

# VISKOPIC

## Consistency Pipeline R&D Textbook

Session record: 10 August 2026  
Next-session plan: 11 August 2026

*From a single-window stylometric baseline to multi-window verification,  
multi-author stress testing, and the calibration problem discovered today.*

**IMPORTANT:** This is an internal technical learning document. The similarity score is not a probability of authorship and should not be used as an automatic academic-misconduct decision.

# How to use this textbook

This document records the technical decisions, experiments, code, failures, fixes, results, and next steps from today's Viskopic consistency-pipeline session. It is written so that tomorrow's work can start without reconstructing context from terminal history.

- Chapters 1-3 explain the baseline mathematically and separate current implementation from future R&D.
- Chapters 4-6 document the cleaning and multi-window experiments that motivated Baseline V1.1.
- Chapters 7-9 document cross-author validation and the calibration weakness exposed by the five-author run.
- Chapter 10 records engineering/debugging lessons so the same mistakes are not repeated.
- Chapter 11 is the concrete plan for 11 August 2026.
- The appendices contain the exact multi-window code written today plus key baseline scoring code.

## Contents

1. Research question and system architecture
  2. Baseline Consistency V1 - mathematical mechanics
  3. Distribution assumptions, normality, and why mixture models moved to future R&D
  4. Author 007: cleaning, passage sampling, and what failed
  5. Baseline V1.1: multi-window document representation
  6. Author 007 multi-window results
  7. First cross-author discrimination test
  8. Dataset expansion and five-author validation
  9. The calibration problem discovered today
  10. Engineering and debugging diary
  11. Tomorrow's plan: calibration-only experiments
- Appendix A. `build_multiwindow_corpus_v1_fixed.py`
- Appendix B. `evaluate_multiwindow_authorship_v1.py`
- Appendix C. `evaluate_multiwindow_discrimination_v1.py`
- Appendix D. Key baseline scoring code
- Appendix E. Calibration-comparison prototype for tomorrow
- Appendix F. Reproducibility commands and decision log

# 1. Research question and system architecture

The immediate product question is not 'Can we detect AI text?' Today's work stayed focused on the consistency branch: can a new document be compared with an author's historical writing in a way that is stable, interpretable, and empirically defensible?

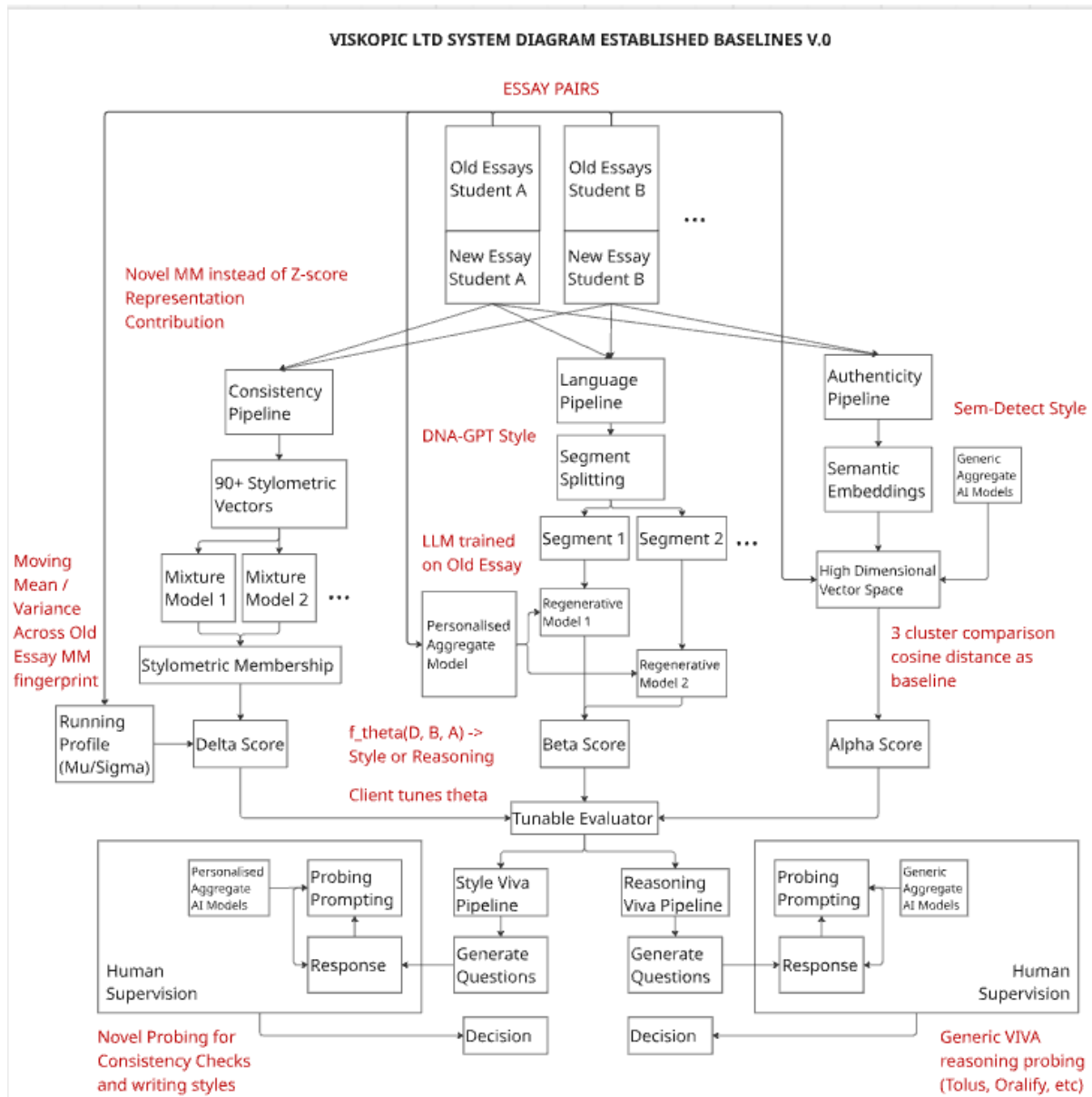


Figure 1. The broader Viskopic system diagram discussed today. The left-hand Consistency Pipeline is the branch implemented and tested in this session.

## 1.1 What belongs in the current baseline

| Diagram idea                   | Current status                                                                   | Decision                                 |
|--------------------------------|----------------------------------------------------------------------------------|------------------------------------------|
| 90+ stylometric vectors        | Implemented                                                                      | Keep in baseline                         |
| Running profile (mu/sigma)     | Implemented as recalculated reference profile                                    | Keep, but not yet an online profile      |
| Delta score                    | Implemented as distance $\rightarrow$ relative distance $\rightarrow$ similarity | Keep, calibration now under review       |
| Mixture models                 | Not implemented                                                                  | Move to future R&D                       |
| Language / regeneration branch | Not part of today's work                                                         | Separate future branch                   |
| AI-generation evidence branch  | Not part of today's work                                                         | Separate future branch                   |
| Tunable evaluator              | Future                                                                           | Only after branch outputs are calibrated |

**NOTE:** The diagram originally mixed current implementation and future research. Today's key architecture decision was to freeze a simple consistency baseline and force future complexity to beat it on held-out data.

## 1.2 Important architecture corrections

- Use a shared ingestion/cleaning/segmentation stage before downstream branches. Segmentation matters to stylometry, not only to language/regeneration.
- Treat the input as an historical author corpus plus a candidate document, not as isolated essay pairs.
- Do not automatically update an author profile with every submitted document; that creates a profile-poisoning risk.
- Keep multi-modal/mixture-based author profiling in future R&D until the baseline is properly measured.
- Treat the consistency score as evidence for review, not as a probability or an automatic misconduct verdict.

## 1.3 Working baseline after today's decisions

```

Historical documents
  |
  v
clean prose extraction
  |
  v
3 distributed ~550-word windows per document
  |
  v
stylometric vector per window
  |
  v
reference author profile (mean + SD per usable feature)
  |
  v
candidate window z-scores
  |
  v
group-balanced RMS distance per window
  |
  v
median(candidate window distances)
  |
  v
document-level genuine leave-one-out calibration
  |
  v
Delta / relative distance + similarity
PLUS: window-level variation diagnostics

```

## 2. Baseline Consistency V1 - mathematical mechanics

Before the multi-window upgrade, each essay was represented by one feature vector. The current scorer is deliberately hand-designed and interpretable. It is not a learned neural metric and it is not a simple Euclidean distance over every raw column.

### 2.1 Feature extraction

extract\_features\_v2\_new.py produces 132 total output columns in the earlier runs, with roughly 122 candidate stylometric features considered before size/variance filtering. The families include sentence structure, vocabulary, readability, punctuation, pronouns/stance, function words, transitions, and sentence openings.

| Group              | Examples                                                                  |
|--------------------|---------------------------------------------------------------------------|
| Sentence structure | average sentence length, sentence-length SD, long/short sentence ratios   |
| Vocabulary         | word length, TTR variants, hapax/dislegomena ratios, long-word ratios     |
| Readability        | syllables/word, complex-word ratio, Flesch, FK grade, Gunning Fog         |
| Punctuation        | commas, semicolons, colons, questions, exclamations, parentheses, hyphens |
| Pronouns / stance  | first-, second-, third-person rates, contractions                         |
| Function words     | feature columns prefixed function_                                        |
| Transitions        | feature columns prefixed transition_                                      |
| Sentence openings  | feature columns prefixed sentence_opening_                                |

### 2.2 Excluded raw-size features

Raw document-size quantities are excluded from distance so that a longer document does not look stylistically different merely because it contains more words or characters. The current exclusion set also removes several size-derived extrema and paragraph-length measures.

