BCR & cBCR Modeler — Full Context + Full Code (Dual Spec)

This document intentionally contains **two complete, complementary representations** of the same modeler: 1) A **language-only specification** (spec, SOP, and guardrails) so an operator or AI can run the bootstrap with **zero assumptions**. 2) The **full Python implementation** that follows the spec verbatim and defaults to **PDF-only** outputs.

PART I — Full Context (Spec, SOP, Guardrails)

> Consolidates the entire operator-proof specification (including the Zero-Assumption Playbook) so any AI can execute without guessing.

0) Purpose & Scope

- **Goal:** From baryonic structure alone, predict kinematics (galaxies) or mass/acceleration (clusters) via the **Baryon Crowding Relation** (BCR, cBCR), compare to references/observations, and produce publication-ready PDFs.
- **Domains:**
- **Galaxies (SPARC-like):** baryon enclosed mass $M_{\rm bar}(r)$ → predict $v_{\rm pred}(r)$. Optionally compare to $v_{\rm obs}(r)$ and compute DEX/bias; overlay RAR.
- **Clusters (X-COP-like):** gas/baryon profiles + hydrostatic mass → predict cBCR curves; compare to hydro mass and, when available, observed kinematics ($v_{\rm circ}$ or dispersion σ with a chosen mapping).
- **Deliverables:** per-object PDF pages; global PDF summaries; optional master-curve PDF; optional DEX pack PDF.

1) Physics Kernel & Defaults (do not change unless explicitly instructed)

- **Kernel:** *Kernel V6* (exponent morph, interior–exterior continuous mapping).
- **Soft pivot:** $x_0 = 0.28$.
- **Characteristic accelerations:**
- **Galaxies:** $g^\dagger_{\rm gal} = 8.6 \times 10^{-11}\,\mathrm{m\,s^{-2}}$ *(fixed)*.
- **Clusters:** $g^\dagger_{\rm cl} = 1.771 \times 10^{-9}\,\mathrm{m\,s^{-2}}$ *(default)*, with optional **cluster-only** scale factor s such that $g^\dagger_{\rm cl} = s\, g^\dagger_{\rm gal}$. *Galaxies remain unchanged.*
- **RAR overlay (galaxies):** $a_0 = 1.2 \times 10^{-10}\,\mathrm{m\,s^{-2}}$ using the continuous McGaugh form (overlay only; does not affect BCR).

Kernel V6 mapping: $x = \bar{g}/g^\dagger$, $\chi(x) = x/(x + x_0)$, $\varepsilon = x_0/(1 + x_0)$, $p(x) = \frac{1}{2} + \left(\frac{1}{2} - \varepsilon\right)\chi(x)$, $g = x^{p(x)} g^\dagger$. Velocities: $v_{\rm pred} = \sqrt{g\,r}$, $v_{\rm} = \sqrt{\bar{g}\,r}$.

2) Inputs (strict schemas)

Galaxies — Required: `galaxy, r_kpc, Mbar_enc_Msun`. Optional: `v_obs_kms`, `v_err_kms`.

Clusters — Gas/Baryon: either `RADIUS` (Mpc) + `MGAS` (M⊙), or `RADIUS` (Mpc) + `FGAS` (0–1) paired with hydro mass. Hydrostatic mass: `R_KPC` (kpc) **or** `RADIUS` (guarded), plus one of `{M_FORW, M_NFW, M_EIN, M_ISO, M_BUR, M_HSE, M_TOT, MTOTAL, M_TOTAL}`. Optional observed kinematics: `r_kpc` + `v_obs_kms` or `sigma_kms` (with configurable mapping, default $\sqrt{2}$).

3) Units & Numeric Guards (mandatory)

- **Galaxies:** radii **kpc**; **Clusters:** gas `RADIUS` **Mpc**, hydro `R_KPC` **kpc** (or guarded `RADIUS`). Compute in SI internally; plot axes per spec.
- **Hydro radius guard:** If `RADIUS` max ∈ [30, 5000] → **kpc**; if ≤20 → **Mpc**; else fail `UNIT_ASSERT_HYDRO_RADIUS`.
- **Interpolation:** sort radii; require ≥3 points; **no extrapolation** beyond baryon grid; mask out-of-range obs.

4) Pipelines

> **RUN ORDER HARD GUARDRAIL** - **Stage 1 — BLIND (mandatory):** Start immediately using **enclosed mass only**. Compute BCR predictions and render pages (**BCR pred + RAR overlay + baryon baseline**). **Do not** wait for any observed velocities to begin. - **Stage 2 — Observed overlay (optional):** After Stage 1 completes, if `v_obs_kms` is provided, overlay observations on the same plots and compute **median |DEX|** and **bias**. Keep PDF-only policy.

Galaxies — Compute $\bar{g}$, map via Kernel V6 with $g^{\dagger}_{\rm gal}$; derive `v_bar`, `v_pred`. If `v_obs_kms` present, compute DEX/bias; plot one page per galaxy with BCR, RAR overlay, baryon baseline, and obs.

Clusters (blind by default) — Pair gas (Mpc) and hydro (kpc/guarded). Build baryon mass (prefer `MGAS`; else `FGAS×M_hydro`). Compute $\bar{g}$ on gas grid; map via $g^{\dagger}_{\rm cl}$ (or tuned s). Predict $M_{\rm pred}$ and compare to hydro mass on the same grid. Plot two pages per cluster: (1) enclosed mass panel (log y) and (2) g vs ḡ mapping.

Master curve (optional) — Build point clouds in $(x, y) = (\bar{g}/g^{\dagger}, g/g^{\dagger})$ and overlay reference mapping.

