tests/                                  # Unit tests for individual checks
|-- example_packs/                          # Sample asset packs for testing (~37GB)
|-- working/                                # Temp extraction directory (gitignored)
|-- reports/                                # Local QC reports output (gitignored)
|-- download_tmp/                           # Box download cache (gitignored)
|-- log/                                    # Service logs (gitignored)
|-- qc_engine.py                             # CLI entry point
|-- qc_module.py                             # Library interface (used by hotfolder)
|-- ford_qc_box_hotfolder_process.py          # Production Box monitoring service
|-- ford-qc-hotfolder-prod.service            # Systemd service (production)
|-- ford-qc-hotfolder-dev.service            # Systemd service (development)
|-- .env.example                             # Environment config template
|-- requirements.txt                          # Python dependencies
|-- CLAUDE.md                               # AI assistant instructions
```

Technology Stack

| Component | Technology | Version | Purpose |
|--------------------|----------------------|-----------------|---------------------------------|
| Runtime | Python | 3.8+ | Core language |
| Image Processing | Pillow | 11.1.0 | Resolution/format validation |
| Cloud Storage | Box SDK | 3.13.0 | Box API integration |
| Authentication | PyJWT + cryptography | 2.10.1 / 44.0.0 | Box JWT auth |
| Configuration | python-dotenv | 1.0.1 | Environment variable management |
| HTTP | requests | 2.32.3 | Network operations |
| Process Management | systemd | 235 | Production service lifecycle |

Data Model & Asset Pack Structure

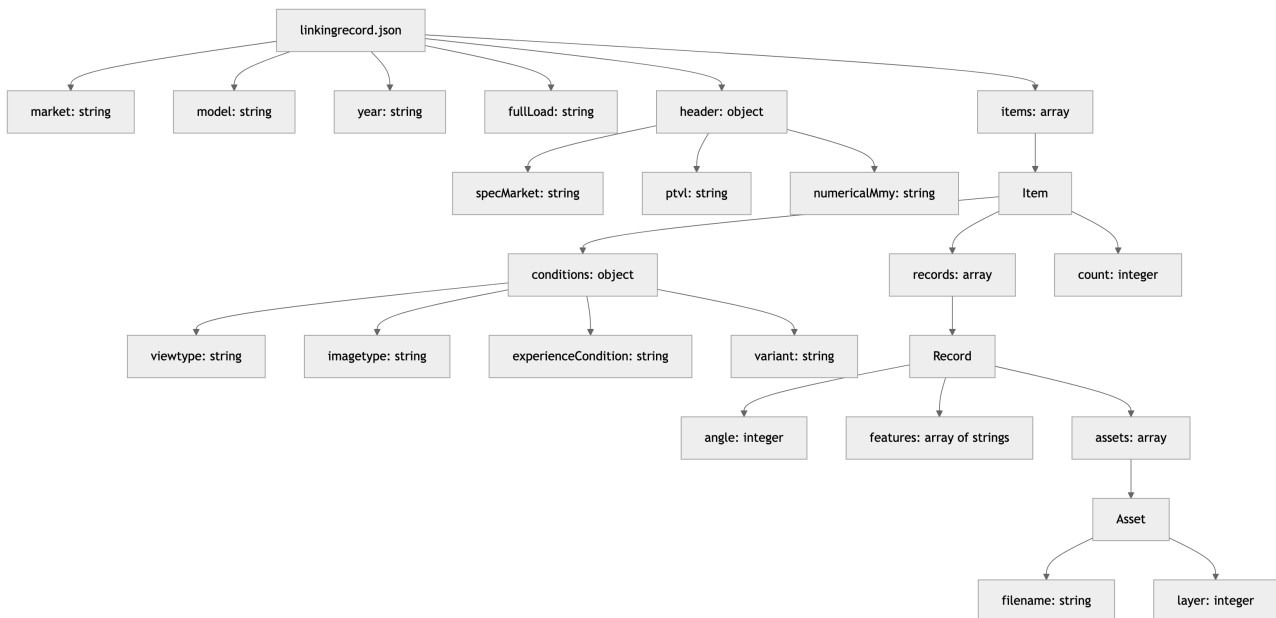
Asset Pack Overview

A Ford BnP asset pack is a ZIP file containing:

1. `linkingrecord.json` — The manifest that describes every image, its category, viewing angle, associated features, and layering info
2. **Image files** — Organized in subdirectories by viewtype and imagetype

The `linkingrecord.json` is the single source of truth. Every QC check reads this file to understand what the pack should contain.

linkingrecord.json Schema



Top-Level Fields

| Field | Type | Example | Validation Rules |
|-----------------------|--------|---------|-----------------------------------|
| <code>market</code> | string | "DEU" | Exactly 3 uppercase letters (A-Z) |
| <code>model</code> | string | "CGW07" | Uppercase letters and digits only |
| <code>year</code> | string | "YYY" | Must be the literal string "YYY" |
| <code>fullLoad</code> | string | "Y" | Load indicator |
| <code>header</code> | object | — | Contains metadata fields |
| <code>items</code> | array | — | Array of item objects |

Header Fields

| Field | Type | Example | Validation Rules |
|---------------------------|--------|----------|---------------------------|
| <code>specMarket</code> | string | "WAEDX" | Specification market code |
| <code>ptvl</code> | string | "CGW" | Platform code |
| <code>numericalMmy</code> | string | "202400" | Must contain no dots |

Item Fields

Each item represents a group of images sharing the same viewtype/imagetype conditions.

| Field | Type | Example | Description |
|---|---------|-----------------|---------------------------------------|
| <code>conditions.viewtype</code> | string | "exterior" | View category (see matrix below) |
| <code>conditions.imagetype</code> | string | "layeroptext" | Image subcategory (see matrix below) |
| <code>conditions.experienceCondition</code> | string | "2d-background" | Optional. Presence indicates MEC pack |
| <code>conditions.variant</code> | string | "desktop" | Optional. Used for beltline images |
| <code>records</code> | array | — | Array of record objects |
| <code>count</code> | integer | 8 | Number of records in this item |

Record Fields

Each record represents a specific configuration (angle + feature combination) with its image files.

| Field | Type | Example | Description |
|-----------------------|---------|-------------------------------|-----------------------------|
| <code>angle</code> | integer | 21 | Camera viewing angle |
| <code>features</code> | array | ["bs-sa", "vs-kz", "pn4hq"] | WERS feature/option codes |
| <code>assets</code> | array | — | Image files for this record |

Asset Fields

| Field | Type | Example | Description |
|-----------------------|---------|-----------------------------|---------------------------------------|
| <code>filename</code> | string | "ext/base/ext_21_bs-sa.jpg" | File path relative to ZIP root |
| <code>layer</code> | integer | 0 | Layer stacking order (0 = base image) |

ViewType / ImageType Matrix

| ViewType | ImageType | Description | Format | Resolution (BAU) |
|-----------------|-------------|------------------------------|---------------------|------------------|
| exterior | (none) | Base exterior renders | JPEG (PNG accepted) | 1600x900 |
| exterior | layeroptext | Exterior layered options | PNG | 1600x900 |
| exterior | showroom | Showroom previews | PNG | 768x432 |
| interior | (none) | Base interior renders | JPEG (PNG accepted) | 1600x900 |
| interior | layeroptint | Interior layered options | PNG | 1600x900 |
| carousel | colour | Color chip thumbnails | JPEG | 148x83 |
| carousel | powertrain | Powertrain images | JPEG | 678x381 |
| carousel | extra | Extra option carousel images | JPEG | 678x381 |
| carousel | bodystyle | Body style images | JPEG | 678x381 |
| carousel | series | Series images | JPEG | 678x381 |
| carousel | trim | Trim images | JPEG | 678x381 |
| lifestyle | (none) | Lifestyle/marketing photo | JPEG | 1440x810 |
| inventory | (none) | Inventory/spec image | JPEG | 2900x1280 |
| vehicleselector | desktop | Beltline desktop frames | AVIF | 1920x842 |
| vehicleselector | mobile | Beltline mobile frames | AVIF | 640x428 |

MEC packs use different resolutions for some types — see the [QC Checks Reference](#).

Pack Types: MEC vs BAU

Asset packs come in two variants with different validation rules:

BAU (Business As Usual)

- Standard configurator packs
- No `experienceCondition` field present on any item
- Standard resolution and format requirements

MEC (Multi-Experience Configurator)

- Enhanced packs with 2D background experience

- **Detection:** Any item with `conditions.experienceCondition = "2d-background"` makes the entire pack MEC
- Additional requirements:
- Must have exterior 2d-background section with angle 30
- Must have carousel/powertrain with assets
- Required section pairs (regular + 2d-background) for exterior base, interior base, exterior layered options, interior layered options
- Different resolution requirements (e.g., 1600x1600 instead of 1600x900 for base images)
- Stricter format rules (base assets must be JPEG only, no PNG)
- 100% WERS code coverage required for extra carousel (vs 50% minimum for BAU)

Allowed File Types

Only these file extensions are permitted in asset packs:

| Extension | Case Rule | Used For |
|--------------------|---------------------------|-----------------------------------|
| <code>.jpg</code> | Lowercase | Most image types |
| <code>.jpeg</code> | Lowercase | Alternative JPEG extension |
| <code>.png</code> | Lowercase | Layer options, showroom |
| <code>.AVIF</code> | Uppercase required | Beltline (vehicleselector) images |
| <code>.json</code> | Lowercase | Linking record manifest |

Any other extensions are flagged as unauthorized. Lowercase `.avif` is flagged as a case violation.

Asset Naming Conventions

General Pattern

Filenames typically follow: `{viewtype_prefix}/{subdir}/{viewtype}_{angle}_{features}.{ext}`

Example: `ext/base/ext_21_bs-sa.jpg`

Beltline Files

- Desktop: `vsd_{frame}_{features}.AVIF` (frames 0-71)
- Mobile: `vsm_{frame}_{features}.AVIF` (frames 49-71)

MEC-Specific Files

- Files with `_mec` in the filename are only allowed in `2d-background` `experienceCondition` sections

Layer Numbering

- Layer 0 = base image (no layer suffix needed)

- Layer N > 0 = overlay image; the layer number must appear as the last number before the file extension in the filename (e.g., `_21_20.png` means layer 20)

Feature Codes

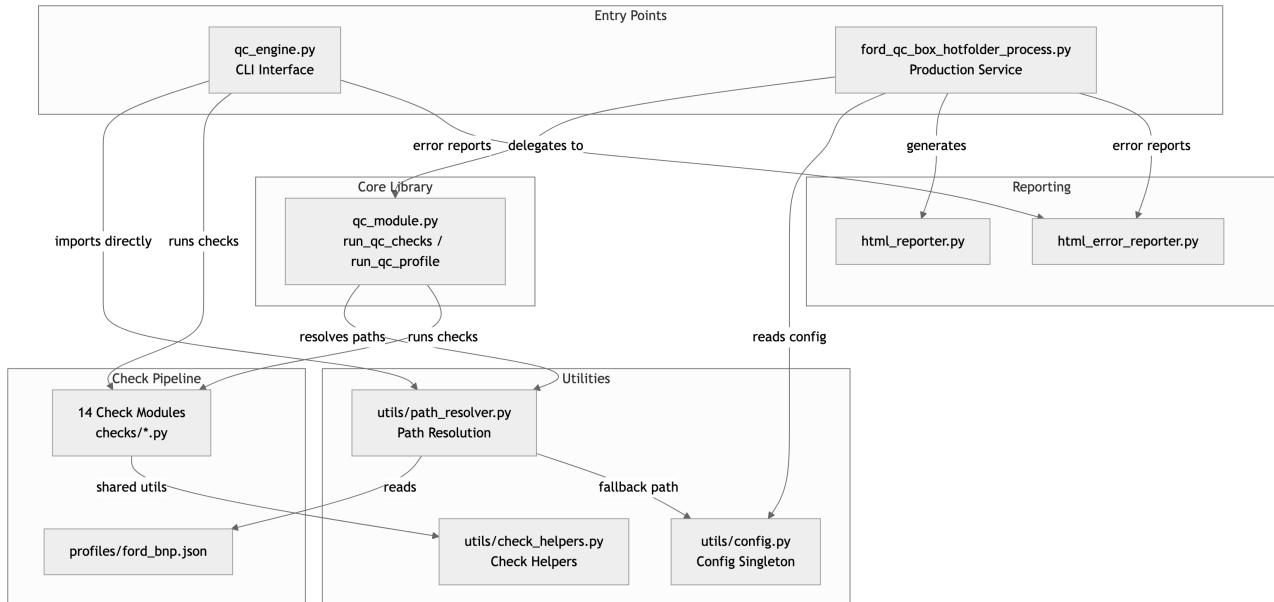
The `features` array on each record contains WERS (World Engineering Release System) codes that identify vehicle options:

- **vs- prefix:** Vehicle Series codes (e.g., `vs-kz`)
- **acm prefix:** ACM codes (e.g., `acm5b`)
- **bs- prefix:** Body style codes (e.g., `bs-sa`)
- **Other codes:** Paint colors, trim options, packages (e.g., `pn4hq`)

These codes are used by `check_series_permutations` and `extra_carousel_validation_check` to verify coverage completeness.

Architecture & Data Flow

Module Relationships



Key distinction: `qc_engine.py` imports check modules and runs them directly. The Box hotfolder delegates to `qc_module.py`, which provides the same functionality as a library with enhanced error handling.

CLI Workflow



CLI Usage