### 2.3 Low-variance filter

Nearly constant features are dangerous in small author profiles because dividing by a tiny standard deviation creates enormous z-scores. A feature is retained only if its standard deviation is at least  $\max(0.02, 5\% \text{ of } |\mu_j|)$ .

$$\text{Keep feature } j \text{ only if } s_j \geq \max(0.02, 0.05 * |\mu_j|)$$

### 2.4 Standardisation

$$z_j = (x_j - \mu_j) / s_j$$

Candidate values are converted to z-scores using the current reference set and clipped to  $[-4, +4]$ . Clipping limits the effect of a single extreme feature without pretending the outlier does not exist.

### 2.5 Group-balanced distance

$$D_g = \sqrt{\text{mean}(z_j^2)} \text{ for features } j \text{ in group } g$$

$$D = \text{mean}(D_g \text{ across available feature groups})$$

This prevents large feature families such as function words or transitions from dominating simply because they contain more columns.

### 2.6 Genuine leave-one-out calibration

For a candidate document, the reference author profile is built without that candidate. To estimate what 'normal genuine variation' looks like, each reference document is in turn held out and scored against the remaining reference documents. The median genuine distance becomes the author-specific denominator.

$$R = D_{\text{candidate}} / \text{median}(D_{\text{genuine, leave-one-out}})$$

Interpretation: R around 1 means the candidate is about as far from the reference profile as a typical genuine held-out document; R below 1 is closer than typical; R above 1 is more unusual.

## 2.7 Display similarity

$$S = 100 * \exp(-R^2 / 2)$$

This is a monotonic display transform, not a probability. In particular,  $R = 1$  maps to about 60.65, which explains why genuine median scores naturally sit near 60 in many of the experiments.

**NOTE:** Group-level display scores in the older reports use the raw group distance transform directly; they are not independently author-calibrated. Overall similarity is calibrated, group similarity is diagnostic.

## 3. Distribution assumptions, normality, and why mixture models moved to future R&D

### 3.1 Does the baseline assume every stylometric feature is normally distributed?

No formal normality test or Gaussian likelihood is used. A z-score can be calculated for any numerical distribution. What the baseline does assume is weaker: mean and standard deviation are useful summaries of the author's feature values.

Normality would become necessary only if we interpreted a z-score as a normal-theory probability or p-value. We do not do that. Likewise, the  $\exp(-R^2/2)$  similarity transform has a Gaussian-shaped form but is used only as a smooth similarity scale.

### 3.2 Why this still matters

If an author's feature distribution is strongly skewed, heavy-tailed, or genuinely multi-modal, one mean and one standard deviation can be misleading. A feature may have two stable writing modes with the global mean lying in a region the author rarely occupies.

Example conceptual feature values

Mode A: 18 19 19 20 20  
Mode B: 31 31 32 32 33

Global mean ~25.5

A candidate near 25.5 would look central to a one-centroid model, even though the author may rarely write at that value.

### 3.3 Mixture model decision

The mixture-model idea is scientifically interesting, but it is future R&D, not Baseline V1. A high-dimensional Gaussian mixture over roughly 100 features with only a small number of papers would be unstable and could invent clusters from noise.

- First establish the simple baseline.
- Use multi-window data to measure within-document and between-document variation.
- Inspect whether stable multiple modes actually exist.
- Only then test mixture models, robust distributions, dimensional reduction, or learned metrics against the frozen baseline.

**NOTE:** Decision recorded today: prove the baseline first; complexity has to earn its place.

## 4. Author 007: cleaning, passage sampling, and what failed

Author 007 (Luciano Floridi) became the main methodological debugging corpus because the collection was expanded to 20 solo English papers and included works across years and topics.

### 4.1 Original 20-paper single-window baseline

| Statistic                | Original single-window |
|--------------------------|------------------------|
| Papers                   | 20                     |
| Mean similarity          | 58.50                  |
| Median similarity        | 60.42                  |
| Minimum                  | 27.16                  |
| Maximum                  | 78.86                  |
| Mean relative distance   | 1.040                  |
| Median relative distance | 1.004                  |

The wide genuine range prompted an investigation into whether extraction dirt, passage choice, or genuine stylistic variation was driving the outliers.

### 4.2 Cleaning experiment 1: full-PDF cleaning plus resampling

`clean_and_sample_text_v2_1.py` reopened each PDF, cleaned repeated headers/footers and reference-like material, then selected representative clean prose. It produced 20 usable samples with tightly controlled lengths around 1,500 words.

| Experiment                  | Mean  | Median | Min   | Max   | Mean R |
|-----------------------------|-------|--------|-------|-------|--------|
| Original passages           | 58.50 | 60.42  | 27.16 | 78.86 | 1.040  |
| Same passages + light clean | 58.62 | 60.78  | 21.95 | 80.51 | 1.040  |
| Clean + resampled v2.1      | 55.27 | 60.09  | 23.22 | 80.63 | 1.101  |

The resampled pipeline did not improve overall genuine consistency. Its mean fell by more than 3 points, while the median remained similar. This immediately suggested that 'cleaning' and 'which passage was selected' had been confounded.

### 4.3 Controlled experiment: clean the exact same existing passages

`clean_existing_samples_v1.py` removed only light extraction artefacts from the existing text samples without changing passage selection or ordering. The mean word-count change was about -1.55%, and the overall Author 007 score distribution was almost unchanged.

| Measure                  | Controlled light clean |
|--------------------------|------------------------|
| Mean similarity          | 58.62                  |
| Median similarity        | 60.78                  |
| Min / max                | 21.95 / 80.51          |
| Mean relative distance   | 1.040                  |
| Median relative distance | 0.998                  |

**NOTE:** Strong finding: basic PDF dirt was not the dominant source of overall instability. Passage selection mattered much more.

### 4.4 Paired sampling comparison

The fixed filename-matching comparison aligned all 20 papers across original, light-clean, and resampled experiments. Mean absolute sampling swing was 8.93 similarity points; the median absolute swing was 6.38; the maximum swing was 31.53.

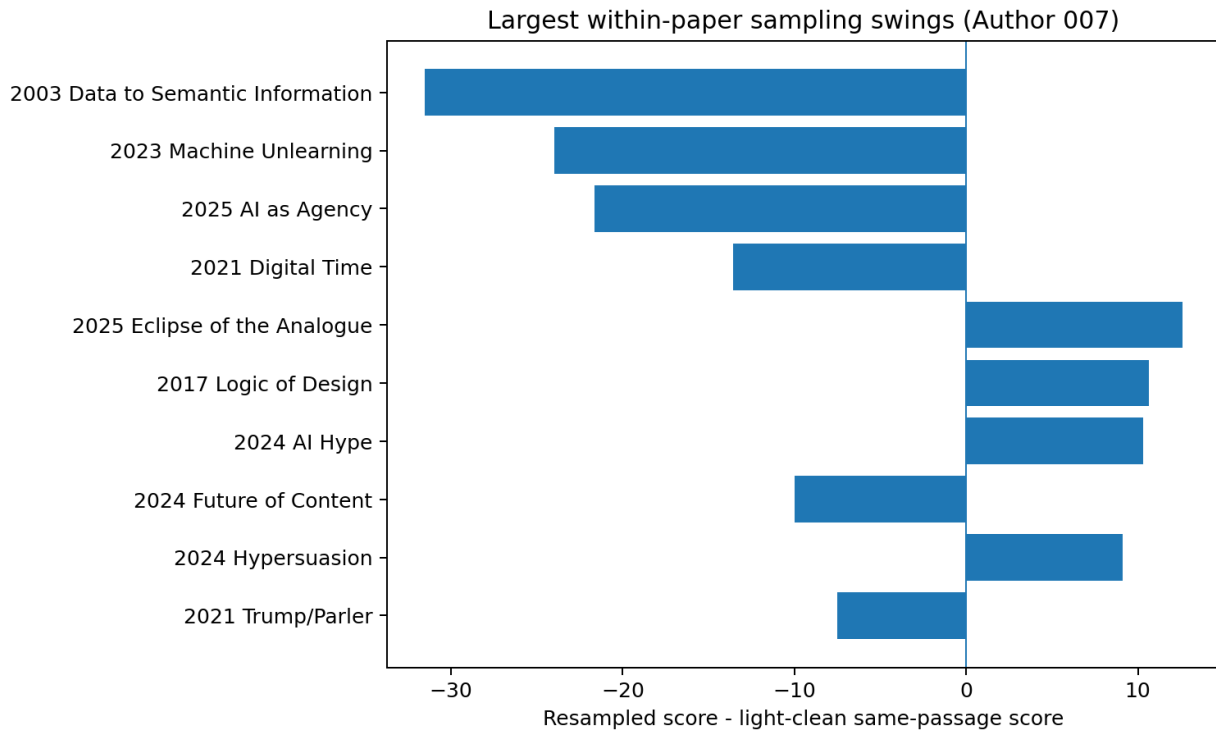


Figure 2. Largest score changes caused by selecting a different passage from the same genuine paper.

| Paper                             | Original | Light clean | Resampled | Sampling swing |
|-----------------------------------|----------|-------------|-----------|----------------|
| 2003 Data -> Semantic Information | 68.27    | 68.97       | 37.44     | -31.53         |
| 2023 Machine Unlearning           | 78.86    | 80.51       | 56.51     | -24.00         |
| 2025 AI as Agency                 | 42.74    | 44.86       | 23.22     | -21.64         |
| 2021 Digital Time                 | 68.16    | 68.60       | 55.04     | -13.56         |
| 2025 Eclipse of the Analogue      | 58.78    | 57.99       | 70.59     | +12.60         |
| 2017 Logic of Design              | 60.77    | 57.83       | 68.48     | +10.65         |
| 2024 AI Hype                      | 64.27    | 53.37       | 63.68     | +10.31         |

The direction was balanced: different passages could make a genuine paper look much closer or much farther from the same author profile. The central weakness was therefore not just noise removal; it was the assumption that one arbitrary ~1,500-word extract adequately represented an entire document.