5) Metrics & Reports (PDF-embedded)

- **Per galaxy:** points, median |DEX| (BCR), signed bias (BCR).
- **DEX Pack (MANDATORY when observations are present):**
- **Headline DEX score page** — sample size, dwarf/non-dwarf counts (dwarf = vmax ≤ 50 km/s), and **headline median RMS-dex (acceleration)** for: **Baryon-only**, **RAR**, **BCR**.
- **Accumulated DEX chart (CDF)** — CDF of per-galaxy **RMS-dex** for the three models.
- **Histogram** — per-galaxy **RMS-dex** distribution for the three models.

- **Single output file:** `BCR_full_dex_report.pdf` containing **three pages in this exact order**: (1) Headline DEX score page, (2) CDF chart, (3) Histogram. **No PNGs** or other artifacts unless `--export-csv` is explicitly set.

- Per object: point count, low-x slope (target ~0.5), RMS-dex.

- Global (galaxies): mean of per-galaxy median DEX (BCR, RAR, baryon), signed bias.
- Global (clusters): RMS-dex vs hydro; when obs present, DEX + signed bias vs kinematics.

- **DEX pack policy:** When observed velocities are present, building `BCR_full_dex_report.pdf` is **mandatory**. Produce **PDF pages only**; **no PNGs** or CSV unless `--export-csv` is explicitly set.

6) Cluster-Only Tuning (galaxies fixed)

- Keep $g^{\dagger}_{\rm gal}$ fixed. Define $g^{\dagger}_{\rm cl} = s\, g^{\dagger}_{\rm gal}$.
- **Option A (bias target):** choose s with $\langle \log_{10}(v_{\rm model}/v_{\rm obs}) \rangle \approx 0$.
- **Option B (DEX target):** choose s minimizing median DEX across clusters.
- Use **baryon-driven** $\bar{g}$; if obs are dispersions, apply configured σ→$v_{\rm circ}$ mapping (default $\sqrt{2}$).

7) Charting Protocols (strict)

- **Galaxies:** X `r (kpc)`; Y `Velocity (km s⁻¹)`; BCR solid 1.8pt; RAR dashed 1.2pt; baryon dotted 1.2pt.
- **Observed markers (high contrast):**
- Marker `o`, size 4, face `#111111` (near-black), edge `white`, edge-width 0.4, alpha 0.95, legend label `Observed`.
- If error bars: `ecolor='#404040'`, `elinewidth=0.8`, `capsize=2`.
- Axis limits from **prediction curves only** (AXIS_ISOLATION); X pad 4%, Y pad 8%.

8) Sanity Checks & Failure Codes

- **Interior Response Guarantee:** allow $v_{\rm pred} < v$ at high x; never clamp above v. → `INTRESP_FAIL` if violated.
- **Hydro Radius Guard:** see §3. → `UNIT_ASSERT_HYDRO_RADIUS` if ambiguous.
- **Overlap-only plotting** for obs.
- Failure codes include: `SCHEMA_FAIL_*`, `MODEL_TIER_TOO_LOW`, `MODEL_SELFTEST_FAIL`, `MODEL_POLICY_VIOLATION`.

9) Batching, Resume & Environment

- **Batches:** galaxies in 5 batches (~35/pages each for 175), clusters in sensible batches; maintain `manifest.json` with indices/status.
- **Resume:** `--resume batch=k` continues from last complete page; no duplicates.
- **Environment guard:** run a 2-page self-test to auto-select profile `{fast|standard|high}`.

10) Model / Runtime Auto-Selector Guardrails (o3-class or better)

- **Tier A (default requirement):** time/page ≤ 5s; stable PDF; ≥2 GB free RAM; Python ≥3.10; Matplotlib ≥3.7.
- **Tier B (constrained):** time/page ≤ 12s; ≥1 GB RAM; auto-reduce DPI/batch; PNG staging allowed.
- **Protocol:** pilot → decide → lock; or abort with `MODEL_TIER_TOO_LOW`.

11) Optional Micro-Bias (OFF by default)

- Disk-thickness tweak with β, h_z only if data supplied; otherwise skip and log.

12) Outputs (PDF-only by default)

- **Galaxies:** `BCR_galaxies_plots.pdf`

- **Clusters:** `cBCR_clusters_plots.pdf`
- **Master-curve (optional):** `BCR_cBCR_master_curve.pdf`
- **DEX Pack (optional):** `BCR_full_dex_report.pdf`
- **Strict rule:** No CSV unless explicitly requested (e.g., `--export-csv`).

13) Operator Checklist (boot)

- Confirm units (gal kpc; gas Mpc; hydro kpc/guard). ≥3 points per curve. Monotone mass (cummax) ON. No extrapolation; overlap only. Baryon-driven cBCR. Cluster tuning adjusts **only** s. PDF-only policy enforced.

14) Roadmap for Code Pass

- A single script implements all of the above; embedded metadata on PDF title pages; deterministic outputs.

APPENDIX — Zero-Assumption Execution Playbook (Micromanaged)

A1) Canonical Units, **A2) File Names & Matching**, **A3) Column Detection**, **A4) Data Hygiene**, **A5) Kernel & Mapping**, **A6) Observations & Conversions**, **A7) Plot Templates (fonts, DPI, line styles, axis padding, footer text)**, **A8) Batching & Resume (manifest schema)**, **A9) Runtime Self-Test gates**, **A10) Failure Codes**, **A11) Cluster-Only Tuning Protocol**, **A12) PDF-Only Policy** — all exactly as previously specified.