```
# Standard run
python qc_engine.py profiles/ford_bnp.json --input_file asset_pack.zip --reports_dir reports

# Arguments:
#   profile      (positional) Path to QC profile JSON
#   --input_file (optional)   Path to asset pack ZIP
```

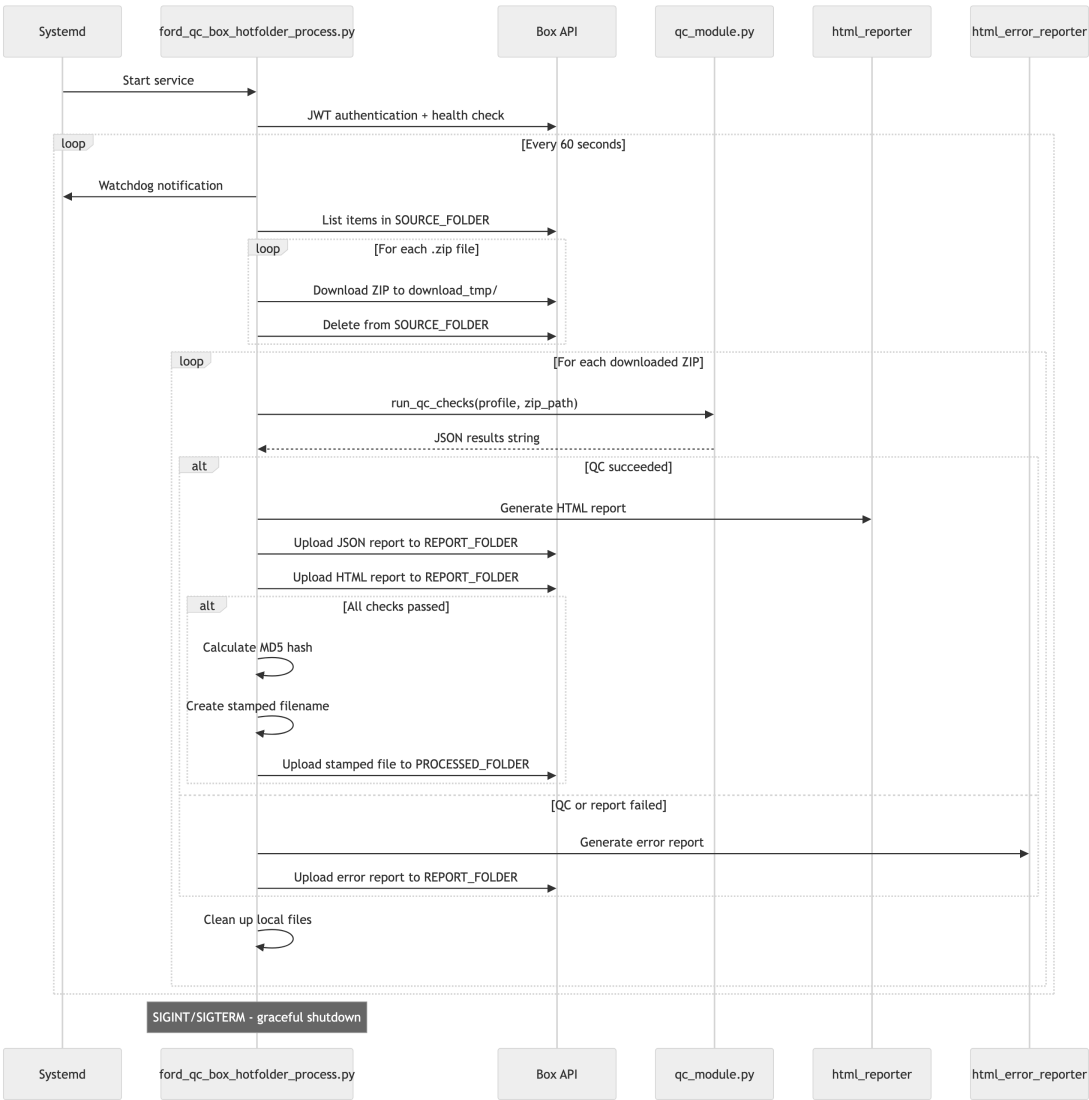
```
# --reports_dir (optional)    Output directory (default: reports)
```

CLI Output

Results are written to `reports/YYYYMMDD_HHMMSS/qc_results.json`. The CLI does **not** automatically generate the HTML report — use the HTML reporter separately:

```
python -m checks.html_reporter reports/YYYYMMDD_HHMMSS/qc_results.json reports/
```

Box Hotfolder Workflow



Box Folder Structure

| Folder | Purpose | Folder ID (via .env) |
|--------|--|-----------------------------------|
| Source | Incoming ZIP uploads — monitored for new files | <code>BOX_SOURCE_FOLDER_ID</code> |

| Folder | Purpose | Folder ID (via .env) |
|-----------|---|--------------------------------------|
| Reports | QC reports (HTML + JSON) uploaded here | <code>BOX_REPORT_FOLDER_ID</code> |
| Processed | Passed files re-uploaded with MD5 stamp | <code>BOX_PROCESSED_FOLDER_ID</code> |

File Stamping

When all checks pass, the original ZIP is re-uploaded with an MD5 hash appended to the filename:

```
input:  pv_waedx_cgw07_yyi_24072024.zip
output: pv_waedx_cgw07_yyi_24072024_alb2c3d4e5f6...zip
```

This stamp serves as proof of QC completion and file integrity.

Profile System

How It Works

QC profiles are JSON arrays of check definitions. Each entry specifies a check module and its configuration:

```
{
  "script": "checks.image_resolution_check",
  "config": {
    "working_dir": "__WORKING_DIR__",
    "linkingrecord_filename": "linkingrecord.json"
  }
}
```

`__WORKING_DIR__` Substitution

The placeholder `__WORKING_DIR__` appears in profile configs. Before execution:

1. `path_resolver.update_profile_paths()` reads the profile
2. Replaces all `"__WORKING_DIR__"` with the actual working directory path
3. Writes a temporary profile (e.g., `ford_bnp_temp.json`)
4. Returns the temp path for use during the QC run
5. The caller cleans up the temp file after completion

Dynamic Module Loading

Checks are loaded at runtime via `importlib.import_module()`. The `script` field uses Python's dotted module path notation (e.g., `checks.image_resolution_check`), which means:

- Adding a new check only requires creating the module file and adding it to the profile
- No code changes needed in the engine or module
- Checks can be reordered or removed by editing the profile JSON

Configuration System

utils/config.py — Singleton Pattern

The `Config` class uses a singleton pattern — only one instance exists per process:

```
from utils.config import config # Always returns the same instance
folder_id = config.BOX_SOURCE_FOLDER_ID
```

Loading Precedence

Environment variables are loaded in this order (highest priority first):

1. **System environment variables** — Always take precedence
2. `.env.{FORD_QC_ENV}` **file** — Environment-specific (e.g., `.env.production`, `.env.development`)
3. `.env` **file** — Base defaults

The `FORD_QC_ENV` variable itself must be set via system environment (not in a `.env` file) since it determines which `.env` file to load.

Required Variables

The following must be set or the `Config` class raises `ConfigurationError` on initialization:

| Variable | Type | Default | Description |
|--------------------------------------|--------|---------|--|
| <code>BOX_SOURCE_FOLDER_ID</code> | string | — | Box folder for incoming files |
| <code>BOX_REPORT_FOLDER_ID</code> | string | — | Box folder for reports |
| <code>BOX_PROCESSED_FOLDER_ID</code> | string | — | Box folder for processed files |
| <code>BOX_CONNECTION_TIMEOUT</code> | int | 30 | Connection timeout (seconds) |
| <code>BOX_READ_TIMEOUT</code> | int | 90 | Read timeout (seconds) |
| <code>MAX_RETRIES</code> | int | 3 | Max retry attempts |
| <code>RETRY_BACKOFF_BASE</code> | int | 2 | Exponential backoff base |
| <code>WATCHDOG_INTERVAL</code> | int | 30 | Watchdog notification interval (seconds) |

Optional Variables

| Variable | Type | Default | Description |
|-----------------------------------|--------|-------------------------------------|----------------------------|
| <code>FORD_QC_ENV</code> | string | <code>"production"</code> | Environment name |
| <code>BOX_CONFIG_FILE</code> | path | <code>ford_box_config.json</code> | Box JWT config file |
| <code>WORKING_DIR</code> | path | <code>working</code> | Extraction directory |
| <code>DOWNLOAD_PATH</code> | path | <code>download_tmp</code> | Temp download directory |
| <code>QC_PROFILE_PATH</code> | path | <code>profiles/ford_bnp.json</code> | QC profile path |
| <code>FALLBACK_WORKING_DIR</code> | path | (hardcoded) | Fallback working directory |

| Variable | Type | Default | Description |
|-----------|--------|---------|-----------------------------------|
| LOG_LEVEL | string | "INFO" | DEBUG/INFO/WARNING/ERROR/CRITICAL |

Relative paths are resolved against the script directory (Config.SCRIPT_DIR).

Error Handling Architecture

The system has three layers of error handling, each designed for its context:

Layer 1: `qc_engine.py` (CLI)