## 4.5 Persistent vs sampling-sensitive outliers

Two papers illustrated different phenomena. The 2003 Data paper moved from roughly 69 to 37 under resampling, showing strong within-document variation. Maker's Knowledge remained low across original, light-clean, and resampled conditions, suggesting a genuinely unusual document mode rather than only a bad extract.

## 5. Baseline V1.1: multi-window document representation

The next experiment changed only the document representation. The stylometric feature set, z-scoring, group-balanced distance, and genuine calibration were deliberately left unchanged.

### 5.1 Window design

The first builder used five 600-word windows, but the real corpus exposed a design problem: several papers contained fewer than 3,000 clean words, forcing substantial overlap. The experiment was immediately tightened to three distributed windows of about 550 words.

```
Paper
|
+-- early window ~550 words
+-- middle window ~550 words
+-- late window ~550 words
|
v
3 stylometric vectors
```

The builder preserves sentence boundaries, so actual window lengths vary slightly around the target rather than cutting sentences mechanically.

### 5.2 Whole-document leakage rule

**NON-NEGOTIABLE:** When Paper 7 is the candidate, ALL windows from Paper 7 are excluded from the reference profile and from candidate calibration. The unit of holdout is the whole document, never an individual window.

### 5.3 Document score

Each candidate window is scored against reference windows. The three raw window distances are then aggregated with the median to produce one document distance.

$$D_{\text{document}} = \text{median}(D_{\text{window1}}, D_{\text{window2}}, D_{\text{window3}})$$

Median aggregation reduces the chance that one equation-heavy, reference-heavy, rhetorical, or otherwise unusual section defines the entire paper.

### 5.4 QC rule

A later quality-control rule required at least 1,650 clean words for a three-by-550 design. This is not model tuning; it enforces the intended sampling geometry and avoids pretending that three heavily overlapping windows are independent-ish observations.

### 5.5 Window-level diagnostics

V1.1 preserves window minimum, quartiles, median, maximum, IQR, and range. That adds a second descriptive dimension: not only how far the document is from the profile, but how internally variable its style is across sections.

## 6. Author 007 multi-window results

The final Author 007 V1.1 run used 20 papers and 60 windows, exactly three windows per paper. All candidate windows were held out together.

| Statistic                | Multi-window V1.1 |
|--------------------------|-------------------|
| Papers tested            | 20                |
| Mean similarity          | 58.10             |
| Median similarity        | 60.95             |
| Minimum / maximum        | 25.92 / 78.92     |
| Mean relative distance   | 1.047             |
| Median relative distance | 0.995             |
| Median window IQR        | 0.112             |
| Median window range      | 0.224             |

The overall author distribution remained extremely close to the original single-window baseline. That is important: the multi-window method did not simply inflate all genuine scores or replace the scoring system with something fundamentally different.

### 6.1 Case study: 2003 Data -> Semantic Information

| Measure                   | Value         |
|---------------------------|---------------|
| Multi-window similarity   | 53.9          |
| Relative distance         | 1.111         |
| Window raw-distance range | 0.681 - 1.169 |
| Window IQR                | 0.244         |

The document-level result landed between the earlier single-passage scores of about 68 and 37. This is exactly what the experiment was designed to reveal: the paper contains stylistically different regions, so one passage can give a misleadingly extreme representation.

### 6.2 Case study: Hypersuasion

Hypersuasion scored 78.9 and had a very narrow window-distance range (0.576-0.670; IQR 0.047). Its sections consistently look close to the author profile.

### 6.3 Case study: AI as Agency

AI as Agency scored 25.9 and had a broad window range (0.880-1.560; IQR 0.340). It is unusual overall and internally heterogeneous.

### 6.4 New conceptual output

The experiment suggests that a future report should preserve both document deviation and within-document variability rather than collapsing everything into one percentage. We did NOT define a new variability score today; IQR/range remain diagnostics.

## 7. First cross-author discrimination test

After multi-window representation improved document interpretation, the next question was whether the extra tolerance destroyed discrimination between authors. The first sanity test used Author 004 and Author 007.

| Metric                           | 2-author result |
|----------------------------------|-----------------|
| Candidate documents              | 28              |
| Same-author comparisons          | 28              |
| Different-author comparisons     | 28              |
| ROC-AUC                          | 0.835           |
| EER                              | 0.179           |
| Best balanced accuracy           | 0.821           |
| Top-1 author identification      | 0.929           |
| MRR                              | 0.964           |
| Same-author mean similarity      | 58.67           |
| Different-author mean similarity | 43.91           |
| Same-author mean relative D      | 1.036           |
| Different-author mean relative D | 1.292           |

This passed the first sanity gate: same-author candidates remained closer to their own profiles on average, and 26 of 28 documents ranked their true profile first. The two failures were both Author 004 papers, which is consistent with the known noisier/co-authorship concerns around that corpus.

**NOTE:** This result was encouraging but not final evidence. A two-author task can be artificially easy if the authors or disciplines are very different.

## 8. Dataset expansion and five-author validation

Rather than continue tuning on Floridi, the baseline was frozen and attacked with more authors. Three new nearby-domain authors were collected using OpenAlex identity/solo-work discovery and CORE full-text backfill where available.

| Alias      | Author                       | OpenAlex ID | Collected PDFs | QC papers | QC windows |
|------------|------------------------------|-------------|----------------|-----------|------------|
| author_004 | existing noisy/stress corpus | -           | 8              | 8         | 24         |
| author_007 | Luciano Floridi              | A5046574356 | 20             | 20        | 60         |
| author_008 | Mark Coeckelbergh            | A5003571713 | 12             | 11        | 33         |
| author_009 | Shannon Vallor               | A5076883843 | 6              | 5         | 15         |
| author_010 | John Danaher                 | A5057953350 | 12             | 12        | 36         |

The new authors were deliberately close in broad subject area - philosophy, AI ethics, digital technology, moral/political questions - so a successful system could not simply rely on easy discipline separation.

### 8.1 Vallor backfill limitation

CORE added one more Vallor PDF, bringing the corpus from five to six. Many remaining candidates failed because of HTTP 403 PDF access, HTTP 429 rate limiting, or conservative identity/match rejection. After the 1,650-clean-word QC rule, five Vallor papers remained.

**NOTE:** The conservative rejection behavior is desirable. A smaller verified corpus is better than silently contaminating an author profile with the wrong or co-authored work.

### 8.2 Five-author test size

The final run contained 56 candidate documents. Each candidate was scored against all five author profiles, yielding 56 genuine comparisons and 224 impostor comparisons.

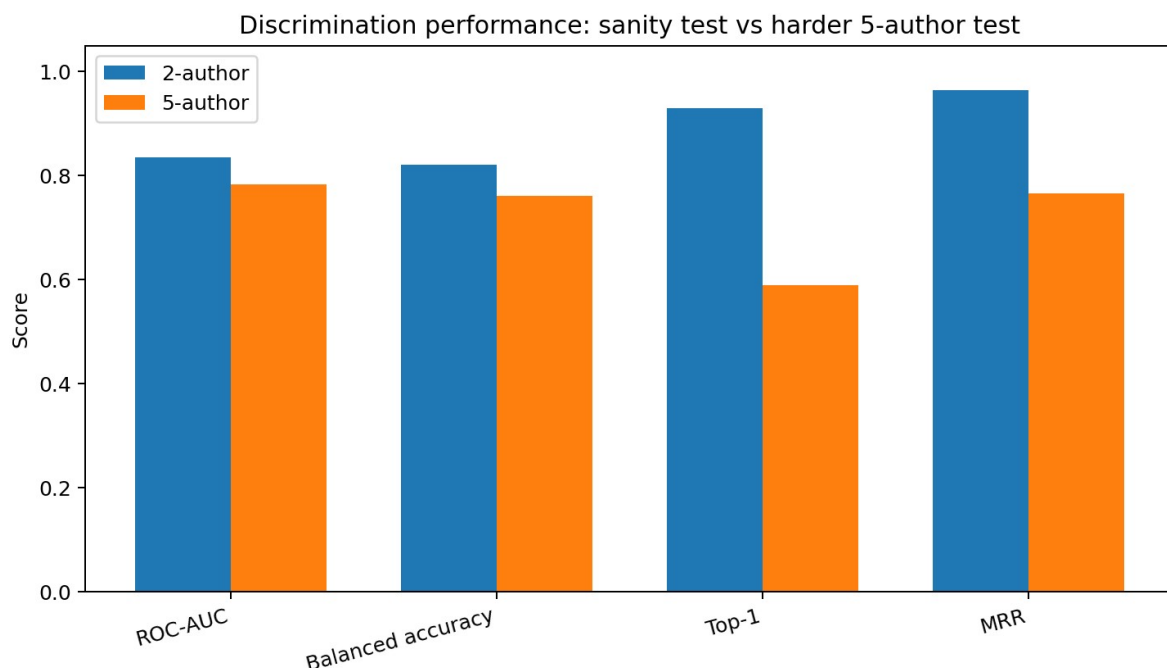


Figure 3. Higher-is-better metrics dropped when the task expanded from two authors to five similar/noisy profiles.