PART II — Full Code (Python, mirrors the spec)

```
#!/usr/bin/env python3
"""
BCR & cBCR Bootstrap Modeler — PDF-only by default
Implements the canvas spec (Zero-Assumption Execution Playbook) end-to-end.

Usage (examples):
  python bcr_cbcr_modeler.py
    --galaxy-baryon sparc_radii_enclosed_baryonic_mass.xlsx
    --galaxy-obs sparc_observed_velocity_profiles.csv
    --cluster-hydro-dir /path/to/
    --cluster-gas-dir /path/to/
    --out-gal-pdf BCR_galaxies_plots.pdf
    --out-clu-pdf cBCR_clusters_plots.pdf
    --require-tier A

  # Optional extras (only when you want them):
  --export-csv  (enables CSV outputs)
  --make-master-curve  --cluster-obs cluster_observations.csv  (enables
cluster scoring/tuning)
  --tune-clusters  (find s_A* and s_B*)
```

```
Strict defaults: PDF-only. No CSV unless --export-csv.
"""

from __future__ import annotations
import argparse, os, re, sys, time, json, math, textwrap, io
from dataclasses import dataclass
from typing import Dict, List, Tuple, Optional

import numpy as np
import pandas as pd
import matplotlib as mpl
mpl.use("Agg")
from matplotlib import pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

# ==============================
# A1) Constants & Conversions
# ==============================
G_SI = 6.67430e-11            # m^3 kg^-1 s^-2
KPC_TO_M = 3.085677581491367e19
MSUN_TO_KG = 1.98847e30

# ==============================
# Kernel defaults (V6)
# ==============================
X0 = 0.28
EPS = X0 / (1.0 + X0)
GDAG_GAL = 8.6e-11            # m s^-2 (fixed)
GDAG_CLU = 1.771e-9           # m s^-2 (default; cluster-only tuning may
scale this)
A0_RAR   = 1.2e-10            # m s^-2 (RAR overlay)

# ==============================
# Plot style (A7)
# ==============================
mpl.rcParams.update({
    "pdf.compression": 6,
    "pdf.fonttype": 42,
    "font.size": 10,
})

# ==============================
# Utilities
# ==============================

def _now_utc_iso():
    return time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())


def log(msg: str):
```

```
    sys.stderr.write(msg + "\n")
    sys.stderr.flush()


def ensure_dir(path: str):
    d = os.path.dirname(os.path.abspath(path))
    if d and not os.path.exists(d):
        os.makedirs(d, exist_ok=True)


# ==============================
# Guards & Schema helpers (A2-A4)
# ==============================

def assert_columns(df: pd.DataFrame, required: List[str], domain_code: str):
    missing = [c for c in required if c not in df.columns]
    if missing:
        raise RuntimeError(f"SCHEMA_FAIL_{domain_code}: missing columns
{missing}; have {list(df.columns)}")


def hydro_radius_guard(series: pd.Series) -> str:
    """Return 'kpc' or 'Mpc' for ambiguous RADIUS column per A1 rule, else
raise."""
    try:
        vmax = float(np.nanmax(pd.to_numeric(series, errors='coerce')))
    except Exception:
        vmax = np.nan
    if np.isnan(vmax):
        raise RuntimeError("UNIT_ASSERT_HYDRO_RADIUS: cannot read RADIUS
values")
    if 30 <= vmax <= 5000:
        return "kpc"
    if vmax <= 20:
        return "Mpc"
    raise RuntimeError(f"UNIT_ASSERT_HYDRO_RADIUS: ambiguous RADIUS scale
(max={vmax})")


def enforce_monotone(series: pd.Series) -> pd.Series:
    return series.cummax()


# ==============================
# Kernel mappings (A5)
# ==============================

def mapping_y_from_gbar(gbar: np.ndarray, gdag: float) -> np.ndarray:
    x = np.clip(gbar / gdag, 1e-300, None)
    chi = x / (x + X0)
    p = 0.5 + (0.5 - EPS) * chi
```

```
    return (x ** p) * gdag


def rar_from_gbar(gbar: np.ndarray) -> np.ndarray:
    gb = np.clip(gbar, 1e-30, None)
    return gb / (1.0 - np.exp(-np.sqrt(gb / A0_RAR)))


# ============================
# Pilot self-test & tiering (A9)
# ============================
@dataclass
class ProfileDecision:
    tier: str
    profile: str
    dpi: int
    batch_pages: int


def pilot_selftest(dpi_high: int = 120) -> ProfileDecision:
    start = time.time()
    # Render 2 light test pages in-memory
    with PdfPages(io.BytesIO()) as pdf:
        for i in range(2):
            fig, ax = plt.subplots()
            r = np.linspace(0.1, 10, 200)
            v = 200 * np.tanh(r/5) + 50
            ax.plot(r, v)
            ax.set_xlabel("r (kpc)"); ax.set_ylabel("Velocity (km s⁻¹)")
            ax.set_title("Self-test page")
            pdf.savefig(fig, dpi=dpi_high)
            plt.close(fig)
    t = (time.time() - start) / 2.0
    # We cannot truly probe RAM/IO; approximate with time/page gate per spec.
    if t <= 5.0:
        return ProfileDecision(tier="A", profile="high", dpi=dpi_high,
batch_pages=35)
    if t <= 12.0:
        return ProfileDecision(tier="B", profile="fast", dpi=110,
batch_pages=25)
    raise RuntimeError("MODEL_TIER_TOO_LOW: self-test time/page exceeds 12s")


# ============================
# Galaxy pipeline (4.1)
# ============================

def load_galaxy_baryon_xlsx(path: str) -> pd.DataFrame:
    # Expect either a single sheet with canonical columns or multi-sheets;
fallback to heuristic.
    xl = pd.ExcelFile(path)
```

```
    frames = []
    for sh in xl.sheet_names:
        df = xl.parse(sh).dropna(axis=1, how='all')
        cols = [c.strip() for c in df.columns]
        df.columns = cols
        # Canonical: galaxy, r_kpc, Mbar_enc_Msun
        lower = {c.lower(): c for c in cols}
        gal = lower.get('galaxy') or lower.get('name') or lower.get('object')
or None
        r = 'r_kpc' if 'r_kpc' in cols else None
        m = 'Mbar_enc_Msun' if 'Mbar_enc_Msun' in cols else None
        if r is None:
            # try to discover a radius