```
try:
    update_profile_paths()          → qc_profile error report
    try:
        run_qc_profile()          → type-specific error report
        try:
            write_results_report() → unknown error report
```

Error type detection uses keyword matching on the exception message (zip, json, file, check) to select the appropriate error reporter.

Layer 2: `qc_module.py` (Library)

- `run_single_check()` catches import errors and execution errors separately, returning error dicts instead of raising
- `run_qc_profile()` catches `FileNotFoundError`, `JSONDecodeError`, `PermissionError` individually
- `run_qc_checks()` lets exceptions propagate to the caller (the hotfolder process)

Layer 3: `ford_qc_box_hotfolder_process.py` (Service)

Every stage has independent error handling:

| Stage | Error Action |
|------------------------|--|
| Box authentication | Generate network error report, return early |
| File download | Generate network error report, continue to next file |
| QC check execution | Generate QC check error report, upload, clean up |
| HTML report generation | Generate error report, upload, clean up |
| Report upload | Generate network error report, clean up |
| Stamped file upload | Generate network error report, clean up |

The service **never crashes** — errors are reported and processing continues with the next file.

Retry & Resilience

Exponential Backoff

The `@retry_with_backoff` decorator (in `ford_qc_box_hotfolder_process.py`) handles transient network failures:

- Catches `Timeout`, `ConnectionError`, `RequestException`
- Retries up to `MAX_RETRIES` times (default: 3)
- Wait time: `RETRY_BACKOFF_BASE ^ attempt` (default: $2^0=1s$, $2^1=2s$, $2^2=4s$)
- Non-network exceptions (`Exception`) fail immediately without retry

Applied to: `get_box_folder_items()` and other Box API calls.

Graceful Shutdown

Signal handlers for `SIGINT` and `SIGTERM` set a `shutdown_requested` flag. The main loop checks this flag:

- Between file downloads
- Before starting each file's QC processing
- During the sleep cycle (checked every `WATCHDOG_INTERVAL` seconds)

The current file finishes processing before shutdown.

Systemd Watchdog

The service sends watchdog notifications to systemd during:

- Each iteration of the main loop (every 60 seconds)
- Each sleep interval check (every `WATCHDOG_INTERVAL` seconds)

The systemd timeout is 3600 seconds (1 hour), allowing for large asset packs that take significant time to process.

Service Restart

Systemd automatically restarts the service on failure with a 30-second delay. Restart is limited to 5 attempts per 600 seconds to prevent restart loops.

QC Checks Reference

This document covers all 14 production checks in execution order as defined in `profiles/ford_bnp.json`. Each check implements a `run_check(config)` function that returns a result dict with a `status` field.

Status values: `"passed"`, `"failed"`, `"error"`, or `"skipped"`

All checks receive a `config` dict containing at minimum `working_dir` (path to extracted files) and typically `linkingrecord_filename` (default: `"linkingrecord.json"`).

1. Archive Extraction & Verification

Source: `checks/unzip_and_verify_check.py`

Extracts the asset pack ZIP and validates the manifest and file structure.

What It Validates

- ZIP file is valid and extractable
- `linkingrecord.json` exists in the extracted contents
- JSON syntax is valid
- Header fields conform to naming rules
- Only allowed file extensions are present
- No extraneous files on disk

Header Validation Rules

| Field | Rule | Example Valid | Example Invalid |
|---------------------------|---|---------------------|--|
| <code>market</code> | Exactly 3 uppercase letters (A-Z) | <code>DEU</code> | <code>De1</code> , <code>DEUU</code> |
| <code>model</code> | Uppercase letters and digits only | <code>CGW07</code> | <code>cgw07</code> , <code>CGW-07</code> |
| <code>year</code> | Must be literal string <code>"YYY"</code> | <code>YYY</code> | <code>2024</code> , <code>yyy</code> |
| <code>numericalMmy</code> | String with no dots | <code>202400</code> | <code>2024.00</code> |

File Type Rules

| Extension | Allowed | Case Rule |
|--------------------|---------|--------------------------|
| <code>.jpg</code> | Yes | Lowercase |
| <code>.jpeg</code> | Yes | Lowercase |
| <code>.png</code> | Yes | Lowercase |
| <code>.AVIF</code> | Yes | Must be uppercase |

| Extension | Allowed | Case Rule |
|--------------------|---------|-------------------------|
| <code>.json</code> | Yes | Lowercase |
| Anything else | No | Flagged as unauthorized |

Lowercase `.avif` is flagged as a case violation.

Pass/Fail

- **Passes:** Valid ZIP, linkingrecord.json present with valid JSON and headers, no unauthorized files
- **Fails:** Any header validation error, unauthorized file types, case violations, or extraneous files

2. Color Chips Verification

Source: `checks/colour_existence_check.py`

Validates that color chip images exist and are accessible.

What It Validates

- At least one item with `image_type="colour"` exists in the linking record
- Every filename referenced by colour items exists on disk

Pass/Fail

- **Passes:** Colour items found and all referenced files exist
- **Fails:** No colour items found, or any referenced colour image file is missing

3. Missing Images Check

Source: `checks/missing_images_check.py`

Cross-references every asset filename in `linkingrecord.json` with the filesystem.

What It Validates

- Every filename referenced in items → records → assets exists in the working directory
- File extensions are in the allowed set (`.jpg`, `.jpeg`, `.png`, `.AVIF`, `.json`)
- AVIF extensions use uppercase `.AVIF`

Pass/Fail

- **Passes:** All referenced files exist with valid extensions
- **Fails:** Any missing files, unauthorized file types, or AVIF case violations

Output Details

Reports three categories: `unauthorized_file_types`, `case_violations`, and `missing_files` — each with counts and file lists.

4. MEC/BAU Special Requirements

Source: `checks/special_requirements_mec_bau.py`

Validates pack-type-specific structural requirements and `_mec` filename rules.

Pack Type Detection

A pack is **MEC** if any item has `experienceCondition="2d-background"`. Otherwise it is **BAU**.

MEC Pack Validation

| Rule | Description |
|--|--|
| Exterior 2d-background must exist | At least one exterior item with <code>experienceCondition="2d-background"</code> |
| Angle 30 powertrain image required | The 2d-background section must contain an <code>angle=30</code> record |
| Required section pairs | Both regular and 2d-background versions must exist for: exterior base, interior base, exterior layered options, interior layered options |
| <code>_mec</code> filename restriction | Files with <code>_mec</code> in the name are only allowed in 2d-background sections |

BAU Pack Validation

| Rule | Description |
|--|--|
| No <code>_mec</code> files outside 2d-background | If any <code>_mec</code> files exist outside 2d-background sections, the check fails |

Pass/Fail

- **Passes:** All section requirements met and `_mec` files correctly placed
- **Fails:** Missing required sections, missing angle 30, or `_mec` files in wrong sections

5. MEC Powertrain Validation

Source: `checks/mec_powertrain_validation.py`

Validates MEC-specific powertrain requirements. **Automatically passes for BAU packs.**

What It Validates (MEC Only)

| Rule | Description |
|---------------------------------|---|
| Carousel/powertrain must exist | Items with viewtype=carousel, imagetype=powertrain must be present |
| Powertrain must have assets | The carousel/powertrain item must contain at least one asset |
| Exterior 2d-background angle 30 | Exterior item with experienceCondition="2d-background" must have angle=30 |

Pass/Fail

- **Passes (BAU):** Always — returns "Not a MEC pack"
- **Passes (MEC):** All powertrain requirements met
- **Fails (MEC):** Missing carousel/powertrain, empty assets, or missing angle 30

6. Lifestyle & Inventory Validation

Source: `checks/lifestyle_inventory_validation.py`

Validates lifestyle and inventory entry constraints.

What It Validates

| Rule | Requirement |
|---------------------------|---|
| Inventory angle | All inventory entries must have <code>angle=2</code> |
| Lifestyle angle | All lifestyle entries must have <code>angle=1</code> |
| Lifestyle count | Exactly 1 lifestyle entry allowed |
| Inventory-showroom parity | Number of inventory entries must equal number of showroom entries |

Pass/Fail

- **Passes:** All angle and count rules satisfied
- **Fails:** Wrong angles, multiple lifestyle entries, or inventory/showroom count mismatch
- **Skipped:** No inventory or lifestyle entries found (not applicable)

7. Image Resolution Check

Source: `checks/image_resolution_check.py`

Validates image pixel dimensions against Ford specifications. Resolution requirements differ between BAU and MEC packs.

BAU Resolution Requirements

| ViewType | ImageType | Width | Height |
|-----------------|----------------|-------|--------|
| exterior | (base) | 1600 | 900 |
| interior | (base) | 1600 | 900 |
| exterior | layeroptionext | 1600 | 900 |
| interior | layeroptionint | 1600 | 900 |
| exterior | showroom | 768 | 432 |
| lifestyle | — | 1440 | 810 |
| inventory | — | 2900 | 1280 |
| carousel | extra | 678 | 381 |
| carousel | powertrain | 678 | 381 |
| carousel | colour | 148 | 83 |
| carousel | bodystyle | 678 | 381 |
| carousel | series | 678 | 381 |
| carousel | trim | 678 | 381 |
| vehicleselector | desktop | 1920 | 842 |
| vehicleselector | mobile | 640 | 428 |

MEC Resolution Differences

These types use different resolutions in MEC packs (all others are same as BAU):

| ViewType | ImageType | BAU | MEC |
|----------|----------------|----------|-----------|
| exterior | (base) | 1600x900 | 1600x1600 |
| interior | (base) | 1600x900 | 1600x1600 |
| exterior | layeroptionext | 1600x900 | 1600x1600 |
| interior | layeroptionint | 1600x900 | 1600x1600 |
| carousel | extra | 678x381 | 1280x720 |
| carousel | powertrain | 678x381 | 1600x1600 |

MEC Override Logic

Even in MEC packs, some items use BAU resolution:

- **Exterior/interior without experienceCondition** — These are regular (non-2d-background) sections that still use BAU resolution
- **Powertrain images** — Always use BAU resolution regardless of pack type

Pass/Fail

- **Passes:** All images match their expected dimensions
- **Fails:** Any image has wrong width or height (deduplicated by filename)
- Missing files are silently skipped (handled by check #3)

8. Image Format Validation

Source: `checks/image_format_check.py`

Validates image file formats (JPEG, PNG, AVIF) based on viewtype/imagetype.

Format Requirements

| ViewType | ImageType | Required Format |
|-----------------|------------------|-----------------|
| exterior | layeroptext | PNG |
| interior | layeroptint | PNG |
| exterior | showroom | PNG |
| carousel | extra | JPEG |
| carousel | powertrain (BAU) | JPEG |
| carousel | colour | JPEG |
| carousel | bodystyle | JPEG |
| carousel | series | JPEG |
| carousel | trim | JPEG |
| lifestyle | — | JPEG |
| inventory | — | JPEG |
| vehicleselector | desktop | AVIF |
| vehicleselector | mobile | AVIF |

Base Asset Special Rules

| Pack Type | Base Asset Rule |
|-----------|---|
| BAU | JPEG preferred; PNG accepted with warning |
| MEC | Strict JPEG only — PNG fails the check |

AVIF Validation

AVIF files are validated by extension rather than PIL format detection. The extension must be uppercase `.AVIF`. Lowercase `.avif` is flagged as a case violation.

Pass/Fail

- **Passes:** All images match their required format
- **Fails:** Any format mismatch (deduplicated by filename)

9. File Size Validation

Source: `checks/file_size_check.py`

Validates file sizes with a three-tier system: compliant, advisory warning, and violation.

Size Limits

| Asset Type | Max Size (KB) |
|--|---------------|
| Base Exterior Images | 750 |
| Base Interior Images | 1024 |
| Engine and Transmission Images (MEC 2d-background) | 1024 |
| Interior Layered Option Images | 1024 |
| Exterior Layered Option Images | 750 |
| Option Carousel Images (extra) | 500 |
| Powertrain Images | 600 |
| Showroom Images | 200 |
| Colour Chips | 30 |
| Bodystyle Images | 50 |
| Series Images | 50 |
| Trim Images | 800 |
| Lifestyle Images | 400 |
| Inventory Images | 800 |
| Beltline Images | 800 |

Three-Tier Validation

| Tier | Range | Result |
|------------------|-------------------|---------------------|
| Compliant | 0–100% of limit | Pass |
| Advisory Warning | 101–125% of limit | Pass (with warning) |
| Violation | 126%+ of limit | Fail |

The 25% buffer zone allows files slightly over the limit to pass with an advisory warning, while significantly oversized files cause a failure.

Pass/Fail

- **Passes:** No violations (advisory warnings are permitted)
- **Fails:** Any file exceeds 125% of its size limit

10. Image Linking Check

Source: `checks/image_linking_check.py`

Reverse-direction check: finds images on disk that are NOT referenced in `linkingrecord.json`.

What It Validates

- Recursively walks the working directory for `.jpg`, `.jpeg`, `.png` files
- Compares against all filenames referenced in the linking record
- Reports any **unreferenced** (orphan) files

This is the complement of Check #3 (Missing Images): Check #3 finds files in JSON but not on disk; Check #10 finds files on disk but not in JSON.

Note: `linkingrecord.json` itself and `.AVIF` files are excluded from this check.

Pass/Fail

- **Passes:** Every image file on disk is referenced in the linking record
- **Fails:** Any unreferenced image files found (sorted list provided)

11. Layer Depth Validation

Source: `checks/layer_depth_check.py`

Validates consistency between the `layer` property in the linking record and the layer number embedded in filenames.

What It Validates

| Asset Type | Rule |
|---|--|
| Layered options (<code>layeroptint</code> , <code>layeroptext</code>) | <code>layer</code> must be > 0 and match the last number in the filename |
| Base assets (no imagetype) | <code>layer</code> must be 0 or absent |

Filename Layer Extraction

The layer number is extracted from the filename using the pattern: last number before the file extension.

Example: `int/opts/int_21_20.png` → layer number is 20

Pass/Fail

- **Passes:** All layer values match their filenames and base assets have layer=0
- **Fails:** Any mismatch between layer property and filename, or base asset with non-zero layer

12. Series Permutations Check

Source: `checks/check_series_permutations.py`

Validates that the series section covers all VS/ACM combinations found in exterior and showroom sections.

What It Validates

1. Builds a "source of truth" by extracting unique [ACM, VS] pairs from: - Exterior items (viewtype=exterior, no imagetype, no experienceCondition) - Showroom items (viewtype=exterior, imagetype=showroom)
2. Extracts [ACM, VS] pairs from series items (viewtype=carousel, imagetype=series)
3. Reports any combinations present in exterior/showroom but **missing** from series

Feature Code Extraction

- **VS codes:** Features with prefix `vs-` (case-sensitive)
- **ACM codes:** Features with prefix `acm` (case-sensitive)
- Each record represents exactly ONE combination (not a cross-product)

Pass/Fail

- **Passes:** All exterior/showroom combinations are covered in series
- **Fails:** Any missing combinations
- Also reports extra combinations (in series but not in exterior/showroom) as informational

13. Beltline Validation

Source: `checks/beltline_validation_check.py`

Comprehensive validation of vehicleselector beltline images (desktop and mobile frame sequences).

What It Validates

| Aspect | Desktop | Mobile |
|---------------------|--|--|
| Frame range | 0–71 (72 frames) | 49–71 (23 frames) |
| File pattern | <code>vsd_{frame}_{features}.AVIF</code> | <code>vsm_{frame}_{features}.AVIF</code> |
| Extension | Uppercase <code>.AVIF</code> required | Uppercase <code>.AVIF</code> required |
| Angle consistency | <code>angle</code> field must match frame number | <code>angle</code> field must match frame number |
| Variant consistency | <code>imagetype</code> must equal <code>variant</code> field | <code>imagetype</code> must equal <code>variant</code> field |

Additional Validations

- **Missing frames:** Any frame numbers in the expected range not present in the linking record
- **Missing files:** Files referenced in the linking record but not on disk
- **Orphaned files:** Beltline files on disk (`vsd_*` / `vsm_*`) not referenced in the linking record
- **Case violations:** `.avif` (lowercase) instead of `.AVIF`
- **Extra mobile frames:** Mobile frames 0–48 should not exist (warning if found)

Pass/Fail

- **Passes:** All frame ranges complete, all files present, no orphans, no case violations
- **Fails:** Missing frames, missing files, unreferenced files, or case violations

14. Extra Carousel Validation

Source: `checks/extra_carousel_validation_check.py`

Validates that carousel/extra images provide adequate WERS feature code coverage.

What It Validates

For each record in carousel/extra items:

1. Extracts feature codes from the record's `features` array
2. Checks if asset filenames contain the feature code as a substring
3. Calculates coverage percentage: matched features / total features

Coverage Thresholds

| Pack Type | 0–49% | 50–99% | 100% |
|-----------|-------|----------------|------|
| BAU | Fail | Warning (pass) | Pass |
| MEC | Fail | Fail | Pass |

MEC packs require **100% feature coverage** — every WERS code must have a corresponding image file. BAU packs allow partial coverage with a warning above 50%.

Pass/Fail

- **Passes:** Coverage meets the threshold for the pack type
- **Fails:** Coverage below threshold
- Reports per-record coverage details and lists any missing features

Check Execution Order

The checks run sequentially in the order listed in `profiles/ford_bnp.json`. This order matters because:

1. **Check 1 (unzip_and_verify)** must run first — it extracts the ZIP and creates the working directory used by all subsequent checks
2. **Checks 2-3** (colour, missing images) validate basic file presence before deeper analysis
3. **Checks 4-6** (MEC/BAU, powertrain, lifestyle/inventory) validate structural/business rules
4. **Checks 7-9** (resolution, format, size) validate image properties
5. **Checks 10-14** (linking, layers, series, beltline, extra carousel) validate cross-referencing and coverage

If Check 1 fails (can't extract ZIP or can't find linkingrecord.json), all subsequent checks will likely error because they depend on the extracted files.

Skipped Types Handling

Many checks encounter viewtype/imagetype combinations they don't have rules for. Rather than failing, they:

1. Call `check_helpers.record_skipped_type()` to log the unknown combination
2. Skip validation for that item
3. Include the skipped types in the result for transparency
4. Still return `"passed"` if no validated items failed

This design ensures new viewtype/imagetype combinations don't cause false failures while remaining visible in reports.

Reports & Output

Report Types

The system produces two types of reports:

| Report | Generator | When |
|--------------|-------------------------------|------------------------------------|
| QC Report | checks/html_reporter.py | After successful QC processing |
| Error Report | checks/html_error_reporter.py | When processing fails at any stage |

JSON Results Structure

Before HTML generation, QC results are stored as JSON. This is the intermediate format between the check pipeline and the HTML reporter.

Top-Level Schema