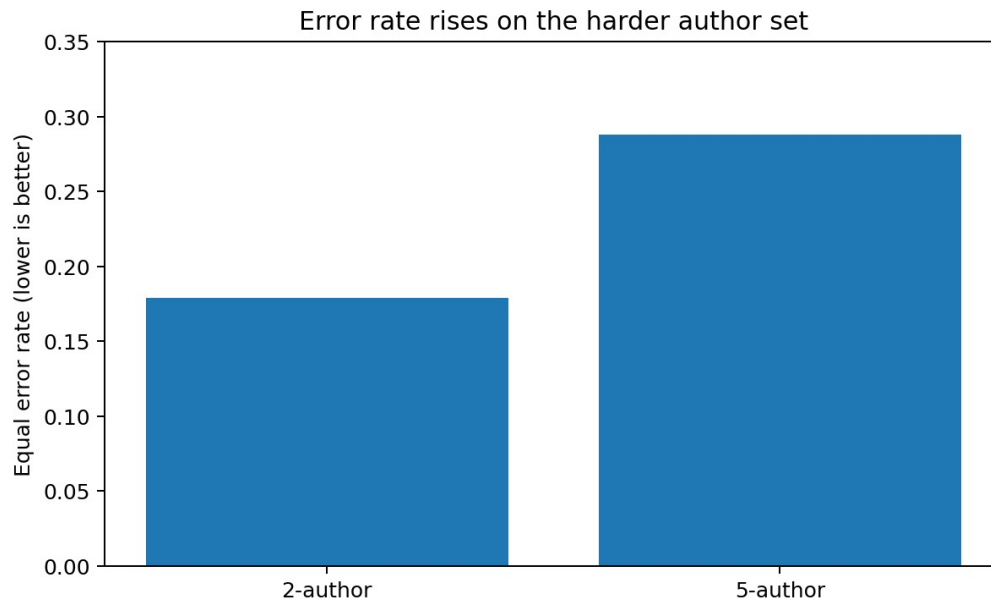


Figure 4. Equal error rate increased on the harder five-author task.

### 8.3 Five-author headline results

| Metric                             | 5-author result |
|------------------------------------|-----------------|
| ROC-AUC                            | 0.783           |
| EER                                | 0.288           |
| Best balanced accuracy             | 0.761           |
| Top-1 author identification        | 0.589           |
| MRR                                | 0.766           |
| Same-author mean similarity        | 59.72           |
| Different-author mean similarity   | 47.17           |
| Same-author median similarity      | 61.45           |
| Different-author median similarity | 48.36           |
| Same-author mean relative D        | 1.017           |
| Different-author mean relative D   | 1.239           |

The baseline still contains real personal-writing signal: genuine and impostor distributions separate on average. But the harder test exposes substantial overlap. An EER around 29% is far too high for an automatic decision system.

Top-1 author identification is not the primary Viskopic product objective. The product problem is verification ('is this compatible with this student's profile?'), not forced identification ('which of these five people wrote it?'). Still, the top-1 failures are valuable diagnostics.

## 9. The calibration problem discovered today

The five-author run exposed a specific weakness rather than a vague 'stylometry failed' conclusion: profiles with high genuine variability - especially small profiles - become overly permissive after author-specific relative-distance calibration.

### 9.1 Calibration denominators

| Profile    | QC papers | Typical genuine D |
|------------|-----------|-------------------|
| author_004 | 8         | 0.900             |
| author_007 | 20        | 0.862             |
| author_008 | 11        | 0.945             |
| author_009 | 5         | 1.079             |
| author_010 | 12        | 0.855             |

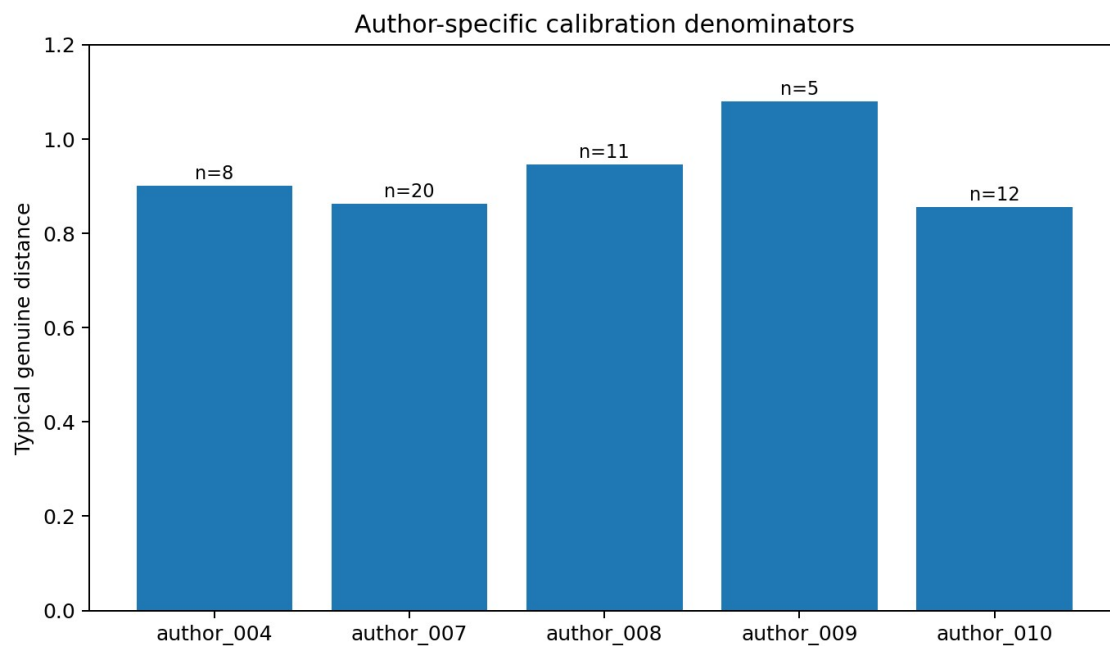


Figure 5. Author 009 has the broadest genuine calibration denominator and the smallest reference corpus.

Because  $R = D / \text{typical\_genuine\_D}$ , the same raw distance is treated as less unusual against a profile with a large denominator. A raw D of 1.10 gives  $R=1.29$  against a denominator of 0.855, but only  $R=1.02$  against a denominator of 1.079.

### 9.2 Wide-profile attractor evidence

| Profile    | Mean genuine similarity | Mean impostor similarity | Mean genuine R | Mean impostor R |
|------------|-------------------------|--------------------------|----------------|-----------------|
| author_004 | 60.13                   | 42.04                    | 1.010          | 1.323           |
| author_007 | 58.09                   | 47.28                    | 1.047          | 1.233           |
| author_008 | 59.89                   | 51.66                    | 1.014          | 1.155           |
| author_009 | 65.40                   | 57.71                    | 0.921          | 1.050           |
| author_010 | 59.68                   | 35.87                    | 1.017          | 1.457           |

Author 009 is the clearest warning: different-author documents score 57.71 on average against that profile, much higher than impostor means for the other profiles. The profile is broad enough that outsiders can look 'typical' after calibration.