            for c in cols:
                if re.search(r'^r(_kpc)?$', c, flags=re.I):
                    r = c; break
        if m is None:
            for c in cols:
                if re.search(r'mbar|m_bar|m\s*enc|enclosed.*mass|
Mbar_enc_Msun', c, flags=re.I):
                    m = c; break
        if r and m:
            if gal is None:
                df['_galaxy'] = sh
                gal = '_galaxy'
            sub = df[[gal, r, m]].copy()
            sub.columns = ['galaxy','r_kpc','Mbar_enc_Msun']
            frames.append(sub)
    if not frames:
        raise RuntimeError("SCHEMA_FAIL_GALAXY: no valid sheets/columns
found")
    out = pd.concat(frames, ignore_index=True)
    # Clean
    out['r_kpc'] = pd.to_numeric(out['r_kpc'], errors='coerce')
    out['Mbar_enc_Msun'] = pd.to_numeric(out['Mbar_enc_Msun'],
errors='coerce')
    out = out.dropna(subset=['r_kpc','Mbar_enc_Msun'])
    return out


def load_galaxy_obs_csv(path: str) -> pd.DataFrame:
    df = pd.read_csv(path)
    cols = [c.strip() for c in df.columns]
    df.columns = cols
    # Expected: galaxy, r_kpc, v_obs_kms [, v_err_kms]
    # Attempt mapping if variants present
    colmap = {}
    for c in cols:
        lc = c.lower()
        if lc in ('galaxy','name','object','id') and 'galaxy' not in colmap:
            colmap['galaxy'] = c
```

```
        if lc == 'r_kpc' or lc == 'r':
            colmap['r_kpc'] = c
        if lc in ('v_obs_kms','v_rot','vrot','velocity','vel'):
            colmap['v_obs_kms'] = c
        if lc in ('v_err_kms','verr','e_v','v_err'):
            colmap['v_err_kms'] = c
    required = ['galaxy','r_kpc','v_obs_kms']
    for k in required:
        if k not in colmap:
            raise RuntimeError(f"SCHEMA_FAIL_GALAXY_OBS: missing {k}; have
{cols}")
    keep = [colmap['galaxy'], colmap['r_kpc'], colmap['v_obs_kms']]
    if 'v_err_kms' in colmap:
        keep.append(colmap['v_err_kms'])
    sub = df[keep].copy()
    sub.columns = ['galaxy','r_kpc','v_obs_kms'] + (['v_err_kms'] if
'v_err_kms' in colmap else [])
    sub['r_kpc'] = pd.to_numeric(sub['r_kpc'], errors='coerce')
    sub['v_obs_kms'] = pd.to_numeric(sub['v_obs_kms'], errors='coerce')
    if 'v_err_kms' in sub.columns:
        sub['v_err_kms'] = pd.to_numeric(sub['v_err_kms'], errors='coerce')
    return sub.dropna(subset=['r_kpc','v_obs_kms'])


def render_galaxies_pdf(bary_df: pd.DataFrame,
                        obs_df: Optional[pd.DataFrame],
                        out_pdf: str,
                        decision: ProfileDecision,
                        export_csv: bool=False) -> Dict:
    ensure_dir(out_pdf)
    galaxies = sorted(set(bary_df['galaxy']))
    dpi = decision.dpi

    # Prepare PDF
    meta = {
        'created_utc': _now_utc_iso(),
        'profile': decision.profile,
        'gdag_gal': GDAG_GAL,
        'x0': X0,
        'rar_a0': A0_RAR,
        'pdf_only': not export_csv,
    }

    summaries = []
    with PdfPages(out_pdf) as pdf:
        # Title page
        fig, ax = plt.subplots(figsize=(8.27, 11.69))
        ax.axis('off')
        ax.text(0.05, 0.95, 'BCR — Galaxies', fontsize=14, va='top')
        ax.text(0.05, 0.90, json.dumps(meta, indent=2), family='monospace')
        pdf.savefig(fig, dpi=dpi); plt.close(fig)
```

```
        for gname in galaxies:
            G = bary_df[bary_df['galaxy']==gname].copy().sort_values('r_kpc')
            if len(G) < 3: continue
            # Monotone enforcement
            G['Mbar_enc_Msun'] =
enforce_monotone(pd.to_numeric(G['Mbar_enc_Msun'], errors='coerce'))
            r_kpc = G['r_kpc'].values.astype(float)
            Mbar = G['Mbar_enc_Msun'].values.astype(float)
            r_m = r_kpc * KPC_TO_M
            M_kg = Mbar * MSUN_TO_KG
            v_bar = np.sqrt(np.clip(G_SI * M_kg / r_m, 0, None))
            g_bar = (v_bar**2) / r_m
            g_bcr = mapping_y_from_gbar(g_bar, GDAG_GAL)
            v_bcr = np.sqrt(np.clip(g_bcr * r_m, 0, None))
            g_rar = rar_from_gbar(g_bar)
            v_rar = np.sqrt(np.clip(g_rar * r_m, 0, None))
            # Observations (overlap only)
            have_obs = obs_df is not None and (gname in
set(obs_df['galaxy']))
            if have_obs:
                O =
obs_df[obs_df['galaxy']==gname].copy().sort_values('r_kpc')
                # mask to overlap radii
                mask = (O['r_kpc'].values >= r_kpc.min()) &
(O['r_kpc'].values <= r_kpc.max())
                O = O[mask]
            # Plot
            fig, ax = plt.subplots()
            # Axis limits from prediction curves only (AXIS_ISOLATION)
            x_min, x_max = r_kpc.min(), r_kpc.max()
            y_series = [v_bcr/1000.0, v_bar/1000.0, v_rar/1000.0]
            y_vals = np.concatenate(y_series)
            y_min, y_max = float(np.min(y_vals)), float(np.max(y_vals))
            pad_x = 0.04*(x_max-x_min) if x_max>x_min else 1.0
            pad_y = 0.08*(y_max-y_min) if y_max>y_min else 10.0
            ax.set_xlim(x_min-pad_x, x_max+pad_x)
            ax.set_ylim(max(0,y_min-pad_y), y_max+pad_y)
            ax.plot(r_kpc, v_bcr/1000.0, label="BCR (pred)", linewidth=1.8)
            ax.plot(r_kpc, v_rar/1000.0, label="RAR (overlay)",
linestyle='--', linewidth=1.2)
            ax.plot(r_kpc, v_bar/1000.0, label="Baryon baseline",
linestyle=':', linewidth=1.2)
            if have_obs:
                if 'v_err_kms' in O.columns:
                    ax.errorbar(O['r_kpc'], O['v_obs_kms'],
yerr=O['v_err_kms'], fmt='o', markersize=3, color='k')
                else:
                    ax.plot(O['r_kpc'], O['v_obs_kms'], 'o', markersize=3,