```
{
  "profile": "ford_bnp_temp.json",
  "timestamp": "2025-11-07T16:45:42.017547Z",
  "checks": [
    {
      "index": 1,
      "script": "checks.unzip_and_verify_check",
      "config": { "working_dir": "/path/to/working", ... },
      "result": {
        "status": "passed",
        "details": { ... }
      }
    }
  ]
}
```

Per-Check Result Schema

Every check result has:

| Field | Type | Description |
|---------------|--------|---|
| status | string | "passed", "failed", "error", or "skipped" |
| details | object | Check-specific details (varies by check) |
| error_message | string | Present only on "error" status |

Common detail fields across checks:

| Field | Found In | Description |
|---------|------------|------------------------|
| message | All checks | Human-readable summary |

| Field | Found In | Description |
|--------------------------------|--------------------------|---|
| <code>pack_type</code> | Resolution, format, size | "MEC" or "BAU" |
| <code>failed_images</code> | Resolution, format | List of images that failed validation |
| <code>missing_files</code> | Colour, missing images | List of files not found on disk |
| <code>skipped_types</code> | Many checks | ViewType/ImageType combinations the check didn't have rules for |
| <code>advisory_warnings</code> | File size | Files slightly over limit (101-125%) |
| <code>size_violations</code> | File size | Files significantly over limit (126%+) |

HTML QC Report

Generation