### 9.3 Post-run diagnostic: raw D vs current R

Using all 280 comparisons from the five-author terminal output, a calibration-only diagnostic was computed after the run. Lower raw profile distance D was compared with lower current relative distance R.

| Metric                 | Raw D | Current R / similarity |
|------------------------|-------|------------------------|
| ROC-AUC                | 0.821 | 0.783                  |
| Approx. EER            | 0.232 | 0.288                  |
| Best balanced accuracy | 0.783 | 0.761                  |
| Top-1 author ID        | 0.661 | 0.589                  |
| MRR                    | 0.810 | 0.766                  |

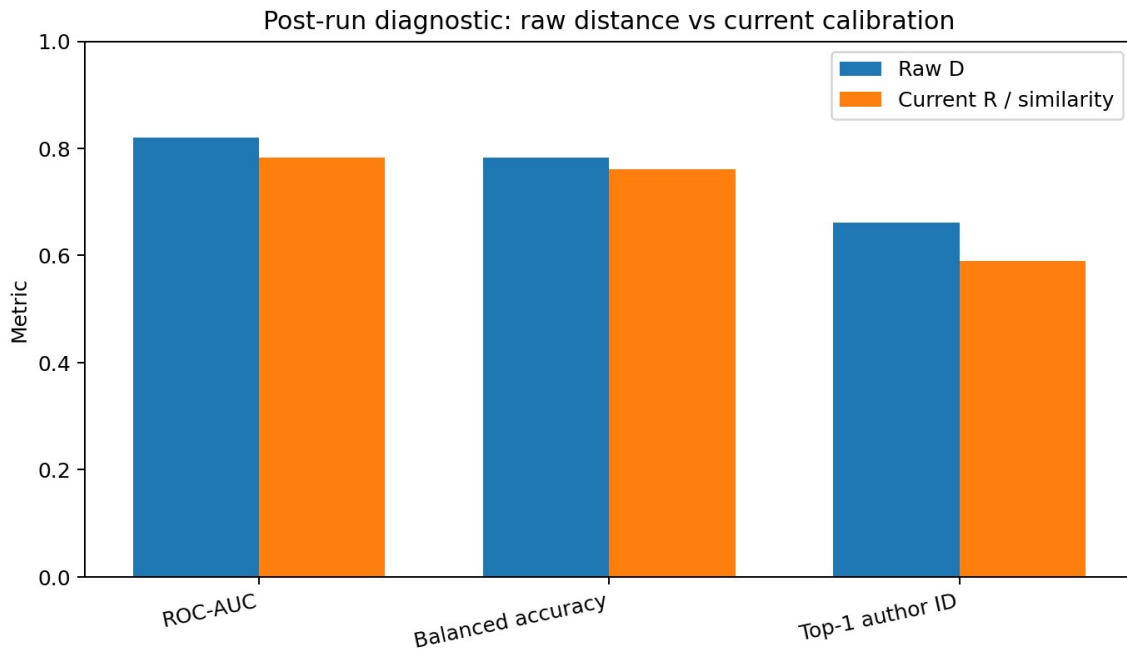


Figure 6. On today's 5-author data, raw stylometric distance outperformed the current per-author ratio calibration on several diagnostics.

**KEY FINDING:** This does NOT prove calibration should be deleted. It shows that the current calibration layer is a plausible source of avoidable error and deserves an isolated experiment before we touch features, windows, or mixture models.

### 9.4 What appears to be working vs what is under question

| Component                                        | Status after today                  |
|--------------------------------------------------|-------------------------------------|
| Stylometric feature representation               | Useful signal observed; keep frozen |
| Low-variance filtering                           | Keep frozen                         |
| Group-balanced RMS distance                      | Useful baseline; keep frozen        |
| Three-window document representation             | Provisionally earned its place      |
| Median across candidate windows                  | Keep for now                        |
| Whole-document LOO                               | Non-negotiable anti-leakage rule    |
| Author-specific D / median genuine D calibration | Under investigation tomorrow        |
| Mixture models / learned metrics                 | Future R&D; do not touch tomorrow   |

## 10. Engineering and debugging diary

---

### 10.1 Multi-window builder API mismatch

The first builder failed on every PDF because it called `cleaner.extract_pdf_text`, which did not exist in `clean_and_sample_text_v2_1.py`. The fixed builder was rewritten against the actual exported functions: `extract_pdf`, `repeated_margins`, `page_to_paragraphs`, `reject_reason`, `clean_text`, `words`, and `trim_sample_to_target`.

Lesson: when a new experiment wraps an existing script, inspect the real module interface first rather than guessing helper names.

### 10.2 Five-window design was too ambitious

The fixed builder successfully produced 100 windows (20 papers x 5), but the clean-word counts showed several papers were shorter than 3,000 words. That meant heavy overlap. The experiment was corrected before evaluation to three ~550-word windows.

Lesson: data availability can invalidate an apparently neat sampling design. Check the actual length distribution before treating windows as independent-ish observations.

### 10.3 CORE script naming confusion

An attempted command referenced `core_backfill_author_v3_1.py`, but the user's current project root contained `core_backfill_author.py` and `core_backfill_author_v2.py`. `grep` confirmed `core_backfill_author.py` uses `https://api.core.ac.uk/v3`, while the `_v2` file uses the obsolete `api-v2` route.

Lesson: script version numbers are historical clutter. A proper repository refactor is still deferred until the architecture is finalized.

### 10.4 Repo cleanup decision

The project root is cluttered, but refactoring was deliberately postponed because the research architecture is still moving quickly. The current priority is deciding what deserves to become the product pipeline. One clean refactor should happen after the baseline is frozen, not during every experiment.

# 11. Tomorrow's plan: calibration-only experiments (11 Aug 2026)

**SCOPE FREEZE:** Tomorrow's objective is narrow: isolate and repair (or remove) the calibration layer. Do not change the feature extractor, window count, window length, cleaner, group distance, or mixture-model architecture.

## 11.1 Questions to answer

1. Does raw profile distance D consistently outperform current relative distance R across overall metrics and per-author metrics?
2. How much of the problem is specifically caused by small/broad profiles such as author\_009?
3. Can a conservative shrinkage calibration retain the interpretability of 'relative to genuine variation' without turning wide profiles into attractors?
4. Does any calibration improvement survive sensitivity checks rather than merely improving the aggregate five-author number?
5. Which quantity should become Delta V1.2: raw D, current R, or a shrunk/regularised R?

## 11.2 Experiment A - raw D baseline

Treat lower raw document distance as stronger same-author evidence. Recompute ROC-AUC, EER, balanced accuracy, top-1 identification, MRR, and per-profile impostor behavior. Today's quick diagnostic already gives a reference point: AUC ~0.821 and EER ~0.232.

## 11.3 Experiment B - current R baseline

Reproduce the current metrics directly from all\_comparisons.csv. This is the control condition: AUC ~0.783, EER ~0.288, balanced accuracy ~0.761.

## 11.4 Experiment C - shrunk calibration

Instead of trusting each author's noisy typical-genuine denominator completely, shrink it toward a global typical distance. Authors with fewer reference documents receive stronger shrinkage.

$$T^*_a = w_a T_a + (1 - w_a) T_{global}$$

$$w_a = n_a / (n_a + k)$$

$$R^*_a = D / T^*_a$$

For exploration, inspect several k values, but do not select a final k solely because it maximizes the same five-author test. Final parameter selection will need nested or separate validation.

## 11.5 Sensitivity analyses

- Re-run summary excluding author\_009 to see how much the small broad profile drives the aggregate error.
- Re-run summary excluding author\_004 to separate the known noisy/co-authorship stress corpus from the philosophically adjacent authors.
- Report results per profile author, not only aggregate AUC.
- Inspect high-scoring impostors against author\_009 and low-scoring genuine documents.
- Keep the entire experiment document-level; never split windows from the same document across calibration and candidate roles.

## 11.6 Stop rule for tomorrow

If a calibration method produces a clear, defensible improvement without changing the underlying representation, freeze it as Delta V1.2. If no calibration beats raw D robustly, prefer the simpler raw distance baseline rather than adding complexity just to preserve the idea of a relative score.

## 11.7 After the calibration decision

6. Freeze Baseline Consistency V1.2.
7. Write a short automated regression test for leakage and document-level holdout.
8. Create one canonical evaluator output schema.
9. Only then resume additional author collection or pipeline refactoring.
10. Keep mixture models, robust/multimodal author distributions, and learned metrics in a future-R&D backlog.

## Appendix A. build\_multiwindow\_corpus\_v1\_fixed.py

Exact fixed multi-window corpus builder created today. It reuses the v2.1 cleaning primitives, generates distributed sentence-aligned windows, writes a manifest, and records clean-word/QC metadata.

```
from __future__ import annotations

import argparse
import csv
import re
from collections import Counter
from pathlib import Path

import clean_and_sample_text_v2_1 as cleaner

def wc(text: str) -> int:
    return cleaner.words(text)

def sentence_units(paragraphs: list[str]) -> list[str]:
    units: list[str] = []

    for paragraph in paragraphs:
        parts = re.split(
            r'(?<=[.!?])\s+(?=[A-Z0-9"'']')',
            paragraph.strip(),
        )

        for part in parts:
            part = part.strip()
            if wc(part) >= 4:
                units.append(part)

    return units

def cumulative_word_positions(
    units: list[str],
) -> tuple[list[int], int]:
    cumulative: list[int] = []
    total = 0

    for unit in units:
        total += wc(unit)
        cumulative.append(total)

    return cumulative, total

def make_centered_window(
    units: list[str],
    cumulative: list[int],
    total_words: int,
    center_fraction: float,
    target_words: int,
) -> str:
    """
    Create one contiguous, sentence-aligned window centered near a
    specified relative position in the cleaned document.
    """
    if not units:
        return ""

    target_pos = center_fraction * total_words

    center_idx = 0
    for i, cumulative_words in enumerate(cumulative):
        if cumulative_words >= target_pos:
            center_idx = i
            break

    selected = [center_idx]
    current_words = wc(units[center_idx])

    left = center_idx - 1
    right = center_idx + 1

    while current_words < target_words and (
        left >= 0 or right < len(units)
    ):
        candidates: list[tuple[int, int]] = []

        if left >= 0:
            candidates.append((left, wc(units[left])))

        if right < len(units):
            candidates.append((right, wc(units[right])))

        best_idx, best_wc = min(
            candidates,
            key=lambda item: (
                abs(target_words - (current_words + item[1])),
                abs(item[0] - center_idx),
            ),
        )
```

```

    ),
)

selected.append(best_idx)
current_words += best_wc

if best_idx == left:
    left -= 1
else:
    right += 1

selected.sort()

text = " ".join(
    units[i]
    for i in selected
).strip()

return cleaner.trim_sample_to_target(
    text,
    target=target_words,
    tolerance=0.08,
)

def clean_full_pdf(
    pdf_path: Path,
) -> tuple[list[str], dict]:
    """
    Reuse the exact public cleaning primitives from
    clean_and_sample_text_v2_1.py.

    Important:
    - extract_pdf -> page text
    - repeated_margins -> repeated header/footer detection
    - page_to_paragraphs -> line-wrap repair / paragraph reconstruction
    - reject_reason -> conservative paragraph filtering
    - clean_text -> light inline cleanup
    """
    pages, backend = cleaner.extract_pdf(pdf_path)

    repeated = cleaner.repeated_margins(pages)

    paragraphs: list[str] = []

    for page in pages:
        paragraphs.extend(
            cleaner.page_to_paragraphs(
                page,
                repeated,
            )
        )

    eligible: list[str] = []
    drops = Counter()
    in_references = False

    for paragraph in paragraphs:
        if cleaner.REF_HEADING_RE.match(
            paragraph.strip()
        ):
            in_references = True
            drops["reference_heading"] += 1
            continue

        reason = cleaner.reject_reason(
            paragraph,
            in_references,
        )

        if reason:
            drops[reason] += 1
            continue

        cleaned = cleaner.clean_text(
            paragraph
        ).strip()

        if wc(cleaned) >= 4:
            eligible.append(cleaned)

    meta = {
        "extractor": backend,
        "pages": len(pages),
        "raw_words": sum(
            wc(page)
            for page in pages
        ),
        "raw_paragraphs": len(paragraphs),
        "clean_words": sum(
            wc(paragraph)
            for paragraph in eligible
        ),
        "kept_paragraphs": len(eligible),
        "repeated_header_footer_lines": len(repeated),
        "drop_reasons": ";".join(
            f"{key}={value}"