color='k')
            ax.set_xlabel('r (kpc)'); ax.set_ylabel('Velocity (km s⁻¹)')
```

```
            ax.set_title(f"{gname}: BCR vs RAR vs Baryon" + (" vs Observed"
if have_obs else ""))
            ax.legend(loc='upper left', frameon=True)
            # Footer stats
            dex_med = None; bias = None
            if have_obs and len(O):
                if 'v_err_kms' in O.columns:
                    ax.errorbar(
                        O['r_kpc'], O['v_obs_kms'], yerr=O['v_err_kms'],
                        fmt='o', markersize=4, mfc='#111111', mec='white',
mew=0.4,
                        ecolor='#404040', elinewidth=0.8, capsize=2,
alpha=0.95,
                        label='Observed'
                    )
                else:
                    ax.plot(
                        O['r_kpc'], O['v_obs_kms'],
                        'o', markersize=4, mfc='#111111', mec='white',
mew=0.4,
                        alpha=0.95, label='Observed'
                    )
            ax.legend(loc='upper left', frameon=True)
            footer = f"points={len(r_bary)}  x0={X0}  g†_cl={gdag:.3e}
profile={decision.profile}"
            ax.text(0.02, 0.02, footer, transform=ax.transAxes, fontsize=8)
            pdf.savefig(fig, dpi=dpi); plt.close(fig)

            # Page 2 — g vs gbar (log-log)
            fig, ax = plt.subplots()
            ax.loglog(gbar, gpred, '.', label='cluster bins')
            # Overlay kernel curve
            gb_grid = np.logspace(-14, -8, 300)
            ax.loglog(gb_grid, mapping_y_from_gbar(gb_grid, gdag), '-',
label='kernel')
            ax.set_xlabel('ḡ (m s⁻²)'); ax.set_ylabel('g (m s⁻²)')
            ax.set_title(f"{cf.key}: g vs ḡ (baryon-driven)")
            ax.legend()
            pdf.savefig(fig, dpi=dpi); plt.close(fig)

    return {"count": len(pairs), "gdag_cluster": gdag}


# ============================
# Cluster observations & tuning (A11)
# ============================

def load_cluster_obs_csv(path: str) -> pd.DataFrame:
    df = pd.read_csv(path)
    cols = [c.strip() for c in df.columns]
    df.columns = cols
```

```
    assert_columns(df, ['cluster','r_kpc'], 'CLUSTER_OBS')
    if 'v_obs_kms' not in cols and 'sigma_kms' not in cols:
        raise RuntimeError('SCHEMA_FAIL_CLUSTER_OBS: need v_obs_kms or
sigma_kms')
    if 'v_obs_kms' in cols and 'sigma_kms' in cols:
        # Prefer v_obs_kms, ignore sigma
        df = df[['cluster','r_kpc','v_obs_kms'] + ([c for c in ['v_err_kms']
if c in cols])]
    elif 'v_obs_kms' in cols:
        df = df[['cluster','r_kpc','v_obs_kms'] + ([c for c in ['v_err_kms']
if c in cols])]
    else:
        df = df[['cluster','r_kpc','sigma_kms'] + ([c for c in ['v_err_kms']
if c in cols])]
    df['r_kpc'] = pd.to_numeric(df['r_kpc'], errors='coerce')
    for c in ['v_obs_kms','sigma_kms','v_err_kms']:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors='coerce')
    return df.dropna(subset=['r_kpc'])


def tune_cluster_scale(pairs: List[ClusterFiles], cluster_obs: pd.DataFrame,
sigma_to_vcirc: float = math.sqrt(2.0),
                       target: str = 'bias') -> float:

"""Return best s scale for cluster gdag. target='bias' or 'dex'. Galaxies
fixed."""
    keys = set([cf.key for cf in pairs])
    grouped = {k:
cluster_obs[cluster_obs['cluster']==k].copy().sort_values('r_kpc') for k in
keys}

    def score(s: float) -> Tuple[float,float,int]:
        gdag = GDAG_GAL * s
        residuals: List[float] = []
        absdex: List[float] = []
        npts = 0
        for cf in pairs:
            rH_kpc, Mh_Msun = load_cluster_hydro(cf.hydro_path)
            rG_kpc, MGAS, FGAS = load_cluster_gas(cf.gas_path)
            # M_bary
            if MGAS is not None:
                r_bary = rG_kpc; Mbar = MGAS
            elif FGAS is not None:
                Mh_interp = np.interp(rG_kpc, rH_kpc, Mh_Msun)
                r_bary = rG_kpc; Mbar = FGAS * Mh_interp
            else:
                continue
            # Predict velocities (from g_pred)
            r_m = r_bary * KPC_TO_M
            gbar, gpred, _Mpred = cBCR_predict_mass_from_baryon(r_bary, Mbar,
```

```
gdag)
            v_pred = np.sqrt(np.clip(gpred * r_m, 0, None)) / 1000.0
            # Observations for this cluster
            O = grouped.get(cf.key)
            if O is None or len(O)==0: continue
            # Build observed v_circ
            if 'v_obs_kms' in O.columns:
                v_obs = O['v_obs_kms'].values
            else:
                v_obs = O['sigma_kms'].values * sigma_to_vcirc
            # Overlap mask
            mask = (O['r_kpc'].values >= r_bary.min()) & (O['r_kpc'].values
<= r_bary.max())
            if mask.sum() < 3: continue
            r_use = O['r_kpc'].values[mask]
            v_obs = v_obs[mask]
            v_pred_use = np.interp(r_use, r_bary, v_pred)
            d = np.log10(np.clip(v_pred_use,1e-12,None)/np.clip(v_obs,
1e-12,None))
            residuals.append(np.mean(d))
            absdex.append(np.median(np.abs(d)))
            npts += len(d)
        if npts==0:
            return (float('inf'), float('inf'), 0)
        return (float(np.mean(residuals)), float(np.mean(absdex)), npts)

    # 1-D search
    grid = np.linspace(10.0, 40.0, 61)  # s grid per spec suggestion
    best_s = None
    if target=='bias':
        # Find s where mean residual ~ 0 (closest to 0)
        best = (1e9, 0.0)
        for s in grid:
            b, dex, n = score(s)
            if n==0: continue
            if abs(b) < abs(best[0]):
                best = (b, s)
        best_s = best[1]
    else:
        # Min median DEX