```
from checks.html_reporter import HTMLReporter

report_path = HTMLReporter.generate_report(
    json_data=qc_results_dict,
    reports_dir="reports/",
    input_filename="asset_pack.zip"
)
```

Or via CLI:

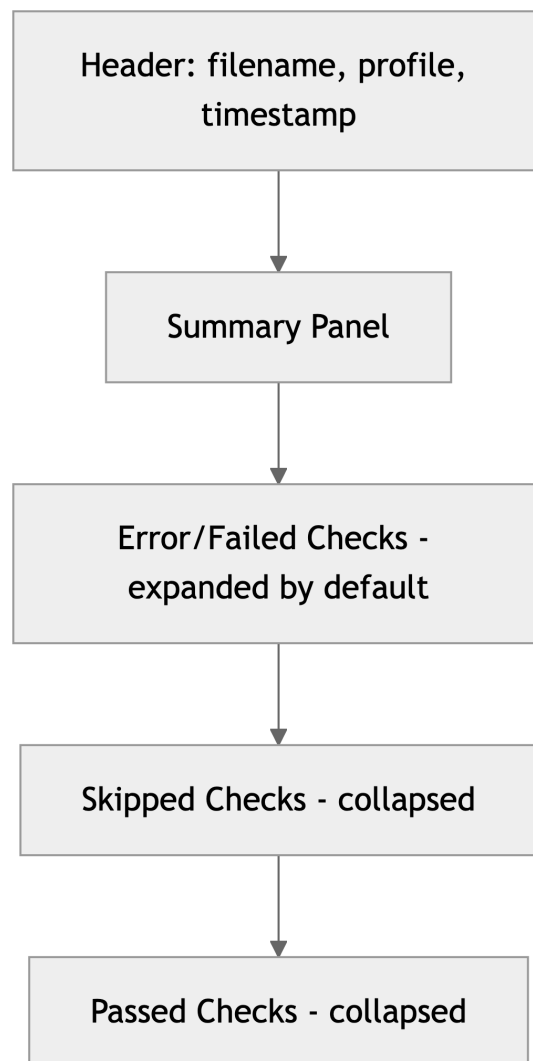
```
python -m checks.html_reporter reports/20251107_164544/qc_results.json reports/
```

Output Filename

```
{input_filename}_{ISO_timestamp}_QC.html
```

Example: `pv_waedx_cgww07_yyi_24072024_1530_image_2025-11-07T164544_QC.html`

Report Layout



The report uses Bootstrap 5.1.3 with accordion components. Checks are grouped by status:

| Group | Color | Default State |
|----------------|--------------|---------------|
| Failed / Error | Red badge | Expanded |
| Skipped | Yellow badge | Collapsed |
| Passed | Green badge | Collapsed |

Check Name Display

Module paths are converted to friendly names in the report:

| Module Path | Display Name |
|--|---------------------------------|
| <code>checks.unzip_and_verify_check</code> | Archive Extraction Verification |
| <code>checks.colour_existence_check</code> | Color Chips Verification |
| <code>checks.missing_images_check</code> | Missing Images Check |
| <code>checks.image_resolution_check</code> | Image Resolution Validation |

| Module Path | Display Name |
|--|----------------------------------|
| <code>checks.image_format_check</code> | Image Format Validation |
| <code>checks.file_size_check</code> | File Size Validation |
| <code>checks.special_requirements_mec_bau</code> | MEC/BAU Special Requirements |
| <code>checks.mec_powertrain_validation</code> | MEC Powertrain Validation |
| <code>checks.lifestyle_inventory_validation</code> | Lifestyle & Inventory Validation |
| <code>checks.image_linking_check</code> | Image Link References Check |
| <code>checks.layer_depth_check</code> | Layer Depth Validation |

Checks not in this map get their module name auto-formatted (underscores replaced with spaces, title-cased).

Detail Rendering

The HTML reporter has specialized rendering for each check's detail fields:

- **Skipped types** → Table with viewtype/imagetype columns
- **Unauthorized file types / case violations** → Tables with file paths and locations
- **Extraneous files** → Table with paths and file sizes
- **Missing files** → Grouped by directory with sub-accordions
- **Resolution failures** → Tabbed view: grouped by resolution mismatch OR by image type
- **Format failures** → Tabbed view: grouped by format mismatch OR by image type
- **Size failures** → Tabbed view: grouped by severity (% over), asset type, OR directory
- **Layer failures** → Grouped by issue type and image type
- **Unreferenced files** → Grouped by directory and file extension
- **Extra carousel** → Per-record coverage with passed/failed breakdown, missing WERS codes listed

HTML Error Report

When It's Generated

Error reports are generated when processing fails at any stage — before, during, or after QC checks. Each error report includes actionable fix instructions for the end user.

Generation

```
from checks.html_error_reporter import HTMLErrorReporter

report_path = HTMLErrorReporter.generate_error_report(
    error_type='zip_extraction',
    error_message='Failed to extract ZIP file',
    filename='asset_pack.zip',
    reports_dir='reports/',
    error_details={'file_size': 12345},
    exception_info=traceback.format_exc()
)
```

Convenience functions are also available:

```
from checks.html_error_reporter import (
    generate_zip_error_report,
    generate_json_error_report,
    generate_file_access_error_report,
    generate_qc_check_error_report,
    generate_network_error_report
)
```

Or via CLI:

```
python -m checks.html_error_reporter zip_extraction corrupted_file.zip reports/
```

Output Filename

```
{input_filename}_{ISO_timestamp}_ERROR.html
```

Error Types

| Error Type | Trigger | User Fix Instructions |
|-----------------|---|--|
| zip_extraction | ZIP file is corrupted or can't be extracted | Re-create ZIP, verify compression, test locally |
| json_parsing | linkingrecord.json has syntax errors | Use a JSON validator, check for common syntax issues |
| file_access | File permission or existence issues | Verify file permissions and existence |
| qc_profile | QC profile configuration error | Contact support (system configuration issue) |
| check_execution | A QC check raised an unhandled exception | Review asset pack contents, validate file structure |
| network_upload | Box API or network connectivity failure | Check connectivity, verify upload service |
| unknown | Any unexpected error | Try again, verify ZIP integrity, contact support |

Error Report Layout

The report includes:

- 1. **Red gradient header** with error icon
- 2. **Alert banner** with the error message
- 3. **Two-column body**: - Left: "How to Fix This Issue" — error-type-specific step-by-step instructions - Right: "Re-upload Instructions" — guidance on resubmitting
- 4. **Technical Details** accordion (collapsed) — error type badge, file info, timestamp, exception traceback
- 5. **"Need Help?" footer** — support guidance

Fallback

If HTML generation itself fails, the reporter falls back to a plain text error report (.txt extension) with basic error details.

Report Destinations

| Mode | QC Reports | Error Reports |
|---------------|---|---|
| CLI | <code>reports/YYYYMMDD_HHMMSS/qc_results.json</code> (JSON only by default) | <code>reports/</code> directory |
| Box Hotfolder | Uploaded to <code>BOX_REPORT_FOLDER_ID</code> (both JSON and HTML) | Uploaded to <code>BOX_REPORT_FOLDER_ID</code> |

In the Box hotfolder flow, local report files are cleaned up after successful upload.

Deployment & Operations

Local Development Setup

Prerequisites

- Python 3.8+
- Access to test asset packs (in `example_packs/` directory, ~37GB)

Installation

```
# Clone the repository
git clone <repo-url>
cd Ford_BnP_QC

# Create and activate virtual environment
python3 -m venv venv
source venv/bin/activate

# Install dependencies
pip install -r requirements.txt
```

Running QC Locally (CLI)

```
# Run all checks against an asset pack
python qc_engine.py profiles/ford_bnp.json --input_file "example_packs/some_pack.zip"
--reports_dir reports

# Generate HTML report from JSON results
python -m checks.html_reporter reports/YYYYMMDD_HHMMSS/qc_results.json reports/

# Generate a test error report
python -m checks.html_error_reporter zip_extraction test_file.zip reports/
```

Testing Individual Checks

```
# Run a single check directly
python -c "from checks.image_format_check import run_check; print(run_check({'working_dir':
'working'}))"