```

```

        for key, value in drops.most_common()
    ),
}

return eligible, meta

def main() -> None:
    parser = argparse.ArgumentParser(
        description=(
            "Build multiple clean stylometric windows per paper. "
            "Uses clean_and_sample_text_v2_1 cleaning primitives, "
            "but preserves multiple locations instead of producing "
            "one representative sample."
        )
    )

    parser.add_argument(
        "--author-folder",
        required=True,
    )
    parser.add_argument(
        "--author-alias",
        required=True,
    )
    parser.add_argument(
        "--windows",
        type=int,
        default=5,
    )
    parser.add_argument(
        "--window-words",
        type=int,
        default=600,
    )
    parser.add_argument(
        "--output-root",
        default="data/multiwindow_v1",
    )
    parser.add_argument(
        "--manifest-root",
        default="data/multiwindow_manifests_v1",
    )
    parser.add_argument(
        "--clear",
        action="store_true",
    )

    args = parser.parse_args()

    if args.windows < 2:
        raise ValueError(
            "--windows must be at least 2"
        )

    if args.window_words < 300:
        raise ValueError(
            "--window-words should be at least 300"
        )

    source = Path(args.author_folder)

    if not source.exists():
        raise FileNotFoundError(source)

    pdfs = sorted(
        source.glob("*.pdf")
    )

    if not pdfs:
        raise RuntimeError(
            f"No PDFs found in {source}"
        )

    out_dir = (
        Path(args.output_root)
        / args.author_alias
    )

    out_dir.mkdir(
        parents=True,
        exist_ok=True,
    )

    if args.clear:
        for txt in out_dir.glob("*.txt"):
            txt.unlink()

    rows: list[dict] = []

    print("=" * 78)
    print(
        "VISKOPIC MULTI-WINDOW CORPUS BUILDER V1 - FIXED"
    )
    print("=" * 78)
    print(f"Author:          {args.author_alias}")
    print(f"Papers:          {len(pdfs)}")

```

```

print(
    f"Windows:      {args.windows} per paper"
)
print(
    f"Window words: {args.window_words}"
)
print(f"Output:      {out_dir}")
print()

for paper_index, pdf in enumerate(
    pdfs,
    start=1,
):
    print(
        f"[{paper_index:02d}/{len(pdfs):02d}] "
        f"{pdf.name[:100]}"
    )

    try:
        paragraphs, meta = clean_full_pdf(
            pdf
        )

        units = sentence_units(
            paragraphs
        )

        (
            cumulative,
            total_words,
        ) = cumulative_word_positions(
            units
        )

        minimum_required = max(
            800,
            args.window_words,
        )

        if total_words < minimum_required:
            print(
                "    SKIP: only "
                f"{total_words} usable clean words"
            )

            rows.append(
                {
                    "author": args.author_alias,
                    "paper_id": pdf.stem,
                    "pdf_file": pdf.name,
                    "status": "too_little_clean_text",
                    **meta,
                }
            )
            continue

        fractions = [
            (i + 0.5) / args.windows
            for i in range(args.windows)
        ]

        window_word_counts: list[int] = []

        for window_idx, fraction in enumerate(
            fractions,
            start=1,
        ):
            text = make_centered_window(
                units=units,
                cumulative=cumulative,
                total_words=total_words,
                center_fraction=fraction,
                target_words=args.window_words,
            )

            words_in_window = wc(text)

            if words_in_window < max(
                300,
                int(args.window_words * 0.65),
            ):
                continue

            out_name = (
                f"{pdf.stem}"
                f"__WINDOW_{window_idx:02d}.txt"
            )

            out_path = (
                out_dir
                / out_name
            )

            out_path.write_text(
                text,
                encoding="utf-8",
            )

```

```

        window_word_counts.append(
            words_in_window
        )

        rows.append(
            {
                "author": args.author_alias,
                "paper_id": pdf.stem,
                "pdf_file": pdf.name,
                "window_id": window_idx,
                "center_fraction": fraction,
                "window_file": out_name,
                "window_path": out_path.as_posix(),
                "window_words": words_in_window,
                "status": "ok",
                **meta,
            }
        )

    print(
        f"    clean={meta['clean_words']} "
        f"windows={len(window_word_counts)} "
        f>window_words={window_word_counts}"
    )

except Exception as exc:
    print(
        "    ERROR: "
        f">{type(exc).__name__}: {exc}"
    )

    rows.append(
        {
            "author": args.author_alias,
            "paper_id": pdf.stem,
            "pdf_file": pdf.name,
            "status": "error",
            "error": (
                f">{type(exc).__name__}: {exc}"
            ),
        }
    )

)

manifest_path = (
    Path(args.manifest_root)
    / (
        f">{args.author_alias}"
        "_multiwindow_v1.csv"
    )
)

manifest_path.parent.mkdir(
    parents=True,
    exist_ok=True,
)

fieldnames = [
    "author",
    "paper_id",
    "pdf_file",
    "window_id",
    "center_fraction",
    "window_file",
    "window_path",
    "window_words",
    "status",
    "extractor",
    "pages",
    "raw_words",
    "raw_paragraphs",
    "clean_words",
    "kept_paragraphs",
    "repeated_header_footer_lines",
    "drop_reasons",
    "error",
]

with manifest_path.open(
    "w",
    encoding="utf-8",
    newline="",
) as handle:
    writer = csv.DictWriter(
        handle,
        fieldnames=fieldnames,
        extrasaction="ignore",
    )
    writer.writeheader()
    writer.writerows(rows)

ok_rows = [
    row
    for row in rows
    if row.get("status") == "ok"
]

```

```

paper_counts: dict[str, int] = {}

for row in ok_rows:
    paper_id = row["paper_id"]
    paper_counts[paper_id] = (
        paper_counts.get(
            paper_id,
            0,
        )
        + 1
    )

print()
print("=" * 78)
print(
    "MULTI-WINDOW BUILD COMPLETE"
)
print("=" * 78)
print(
    "Papers with windows: "
    f"{len(paper_counts)}"
)
print(
    "Total windows:      "
    f"{len(ok_rows)}"
)
print(
    "Windows per paper:   "
    f"{sorted(set(paper_counts.values()))}"
)
print(
    f"Manifest: {manifest_path.resolve()}"
)

if __name__ == "__main__":
    main()