        best = (1e9, 0.0)
        for s in grid:
            b, dex, n = score(s)
            if n==0: continue
            if dex < best[0]:
                best = (dex, s)
        best_s = best[1]
    if best_s is None:
        raise RuntimeError("tune_cluster_scale: no overlapping obs; cannot
tune")
    return float(best_s)
```

```
# ==============================
# Master curve (optional)
# ==============================

def render_master_curve_pdf(out_pdf: str):
    ensure_dir(out_pdf)
    with PdfPages(out_pdf) as pdf:
        fig, ax = plt.subplots()
        ax.set_xscale('log'); ax.set_yscale('log')
        gb = np.logspace(-14, -8, 400)
        ax.plot(gb, mapping_y_from_gbar(gb, GDAG_GAL), label='Galaxy kernel
(g†=8.6e-11)')
        ax.plot(gb, mapping_y_from_gbar(gb, GDAG_CLU), label='Cluster kernel
(g†=1.77e-9)')
        ax.set_xlabel('ḡ (m s⁻²)'); ax.set_ylabel('g (m s⁻²)'); ax.legend()
        ax.set_title('BCR/cBCR Master Mapping (reference)')
        pdf.savefig(fig, dpi=120); plt.close(fig)


# ==============================
# Main
# ==============================

def main():
    ap = argparse.ArgumentParser(description='BCR & cBCR Bootstrap Modeler
(PDF-only by default)')
    ap.add_argument('--galaxy-baryon', type=str,
help='Galaxy baryon enclosed mass XLSX/CSV (expects columns: galaxy, r_kpc,
Mbar_enc_Msun)')
    ap.add_argument('--galaxy-obs', type=str, help='Galaxy observed
velocities CSV (columns: galaxy, r_kpc, v_obs_kms [, v_err_kms])')
    ap.add_argument('--cluster-hydro-dir', type=str, help='Directory with
A####_hydro_mass*.csv files')
    ap.add_argument('--cluster-gas-dir', type=str, help='Directory with
A####_fgas_profile*.csv or A####_gas_profile*.csv files')
    ap.add_argument('--cluster-obs', type=str, help='Cluster observed
kinematics CSV (cluster, r_kpc, v_obs_kms OR sigma_kms)')
    ap.add_argument('--sigma-to-vcirc', type=float, default=math.sqrt(2.0),
help='Factor to convert dispersion to circular velocity (default sqrt(2))')
    ap.add_argument('--tune-clusters', action='store_true', help='Tune
cluster scale s (galaxies fixed) — requires --cluster-obs')
    ap.add_argument('--out-gal-pdf', type=str,
default='BCR_galaxies_plots.pdf')
    ap.add_argument('--out-dex-pdf', type=str,
default='BCR_full_dex_report.pdf', help='DEX pack PDF output (3 pages).
Always PDF-only; no PNG/CSV.')
    ap.add_argument('--out-clu-pdf', type=str,
default='cBCR_clusters_plots.pdf')
    ap.add_argument('--out-master-pdf', type=str, default=None)
```

```
    ap.add_argument('--make-master-curve', action='store_true')
ap.add_argument('--export-csv', action='store_true', help='Allow CSV writes
(OFF by default).')
    ap.add_argument('--require-tier', type=str, default='A',
choices=['A','B'])
    ap.add_argument('--profile', type=str, default=None,
choices=['fast','standard','high'], help='Override auto profile')

    args = ap.parse_args()
raise RuntimeError('LEGACY_DEX_DISABLED: Use built-in render_dex_report() —
auto-runs when observations are present.')

    # Self-test & profile decision
    decision = pilot_selftest()
    if args.profile:
        decision.profile = args.profile
    if args.require-tier == 'A' and decision.tier != 'A':
        raise RuntimeError('MODEL_TIER_TOO_LOW: require Tier-A but
environment scored Tier-B')

    # Galaxies
    if args.galaxy_baryon:
        if args.galaxy_baryon.lower().endswith('.xlsx'):
            gal_bary = load_galaxy_baryon_xlsx(args.galaxy_baryon)
        else:
            df = pd.read_csv(args.galaxy_baryon)
            assert_columns(df, ['galaxy','r_kpc','Mbar_enc_Msun'], 'GALAXY')
            gal_bary = df.copy()
        gal_obs = load_galaxy_obs_csv(args.galaxy_obs) if args.galaxy_obs
else None
        render_galaxies_pdf(gal_bary, gal_obs, args.out_gal_pdf, decision,
export_csv=args.export_csv)
        # Authoritative DEX report: run automatically if observations are
present
        if gal_obs is not None and len(gal_obs):
            render_dex_report(gal_bary, gal_obs, args.out_dex_pdf,
export_csv=False)
            print(f"Wrote {args.out_dex_pdf}")
        log(f"Wrote {args.out_gal_pdf}")

    # Clusters
    if args.cluster_hydro_dir and args.cluster_gas_dir:
        pairs = discover_cluster_pairs(args.cluster_hydro_dir,
args.cluster_gas_dir)
        if not pairs:
            log('No cluster pairs discovered.')
        clu_obs = load_cluster_obs_csv(args.cluster_obs) if args.cluster_obs
else None
        s_scale = 1.0
        if args.tune_clusters:
            if clu_obs is None:
```

```
                raise
RuntimeError('SCHEMA_FAIL_CLUSTER_OBS: tuning requested but no --cluster-obs
provided')
            # Option A (bias target)
            sA = tune_cluster_scale(pairs, clu_obs,
sigma_to_vcirc=args.sigma_to_vcirc, target='bias')
            # Option B (DEX target)
            sB = tune_cluster_scale(pairs, clu_obs,
sigma_to_vcirc=args.sigma_to_vcirc, target='dex')
            log(f"Cluster tuning: s_A* (bias≈0)={sA:.3f}, s_B* (min
DEX)={sB:.3f}")
            # Choose which to render; default to bias target for neutral
scaling
            s_scale = sA
        render_clusters_pdf(pairs, args.out_clu_pdf, decision, clu_obs,
s_scale=s_scale, export_csv=args.export_csv)
        log(f"Wrote {args.out_clu_pdf}")

    # Master curve (optional)
    if args.make_master_curve:
        out = args.out_master_pdf or 'BCR_cBCR_master_curve.pdf'
        render_master_curve_pdf(out)
        log(f"Wrote {out}")


if __name__ == '__main__':
    try:
        main()
    except Exception as e:
        log(str(e))
        sys.exit(2)
```