# Run unit tests
python tests/test_file_size_check.py
python tests/test_image_format_check.py
python tests/test_colour_existence_check.py
python tests/test_lifestyle_inventory_check.py
```

Creating a Test Profile

To test a subset of checks, create a custom profile JSON with only the checks you need:

```
[
  {
    "script": "checks.image_resolution_check",
    "config": {
      "working_dir": "__WORKING_DIR__",
```

```
        "linkingrecord_filename": "linkingrecord.json"
    }
}
]
```

Then run: `python qc_engine.py my_test_profile.json --input_file pack.zip`

Environment Configuration

Configuration Files

| File | Purpose | Git-tracked |
|---------------------------|--|-----------------|
| <code>.env.example</code> | Template with all variables and defaults | Yes |
| <code>.env.prod</code> | Production configuration | No (gitignored) |
| <code>.env.dev</code> | Development configuration | No (gitignored) |
| <code>.env</code> | Base defaults (optional) | No (gitignored) |

Setting Up a New Environment

```
# Copy the template
cp .env.example .env.prod    # for production
cp .env.example .env.dev     # for development

# Edit with your environment-specific values
vim .env.prod
```

Key Variables to Configure

| Variable | What to Change | Notes |
|--------------------------------------|--|--|
| <code>BOX_SOURCE_FOLDER_ID</code> | Box folder ID for incoming files | Different per environment |
| <code>BOX_REPORT_FOLDER_ID</code> | Box folder ID for reports | Different per environment |
| <code>BOX_PROCESSED_FOLDER_ID</code> | Box folder ID for processed files | Different per environment |
| <code>BOX_CONFIG_FILE</code> | JWT config filename | e.g., <code>ford_box_config_prod.json</code> |
| <code>LOG_LEVEL</code> | <code>DEBUG</code> for dev, <code>INFO</code> for prod | Controls verbosity |

See [Architecture > Configuration System](#) for full variable reference.

Box JWT Configuration

Each environment needs its own Box JWT configuration file for authentication.

Setting Up Box JWT

1. Create a Box application in the Box Developer Console

- 2. Configure JWT authentication
- 3. Download the JSON configuration file
- 4. Place it in the project root: - Production: `ford_box_config_prod.json` - Development: `ford_box_config_dev.json`
- 5. Set the filename in your `.env` file: `` `BOX_CONFIG_FILE=ford_box_config_prod.json` ``

The JWT config file contains sensitive credentials (client ID, client secret, private key) — these files are gitignored.

Production Deployment

Service Architecture

The system runs as a systemd service under the `box-cli` user. Production and development can run simultaneously as separate services.

```
/home/box-cli/FORD_SCRIPTS/  
|-- ford_qc_git_prod/ford_qc/      # Production deployment  
|   |-- .env.prod  
|   |-- ford_box_config_prod.json  
|   `-- venv/  
`-- ford_qc_git_dev/ford_qc/      # Development deployment  
    |-- .env.dev  
    |-- ford_box_config_dev.json  
    `-- venv/
```

Installing the Service

```
# Copy the appropriate service file  
sudo cp ford-qc-hotfolder-prod.service /etc/systemd/system/  
  
# Reload systemd to pick up the new service  
sudo systemctl daemon-reload  
  
# Enable auto-start on boot  
sudo systemctl enable ford-qc-hotfolder-prod.service  
  
# Start the service  
sudo systemctl start ford-qc-hotfolder-prod.service
```

Service File Breakdown

The systemd service file (`ford-qc-hotfolder-prod.service`) configures:

| Setting | Value | Purpose |
|-------------------------------|--|--|
| <code>Type=notify</code> | — | Service sends watchdog notifications |
| <code>WorkingDirectory</code> | <code>/home/box-cli/FORD_SCRIPTS/ford_qc_git_prod/ford_qc</code> | Base path |
| <code>FORD_QC_ENV</code> | <code>production</code> | Selects <code>.env.prod</code> config |
| <code>EnvironmentFile</code> | <code>.env.prod</code> | Loads environment variables |
| <code>Restart=always</code> | <code>RestartSec=30</code> | Auto-restart on failure with 30s delay |

| Setting | Value | Purpose |
|-------------------|---------------------------|------------------------------------|
| WatchdogSec=3600 | — | Restart if no heartbeat for 1 hour |
| StartLimitBurst=5 | StartLimitIntervalSec=600 | Max 5 restarts per 10 minutes |
| TimeoutStopSec=30 | — | 30 seconds for graceful shutdown |
| KillMode=mixed | — | SIGTERM first, then SIGKILL |
| User | box-cli | Runs as this system user |

Service Management

Common Commands

```
# Check service status
sudo systemctl status ford-qc-hotfolder-prod.service

# Start / stop / restart
sudo systemctl start ford-qc-hotfolder-prod.service
sudo systemctl stop ford-qc-hotfolder-prod.service
sudo systemctl restart ford-qc-hotfolder-prod.service

# Enable / disable auto-start
sudo systemctl enable ford-qc-hotfolder-prod.service
sudo systemctl disable ford-qc-hotfolder-prod.service

# After editing the .service file
sudo systemctl daemon-reload
```

Viewing Logs

```
# Follow logs in real-time
sudo journalctl -u ford-qc-hotfolder-prod.service -f

# View recent logs
sudo journalctl -u ford-qc-hotfolder-prod.service --since "1 hour ago"

# View logs since last boot
sudo journalctl -u ford-qc-hotfolder-prod.service -b

# Verify environment configuration was loaded
sudo journalctl -u ford-qc-hotfolder-prod.service | grep "Environment:"
```

Running Manually (for debugging)

```
# Switch to service user
sudo su - box-cli

# Navigate to deployment
cd /home/box-cli/FORD_SCRIPTS/ford_qc_git_prod/ford_qc

# Activate venv and set environment
source venv/bin/activate
export FORD_QC_ENV=production

# Run manually
python ford_qc_box_hotfolder_process.py
```


Multi-Environment Operation

Production and development environments can run simultaneously because they use:

- Separate git clones (`ford_qc_git_prod/` vs `ford_qc_git_dev/`)
- Separate `.env` files (`.env.prod` vs `.env.dev`)
- Separate Box JWT configs (`ford_box_config_prod.json` vs `ford_box_config_dev.json`)
- Separate Box folder IDs (different source/report/processed folders)
- Separate systemd services (`ford-qc-hotfolder-prod` vs `ford-qc-hotfolder-dev`)
- Separate systemd log identifiers (filterable independently)

```
# Manage both independently
sudo systemctl status ford-qc-hotfolder-prod.service
sudo systemctl status ford-qc-hotfolder-dev.service
```

Troubleshooting

Service Won't Start

1. **Check logs:** `sudo journalctl -u ford-qc-hotfolder-prod.service -n 50`
2. **Verify Python path:** Ensure the venv path in the `.service` file is correct
3. **Check permissions:** Files must be owned by / readable by `box-cli`
4. **Verify `.env` file:** Ensure all required variables are set (see `.env.example`)
5. **Test manually:** Run the script directly to see error output

Box Authentication Failures

1. **Check JWT config:** Verify `ford_box_config_prod.json` exists and is valid JSON
2. **Check file path:** Verify `BOX_CONFIG_FILE` in `.env` points to the correct file
3. **Verify credentials:** JWT keys may have expired — regenerate in Box Developer Console
4. **Check logs for details:** Auth errors are logged with config details (clientId, publicKeyID)

QC Checks Failing Unexpectedly

1. **Run locally:** Test the same ZIP via CLI to see detailed output
2. **Check working directory:** Ensure `working/` is writable and has sufficient disk space
3. **Check for new asset types:** Unknown viewtype/imagetype combinations are skipped — check if new types need to be added to check rules

Service Stuck / Not Processing

1. **Check watchdog:** If no watchdog notification for 1 hour, systemd auto-restarts
2. **Check disk space:** Large ZIPs can fill `download_tmp/` or `working/`
3. **Check Box API:** Network issues may cause retries to exhaust
4. **Force restart:** `sudo systemctl restart ford-qc-hotfolder-prod.service`

Developer Guide: Extending the System

The Check Module Contract

Every QC check module must follow this contract:

Required Interface

```
def run_check(config: dict) -> dict:
```

Input: A `config` dictionary containing at minimum:

| Key | Type | Description |
|-------------------------------------|------|--|
| <code>working_dir</code> | str | Path to the directory with extracted asset pack files |
| <code>linkingrecord_filename</code> | str | Name of the linking record file (usually <code>"linkingrecord.json"</code>) |
| <code>input_file</code> | str | Path to the original ZIP file (injected by the engine) |

Output: A dictionary with at minimum a `status` field:

```
{
  "status": "passed",      # or "failed", "error", "skipped"
  "details": {
    "message": "Human-readable summary of the result",
    # ... check-specific detail fields
  }
}
```

Status Values

| Status | Meaning | When to Use |
|------------------------|---------------------------|--|
| <code>"passed"</code> | All validations succeeded | No issues found |
| <code>"failed"</code> | Validation found issues | One or more rules violated |
| <code>"error"</code> | Check couldn't complete | Missing files, invalid JSON, unexpected exceptions |
| <code>"skipped"</code> | Check was not applicable | E.g., MEC-only check on a BAU pack |

Error Result Format

When returning an error status, include `error_message` at the top level:

```
{
  "status": "error",
  "error_message": "Linking record 'linkingrecord.json' not found in working/",
  "details": {
    "message": "Same error message for report rendering"
  }
}
```

```
}
```

Adding a New Check: Step by Step

1. Create the Module

Create a new file in `checks/`, e.g., `checks/my_new_check.py`:

```
import os
import json
from typing import Dict, Any

def run_check(config: Dict[str, Any]) -> Dict[str, Any]:
    working_dir = config.get("working_dir", "working")
    linkingrecord_filename = config.get("linkingrecord_filename", "linkingrecord.json")
    linkingrecord_path = os.path.join(working_dir, linkingrecord_filename)

    # Validate prerequisites
    if not os.path.exists(linkingrecord_path):
        return {
            "status": "error",
            "error_message": f"Linking record not found in {working_dir}."
        }

    # Load linking record
    with open(linkingrecord_path, 'r', encoding='utf-8') as f:
        linkingrecord = json.load(f)

    # Your validation logic here
    issues = []
    for item in linkingrecord.get("items", []):
        conditions = item.get("conditions", {})
        # ... validate each item

    # Return result
    if issues:
        return {
            "status": "failed",
            "details": {
                "message": f"Found {len(issues)} issues.",
                "issues": issues
            }
        }

    return {
        "status": "passed",
        "details": {"message": "All validations passed."}
    }
```

2. Add to the QC Profile

Edit `profiles/ford_bnp.json` and add your check entry. Position matters — checks run in order:

```
{
  "script": "checks.my_new_check",
  "config": {
    "working_dir": "__WORKING_DIR__",
    "linkingrecord_filename": "linkingrecord.json"
  }
}
```

3. Test the Check

```
# Quick test with an extracted working directory
python -c "from checks.my_new_check import run_check; print(run_check({'working_dir':
'working'}))"

# Full test with a ZIP file
python qc_engine.py profiles/ford_bnp.json --input_file "example_packs/some_pack.zip"
```

4. Add HTML Report Rendering (if needed)

If your check returns detail fields that need special rendering in the HTML report, update `checks/html_reporter.py`:

1. Add a friendly name in `_get_friendly_check_name()`:

```
python friendly_names = { # ... existing
entries "my_new_check": "My New Validation", }
```
2. Add rendering logic in `_format_details()` for any custom detail fields your check returns.

Using Check Helpers

The `utils/check_helpers.py` module provides utilities for handling unknown viewtype/imagetype combinations gracefully. Use these when your check has a fixed set of rules and may encounter new types it doesn't know about.

Recording Skipped Types

```
from utils.check_helpers import record_skipped_type, prepare_skipped_result

def run_check(config):
    # Initialize skipped types tracker
    skipped_types = {"my_check": set()}

    for item in linkingrecord["items"]:
        viewtype = conditions.get("viewtype")
        imagetype = conditions.get("imagetype")

        # Try to find a rule for this type
        rule = my_rules.get((viewtype, imagetype))
        if rule is None:
            # Unknown type - skip it, don't fail
            record_skipped_type(skipped_types, "my_check", viewtype, imagetype)
            continue

        # ... validate with rule

    # If only skipped types were found (no failures), return a clean pass
    if no_failures and skipped_types["my_check"]:
        return prepare_skipped_result(skipped_types, "my_check")
```

This pattern ensures:

- New viewtype/imagetype combinations don't cause false failures
- Skipped combinations are visible in the report for investigation
- The check still passes when all known types validate correctly

Common Patterns

Pack Type Detection

Many checks behave differently for MEC vs BAU packs. Use this pattern:

```
is_mec = any(
    item.get("conditions", {}).get("experienceCondition") == "2d-background"
    for item in linkingrecord["items"]
)
pack_type = "MEC" if is_mec else "BAU"
```

Deduplication by Filename

Multiple records can reference the same file. To avoid reporting the same file multiple times:

```
failures_dict = {} # keyed by filename

for item in linkingrecord["items"]:
    for record in item.get("records", []):
        for asset in record.get("assets", []):
            filename = asset.get("filename")
            if filename and filename not in failures_dict:
                # ... check and potentially add to failures_dict
                failures_dict[filename] = {"filename": filename, ...}

# Convert to list for result
failed_list = list(failures_dict.values())
```

Iterating the Linking Record

The standard iteration pattern:

```
for item in linkingrecord.get("items", []):
    conditions = item.get("conditions", {})
    viewtype = conditions.get("viewtype")
    imagetype = conditions.get("imagetype")
    experience = conditions.get("experienceCondition")
    variant = conditions.get("variant")

    for record in item.get("records", []):
        angle = record.get("angle")
        features = record.get("features", [])

        for asset in record.get("assets", []):
            filename = asset.get("filename")
            layer = asset.get("layer", 0)
            file_path = os.path.join(working_dir, filename)
```

Writing Tests

Tests are located in `tests/` and follow a straightforward pattern using temporary directories and test fixtures.

Test Structure

```
import os
import sys
import json
import tempfile

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
```

```
from checks.my_new_check import run_check

def test_passing_case():
    with tempfile.TemporaryDirectory() as temp_dir:
        # Create test files
        os.makedirs(os.path.join(temp_dir, "subdir"), exist_ok=True)

        # Create a minimal linkingrecord.json
        linkingrecord = {
            "items": [
                {
                    "conditions": {"viewtype": "exterior"},
                    "records": [
                        {
                            "angle": 21,
                            "assets": [{"filename": "subdir/test.jpg", "layer": 0}]
                        }
                    ]
                }
            ]
        }

        with open(os.path.join(temp_dir, "linkingrecord.json"), 'w') as f:
            json.dump(linkingrecord, f)

        # Create the test image file
        # For resolution/format checks, use PIL:
        # from PIL import Image
        # img = Image.new('RGB', (1600, 900))
        # img.save(os.path.join(temp_dir, "subdir/test.jpg"), "JPEG")

        # For size checks, create a file of specific size:
        # with open(path, 'wb') as f:
        #     f.seek(size_bytes - 1)
        #     f.write(b'\0')

        # Run the check
        config = {
            "working_dir": temp_dir,
            "linkingrecord_filename": "linkingrecord.json"
        }
        result = run_check(config)

        assert result["status"] == "passed", f"Expected passed, got: {result}"

if __name__ == "__main__":
    test_passing_case()
    print("All tests passed!")
```

Testing Tips

- Create minimal linkingrecord fixtures — only include the items your check cares about
- Use `PIL.Image.new()` to create test images with specific dimensions and formats
- Use `f.seek(size_bytes - 1); f.write(b'\0')` to create files of exact sizes without writing real data
- Test both passing and failing cases
- Test edge cases: empty items, missing fields, unknown viewtype/imagetype combinations

Common Pitfalls

File Path Handling

- Filenames in `linkingrecord.json` are relative to the working directory (e.g., `ext/base/image.jpg`)
- Always join with `working_dir`: `os.path.join(working_dir, filename)`
- Normalize paths when comparing filesystem files with JSON references

AVIF Case Sensitivity

- AVIF extensions must be uppercase `.AVIF` — this is checked by multiple modules
- If your check handles AVIF files, preserve case sensitivity

Check Ordering

- Your check likely depends on Check 1 (`unzip_and_verify`) having already run
- Place new checks after the foundational checks (1-3) in the profile
- If your check depends on data from another check, it must come after that check in the profile

Error Handling

- Always handle missing `linkingrecord.json` gracefully (return error status)
- Handle JSON parse errors (return error status)
- Skip missing image files silently — the missing images check (Check 3) handles that
- Wrap PIL/image operations in `try/except` — corrupted images should be skipped, not crash the check

The HTML Reporter

- If you add new detail field names, the reporter won't render them until you add rendering logic
- Unrecognized fields are displayed as raw JSON in the report — functional but not pretty
- The reporter has specific renderers for common field names like `failed_images`, `missing_files`, etc.