```

## Appendix B. `evaluate_multiwindow_authorship_v1.py`

Exact single-author multi-window evaluator created today. It holds whole candidate documents out, scores each candidate window against reference windows, takes the median document distance, and performs nested document-level genuine calibration.

```
from __future__ import annotations

import argparse
import math
from pathlib import Path

import numpy as np
import pandas as pd

import evaluate_authorship as base

def load_manifest(path: Path, author: str) -> pd.DataFrame:
    d = pd.read_csv(path)

    d = d[
        (d["author"] == author)
        & (d["status"] == "ok")
    ].copy()

    if d.empty:
        raise RuntimeError(
            f"No usable windows for {author}"
        )

    required = {
        "paper_id",
        "window_id",
        "window_path",
    }

    missing = required - set(d.columns)

    if missing:
        raise ValueError(
            "Manifest missing columns: "
            + ", ".join(sorted(missing))
        )

    return d.sort_values(
        ["paper_id", "window_id"]
    ).reset_index(drop=True)

def extract_window_features(d: pd.DataFrame) -> tuple[pd.DataFrame, dict]:
    rows = []
    meta = {}

    for i, row in d.iterrows():
        path = Path(row["window_path"])

        if not path.exists():
            raise FileNotFoundError(
                f"Window file missing: {path}"
            )

        text = path.read_text(
            encoding="utf-8",
            errors="replace",
        ).strip()

        features = base.extract_features(text)

        feature_row = {
            "paper_id": row["paper_id"],
            "window_id": int(row["window_id"]),
            "window_path": path.as_posix(),
            **features,
        }

        rows.append(feature_row)

        meta[
            (
                row["paper_id"],
                int(row["window_id"]),
            )
        ] = {
            "window_path": path.as_posix(),
        }

    print(
        f"[FEATURES {i+1:03d}/{len(d):03d}] "
        f"{row['paper_id'][:65]} "
        f"window {int(row['window_id'])}"
    )

    return pd.DataFrame(rows), meta
```

```

def numeric_feature_columns(
    features: pd.DataFrame,
) -> list[str]:
    excluded = {
        "paper_id",
        "window_id",
        "window_path",
    }

    return [
        c
        for c in features.columns
        if c not in excluded
        and c not in base.EXCLUDED_FROM_DISTANCE
        and pd.api.types.is_numeric_dtype(features[c])
    ]

def profile_window_distance(
    reference_windows: pd.DataFrame,
    candidate_window: pd.Series,
    candidate_columns: list[str],
) -> tuple[float, dict[str, float], int]:
    columns = base.usable_features(
        reference_windows,
        candidate_columns,
    )

    if len(columns) < 3:
        raise ValueError(
            "Fewer than 3 usable features"
        )

    _, _, z_scores = base.clipped_z_scores(
        reference_windows,
        candidate_window,
        columns,
    )

    distance, group_distances = (
        base.group_balanced_distance(z_scores)
    )

    return (
        float(distance),
        group_distances,
        len(columns),
    )

def score_document_raw(
    reference_windows: pd.DataFrame,
    candidate_windows: pd.DataFrame,
    candidate_columns: list[str],
) -> tuple[float, list[dict], int]:
    window_rows = []
    usable_counts = []

    for _, candidate in candidate_windows.iterrows():
        distance, group_distances, usable = (
            profile_window_distance(
                reference_windows,
                candidate,
                candidate_columns,
            )
        )

        usable_counts.append(usable)

        row = {
            "paper_id": candidate["paper_id"],
            "window_id": int(candidate["window_id"]),
            "window_path": candidate["window_path"],
            "profile_distance": distance,
            "usable_features": usable,
        }

        for group, gd in group_distances.items():
            row[f"group_{group}_distance"] = gd

        window_rows.append(row)

    distances = np.asarray(
        [
            row["profile_distance"]
            for row in window_rows
        ],
        dtype=float,
    )

    document_distance = float(
        np.median(distances)
    )

    typical_usable = int(

```

```

        round(float(np.median(usable_counts)))
    )

    return (
        document_distance,
        window_rows,
        typical_usable,
    )

def reference_document_distances(
    reference_windows: pd.DataFrame,
    candidate_columns: list[str],
) -> list[float]:
    """
    Nested document-level leave-one-out calibration.

    Each reference paper is held out in full. No window from that paper
    is allowed into the profile used to score it.
    """
    distances = []

    ref_papers = sorted(
        reference_windows["paper_id"].unique()
    )

    for heldout_paper in ref_papers:
        train = reference_windows[
            reference_windows["paper_id"]
            != heldout_paper
        ].copy()

        held = reference_windows[
            reference_windows["paper_id"]
            == heldout_paper
        ].copy()

        if (
            train["paper_id"].nunique() < 2
            or held.empty
        ):
            continue

        try:
            doc_distance, _, _ = (
                score_document_raw(
                    train,
                    held,
                    candidate_columns,
                )
            )
        except Exception:
            continue

        distances.append(
            float(doc_distance)
        )

    return distances

def quantiles(values: list[float]) -> dict[str, float]:
    arr = np.asarray(values, dtype=float)

    return {
        "window_distance_min": float(np.min(arr)),
        "window_distance_q25": float(np.quantile(arr, 0.25)),
        "window_distance_median": float(np.median(arr)),
        "window_distance_q75": float(np.quantile(arr, 0.75)),
        "window_distance_max": float(np.max(arr)),
        "window_distance_iqr": float(
            np.quantile(arr, 0.75)
            - np.quantile(arr, 0.25)
        ),
        "window_distance_range": float(
            np.max(arr) - np.min(arr)
        ),
    }

def main() -> None:
    parser = argparse.ArgumentParser(
        description=(
            "Document-group-aware multi-window authorship evaluation. "
            "Holds out every window of a paper together."
        )
    )

    parser.add_argument(
        "--manifest",
        required=True,
    )
    parser.add_argument(
        "--author",
        required=True,
    )
    parser.add_argument(

```

```

    "--output-dir",
    default="evaluation_multiwindow_v1",
)

args = parser.parse_args()

manifest = load_manifest(
    Path(args.manifest),
    args.author,
)

paper_counts = (
    manifest.groupby("paper_id")
    .size()
    .sort_values()
)

if len(paper_counts) < 5:
    raise ValueError(
        "At least 5 papers are required."
    )

print("=" * 78)
print("VISKOPIC MULTI-WINDOW CONSISTENCY EVALUATOR V1")
print("=" * 78)
print(f"Author: {args.author}")
print(f"Papers: {len(paper_counts)}")
print(f"Windows: {len(manifest)}")
print(
    "Windows per paper: "
    f"{sorted(paper_counts.unique())}"
)
print()

features, _ = extract_window_features(
    manifest
)

candidate_columns = numeric_feature_columns(
    features
)

papers = sorted(
    features["paper_id"].unique()
)

document_results = []
all_window_results = []

for i, paper in enumerate(papers, start=1):
    print()
    print(
        f"[DOCUMENT {i:02d}/{len(papers):02d}] "
        f"{paper[:95]}"
    )

    candidate = features[
        features["paper_id"] == paper
    ].copy()

    reference = features[
        features["paper_id"] != paper
    ].copy()

    reference_doc_count = int(
        reference["paper_id"].nunique()
    )

    (
        raw_doc_distance,
        window_rows,
        usable_features,
    ) = score_document_raw(
        reference,
        candidate,
        candidate_columns,
    )

    calibration = reference_document_distances(
        reference,
        candidate_columns,
    )

    if not calibration:
        raise RuntimeError(
            f"No calibration distances for {paper}"
        )

    typical = float(
        np.median(
            np.asarray(calibration, dtype=float)
        )
    )

    typical = max(typical, 1e-8)

```

```

relative = (
    raw_doc_distance / typical
)

similarity = (
    100.0
    * math.exp(
        -0.5 * relative ** 2
    )
)

window_distances = [
    float(r["profile_distance"])
    for r in window_rows
]

stability = quantiles(
    window_distances
)

# Candidate-window similarity using the same document-level
# genuine calibration, useful for diagnostics only.
for row in window_rows:
    wr = dict(row)
    wr["reference_document_count"] = reference_doc_count
    wr["typical_genuine_document_distance"] = typical
    wr["window_relative_to_document_typical"] = (
        wr["profile_distance"] / typical
    )
    wr["window_similarity_diagnostic"] = (
        100.0
        * math.exp(
            -0.5
            * wr["window_relative_to_document_typical"] ** 2
        )
    )
    all_window_results.append(wr)

result = {
    "paper_id": paper,
    "candidate_windows": len(candidate),
    "reference_document_count": reference_doc_count,
    "reference_window_count": len(reference),
    "candidate_features": len(candidate_columns),
    "usable_features": usable_features,
    "profile_distance": raw_doc_distance,
    "typical_genuine_distance": typical,
    "relative_distance": relative,
    "similarity_score": similarity,
    "calibration_document_count": len(calibration),
    **stability,
}

document_results.append(result)

print(
    f"    windows={len(candidate)} "
    f"refs={reference_doc_count} papers / {len(reference)} windows"
)
print(
    f"    D={raw_doc_distance:.3f} "
    f"typical={typical:.3f} "
    f"R={relative:.3f} "
    f"score={similarity:.1f}"
)
print(
    f"    window D range="
    f"{stability['window_distance_min']:.3f}"
    f"-{stability['window_distance_max']:.3f} "
    f"IQR={stability['window_distance_iqr']:.3f}"
)

out_dir = Path(args.output_dir)
out_dir.mkdir(
    parents=True,
    exist_ok=True,
)

docs = pd.DataFrame(
    document_results
).sort_values(
    "similarity_score",
    ascending=False,
)

windows = pd.DataFrame(
    all_window_results
).sort_values(
    ["paper_id", "window_id"]
)

docs_path = (
    out_dir
    / "document_results.csv"
)
windows_path = (
    out_dir