End of Dual Spec. This canvas now contains both the complete contextual spec and the full code, kept in sync. Edit here once, and we'll keep the implementation matched 1:1.

DEX Pack — Authoritative Microscopic Protocol (PDF-only)

HARD OUTPUT POLICY - Emit **exactly one** file: `BCR_full_dex_report.pdf` with **three pages** in this order: (1) Headline DEX, (2) CDF, (3) Histogram. - Do **not** write PNGs or CSV unless `--export-csv` is explicitly set. If any `*.png` or `*.csv` appears without this flag, abort with `OUTPUT_POLICY_VIOLATION`.

MANDATORY when observations are present (≥1 galaxy with ≥3 overlap points).

Microscopic Steps (acceleration space; three models)

1. For each eligible galaxy, at observed radii r_i within the baryon range:

2. Compute $\bar{g}(r_i)$ from enclosed baryon mass (enforce cummax).
3. Compute g_BCR(r_i) via Kernel V6 with $g\dagger = 8.6\times10^{-11}$ m s^{-2}.
4. Compute g_RAR(r_i) with $a_0 = 1.2\times10^{-10}$ m s^{-2}.
5. Compute g_obs(r_i)=v_obs(r_i)^2/r_i (km/s→m/s; kpc→m).
6. Per-galaxy RMS-dex: RMSdex_m = sqrt(mean((log$_{10}$(g_m/g_obs))^2)) for **Baryon**, **RAR**, **BCR**. Also record v_max (km/s) and dwarf flag (v_max $\leq$ 50).
7. Aggregate medians (3 decimals) across the sample: Baryon-only, RAR, BCR; print sample size and dwarf/non-dwarf counts.

CDF Page (must contain 3 curves)

- Axes: X=`RMS-dex`, Y=`CDF`; Y∈[0,1]; X∈[0, max(1.0, ceil(max_all*10)/10)].
- ECDF: sort ascending; y=linspace(1/n,1,n).
- Colors: BCR #F5B000, RAR #F28E2B, Newtonian #E15759; linewidth=2.0; legend upper-left with labels `BCR (n={n})`, `RAR (n={n})`, `Newtonian (n={n})`.

Histogram Page (must contain 3 series)

- Bins: 30 over same X limits; overlaid, alpha=0.60; same colors/labels; legend upper-left.

Validation Checks

- `DEX_SERIES_MISSING` if any model series is empty.
- `DEX_LEGEND_INCOMPLETE` if legend misses any of the three labels on either page.
- `DEX_REPORT_INCOMPLETE` if PDF pages != 3.
- `OUTPUT_POLICY_VIOLATION` if disallowed artifacts are written.

Code — Authoritative DEX Routine (supersedes earlier drafts)

```
import glob

def _ecdf(data):
    x = np.sort(np.asarray(data)); y = np.linspace(1/len(x), 1.0, len(x));
return x, y

def render_dex_report(gal_bary: pd.DataFrame, gal_obs: pd.DataFrame, out_pdf:
str, export_csv: bool=False):
    if export_csv:
        raise RuntimeError('OUTPUT_POLICY_VIOLATION: CSV export disabled for
DEX report in this build')

    rows = []
    for g in sorted(set(gal_bary['galaxy'])):
        G = gal_bary[gal_bary['galaxy']==g].copy().sort_values('r_kpc')
        if len(G) < 3: continue
        G['Mbar_enc_Msun'] = G['Mbar_enc_Msun'].cummax()
        r_b = G['r_kpc'].values.astype(float); M_b =
G['Mbar_enc_Msun'].values.astype(float)
        O = gal_obs[gal_obs['galaxy']==g].copy().sort_values('r_kpc')
```

```
        O = O[(O['r_kpc'].values >= r_b.min()) & (O['r_kpc'].values <=
r_b.max())]
        if len(O) < 3: continue
        r_o = O['r_kpc'].values.astype(float)
        # ḡ at baryon radii then interpolate to obs radii
        r_m_b = r_b*KPC_TO_M; M_kg = M_b*MSUN_TO_KG
        vbar_b = np.sqrt(np.clip(G_SI*M_kg/r_m_b, 0, None)); gbar_b =
(vbar_b**2)/r_m_b
        gbar_o = np.interp(r_o, r_b, gbar_b)
        # models & obs
        g_bary = gbar_o; g_rar = rar_from_gbar(gbar_o); g_bcr =
mapping_y_from_gbar(gbar_o, GDAG_GAL)
        v_obs = O['v_obs_kms'].values.astype(float)*1000.0; r_m_o =
r_o*KPC_TO_M; g_obs = (v_obs**2)/r_m_o
        def rmsdex(gm): d = np.log10(np.clip(gm,1e-50,None)/np.clip(g_obs,
1e-50,None)); return float(np.sqrt(np.mean(d**2)))
        rows.append({'galaxy': g, 'rmsdex_baryon': rmsdex(g_bary),
'rmsdex_rar': rmsdex(g_rar), 'rmsdex_bcr': rmsdex(g_bcr), 'vmax':
float(np.max(v_obs)/1000.0)})

    df = pd.DataFrame(rows); n = len(df)
    if n == 0: raise RuntimeError('DEX_SERIES_MISSING: no eligible galaxies
with ≥3 overlap points')
    n_dwarf = int((df['vmax'] <= 50.0).sum()); n_ndwarf = int(n - n_dwarf)
    med_bar = float(df['rmsdex_baryon'].median()); med_rar =
float(df['rmsdex_rar'].median()); med_bcr = float(df['rmsdex_bcr'].median())

    ensure_dir(out_pdf)
    with PdfPages(out_pdf) as pdf:
        # Page 1
        fig, ax = plt.subplots(figsize=(11,8.5)); ax.axis('off')
        ax.text(0.03, 0.92, 'Full DEX Report — Baryon-only vs RAR vs BCR
(tuned)', fontsize=18, weight='bold')
        s = (f'Sample size: {n} galaxies (N_overlap ≥ 3)   Dwarfs (vmax ≤ 50
km/s): {n_dwarf}   |   Non-dwarfs: {n_ndwarf}
'
             f'Headline median RMS-dex (acceleration):   Baryon-only:
{med_bar:.3f}   RAR: {med_rar:.3f}   BCR (tuned): {med_bcr:.3f}')
        ax.text(0.03, 0.83, s, fontsize=12); pdf.savefig(fig, dpi=120);
plt.close(fig)

        max_all = float(np.max([df['rmsdex_baryon'].max(),
df['rmsdex_rar'].max(), df['rmsdex_bcr'].max()])); x_max = max(1.0,
math.ceil(max_all*10)/10)
        # Page 2 — CDF (3 curves required)
        fig, ax = plt.subplots(figsize=(11,8.5))
        for series, label, color in [ (df['rmsdex_bcr'].values, f'BCR
(n={n})', '#F5B000'), (df['rmsdex_rar'].values, f'RAR (n={n})', '#F28E2B'),
(df['rmsdex_baryon'].values, f'Newtonian (n={n})', '#E15759') ]:
            x,y = _ecdf(series); ax.plot(x,y, label=label, linewidth=2.0,
color=color)
```

```
        ax.set_xlabel('RMS-dex'); ax.set_ylabel('CDF'); ax.set_xlim(0.0,
x_max); ax.set_ylim(0.0, 1.0); ax.grid(True, alpha=0.30)
        leg = ax.legend(loc='upper left', frameon=True); labels =
{t.get_text() for t in leg.get_texts()}
        must = {f'BCR (n={n})', f'RAR (n={n})', f'Newtonian (n={n})'}
        if not must.issubset(labels): raise
RuntimeError('DEX_LEGEND_INCOMPLETE: CDF legend missing one or more series')
        pdf.savefig(fig, dpi=150, bbox_inches='tight'); plt.close(fig)

        # Page 3 — Histogram (3 series required)
        fig, ax = plt.subplots(figsize=(11,8.5))
        bins = np.linspace(0.0, x_max, 31)
        ax.hist(df['rmsdex_bcr'].values, bins=bins, alpha=0.60, label=f'BCR
(n={n})', color='#F5B000')
        ax.hist(df['rmsdex_rar'].values, bins=bins, alpha=0.60, label=f'RAR
(n={n})', color='#F28E2B')
        ax.hist(df['rmsdex_baryon'].values, bins=bins, alpha=0.60,
label=f'Newtonian (n={n})', color='#E15759')
        ax.set_xlabel('RMS-dex'); ax.set_ylabel('Count'); ax.grid(True,
alpha=0.30)
        leg = ax.legend(loc='upper left', frameon=True); labels =
{t.get_text() for t in leg.get_texts()}
        if not must.issubset(labels): raise
RuntimeError('DEX_LEGEND_INCOMPLETE: Histogram legend missing one or more
series')
        pdf.savefig(fig, dpi=150, bbox_inches='tight'); plt.close(fig)

    # Post-run guard: disallowed artifacts
    rogue = [p for p in glob.glob('*.png')+glob.glob('*.csv')]
    if rogue: raise RuntimeError('OUTPUT_POLICY_VIOLATION: Disallowed
artifacts present: ' + ', '.join(rogue))
```