```

```

    / "window_results.csv"
)

docs.to_csv(
    docs_path,
    index=False,
)
windows.to_csv(
    windows_path,
    index=False,
)

print()
print("=" * 78)
print("MULTI-WINDOW AUTHOR SUMMARY")
print("=" * 78)
print(f"Papers tested:          {len(docs)}")
print(
    "Mean similarity:          "
    f"{docs['similarity_score'].mean():.2f}"
)
print(
    "Median similarity:         "
    f"{docs['similarity_score'].median():.2f}"
)
print(
    "Minimum similarity:        "
    f"{docs['similarity_score'].min():.2f}"
)
print(
    "Maximum similarity:        "
    f"{docs['similarity_score'].max():.2f}"
)
print(
    "Mean relative distance:    "
    f"{docs['relative_distance'].mean():.3f}"
)
print(
    "Median relative dist:      "
    f"{docs['relative_distance'].median():.3f}"
)
print(
    "Median window IQR:         "
    f"{docs['window_distance_iqr'].median():.3f}"
)
print(
    "Median window range:       "
    f"{docs['window_distance_range'].median():.3f}"
)

print()
print("LOWEST 5")
print(
    docs.nsmallest(
        5,
        "similarity_score",
    )[
        [
            "paper_id",
            "similarity_score",
            "relative_distance",
            "window_distance_iqr",
            "window_distance_range",
        ]
    ].to_string(index=False)
)

print()
print("HIGHEST 5")
print(
    docs.nlargest(
        5,
        "similarity_score",
    )[
        [
            "paper_id",
            "similarity_score",
            "relative_distance",
            "window_distance_iqr",
            "window_distance_range",
        ]
    ].to_string(index=False)
)

print()
print(f"Document results: {docs_path.resolve()}")
print(f"Window results:   {windows_path.resolve()}")
print()
print(
    "Important: all windows from the candidate paper are held out "
    "together. No candidate-paper window enters its own profile."
)

if __name__ == "__main__":
    main()

```

## Appendix C. evaluate\_multiwindow\_discrimination\_v1.py

Exact multi-author discrimination evaluator created today. It scores every candidate document against every profile, uses genuine profile calibration, produces verification metrics and author rankings, and saves comparison CSVs.

```
from __future__ import annotations

import argparse
import math
from pathlib import Path

import numpy as np
import pandas as pd

import evaluate_multiwindow_authorship_v1 as mw

def load_manifest_auto(path: Path) -> tuple[str, pd.DataFrame]:
    d = pd.read_csv(path)

    if "author" not in d.columns:
        raise ValueError(f"{path}: missing 'author' column")

    d = d[d["status"] == "ok"].copy()

    authors = sorted(d["author"].dropna().astype(str).unique())

    if len(authors) != 1:
        raise ValueError(
            f"{path}: expected exactly one author, found {authors}"
        )

    author = authors[0]

    required = {
        "paper_id",
        "window_id",
        "window_path",
    }

    missing = required - set(d.columns)

    if missing:
        raise ValueError(
            f"{path}: missing columns {sorted(missing)}"
        )

    d["author"] = author

    return (
        author,
        d.sort_values(
            ["paper_id", "window_id"]
        ).reset_index(drop=True),
    )

def extract_features_for_author(
    author: str,
    manifest: pd.DataFrame,
) -> pd.DataFrame:
    feature_rows, _ = mw.extract_window_features(
        manifest
    )

    feature_rows.insert(
        0,
        "author",
        author,
    )

    return feature_rows

def auc_from_scores(
    y_true: np.ndarray,
    scores: np.ndarray,
) -> float:
    """
    ROC-AUC using the Mann-Whitney rank formulation.
    Higher score = more likely same-author.
    """
    y_true = np.asarray(y_true, dtype=int)
    scores = np.asarray(scores, dtype=float)

    pos = y_true == 1
    neg = y_true == 0

    n_pos = int(pos.sum())
    n_neg = int(neg.sum())
```

```

if n_pos == 0 or n_neg == 0:
    return float("nan")

ranks = pd.Series(scores).rank(
    method="average"
).to_numpy()

rank_sum_pos = float(
    ranks[pos].sum()
)

auc = (
    rank_sum_pos
    - n_pos * (n_pos + 1) / 2.0
) / (
    n_pos * n_neg
)

return float(auc)

def threshold_metrics(
    y_true: np.ndarray,
    scores: np.ndarray,
) -> tuple[dict, pd.DataFrame]:
    """
    Higher similarity means 'same author'.
    Returns EER-like operating point and best balanced-accuracy point.
    """
    y_true = np.asarray(
        y_true,
        dtype=int,
    )
    scores = np.asarray(
        scores,
        dtype=float,
    )

    thresholds = np.unique(
        np.concatenate(
            [
                scores,
                [scores.min() - 1e-9],
                [scores.max() + 1e-9],
            ]
        )
    )

    rows = []

    pos_n = max(
        1,
        int((y_true == 1).sum()),
    )
    neg_n = max(
        1,
        int((y_true == 0).sum()),
    )

    for threshold in thresholds:
        pred = (
            scores >= threshold
        ).astype(int)

        tp = int(
            ((pred == 1) & (y_true == 1)).sum()
        )
        fn = int(
            ((pred == 0) & (y_true == 1)).sum()
        )
        fp = int(
            ((pred == 1) & (y_true == 0)).sum()
        )
        tn = int(
            ((pred == 0) & (y_true == 0)).sum()
        )

        tpr = tp / pos_n
        fnr = fn / pos_n
        fpr = fp / neg_n
        tnr = tn / neg_n

        balanced_accuracy = (
            tpr + tnr
        ) / 2.0

        rows.append(
            {
                "threshold": float(threshold),
                "tpr": tpr,
                "fnr": fnr,
                "fpr": fpr,
                "tnr": tnr,
                "balanced_accuracy": balanced_accuracy,
                "eer_gap": abs(fpr - fnr),
            }
        )
    )

```

```

curve = pd.DataFrame(rows)

eer_row = curve.sort_values(
    [
        "eer_gap",
        "threshold",
    ],
    ascending=[
        True,
        False,
    ],
).iloc[0]

best_row = curve.sort_values(
    [
        "balanced_accuracy",
        "threshold",
    ],
    ascending=[
        False,
        False,
    ],
).iloc[0]

summary = {
    "eer": float(
        (
            eer_row["fpr"]
            + eer_row["fnr"]
        )
        / 2.0
    ),
    "eer_threshold": float(
        eer_row["threshold"]
    ),
    "best_balanced_accuracy": float(
        best_row["balanced_accuracy"]
    ),
    "best_balanced_threshold": float(
        best_row["threshold"]
    ),
}

return summary, curve

def calibrate_profile(
    profile_windows: pd.DataFrame,
    candidate_columns: list[str],
) -> tuple[float, list[float]]:
    distances = mw.reference_document_distances(
        profile_windows,
        candidate_columns,
    )

    if not distances:
        raise RuntimeError(
            "Could not calculate genuine document calibration distances"
        )

    typical = float(
        np.median(
            np.asarray(
                distances,
                dtype=float,
            )
        )
    )

    return max(
        typical,
        1e-8,
    ), distances

def main() -> None:
    parser = argparse.ArgumentParser(
        description=(
            "Evaluate multi-window same-author vs different-author "
            "discrimination with whole-document holdout."
        )
    )

    parser.add_argument(
        "--manifest",
        action="append",
        required=True,
        help=(
            "Repeat once per author. Example: "
            "--manifest data/.../author_002_multiwindow_v1.csv "
            "--manifest data/.../author_007_multiwindow_v1.csv"
        ),
    )

    parser.add_argument(
        "--output-dir",

```

```

    default="evaluation_results_multiwindow_discrimination_v1",
)

args = parser.parse_args()

if len(args.manifest) < 2:
    raise ValueError(
        "At least two author manifests are required."
    )

manifest_by_author = {}
features_by_author = {}

print("=" * 82)
print(
    "VISKOPIC MULTI-WINDOW DISCRIMINATION EVALUATOR V1"
)
print("=" * 82)

for manifest_arg in args.manifest:
    path = Path(manifest_arg)

    author, manifest = load_manifest_auto(
        path
    )

    if author in manifest_by_author:
        raise ValueError(
            f"Duplicate author manifest: {author}"
        )

    paper_count = int(
        manifest["paper_id"].nunique()
    )

    if paper_count < 4:
        raise ValueError(
            f"{author}: at least 4 papers required, found {paper_count}"
        )

    manifest_by_author[author] = manifest

    print()
    print(
        f"{author}: {paper_count} papers / "
        f"{len(manifest)} windows"
    )

    features_by_author[
        author
    ] = extract_features_for_author(
        author,
        manifest,
    )

authors = sorted(
    features_by_author
)

all_features = pd.concat(
    [
        features_by_author[a]
        for a in authors
    ],
    ignore_index=True,
)

candidate_columns = mw.numeric_feature_columns(
    all_features.drop(
        columns=["author"]
    )
)

print()
print(
    f"Authors: {len(authors)}"
)
print(
    f"Candidate feature columns: {len(candidate_columns)}"
)
print()

# Full-profile calibrations are reusable for different-author comparisons.
full_profile_calibration = {}

for profile_author in authors:
    profile_windows = (
        features_by_author[
            profile_author
        ].drop(
            columns=["author"]
        )
    )

    typical, distances = calibrate_profile(
        profile_windows,
        candidate_columns,

```

```

)
full_profile_calibration[
    profile_author
] = {
    "typical": typical,
    "distances": distances,
}

print(
    f"[CALIBRATE] {profile_author}: "
    f"papers={profile_windows['paper_id'].nunique()} "
    f"typical genuine D={typical:.3f}"
)

comparisons = []
candidate_total = sum(
    int(
        features_by_author[a][
            "paper_id"
        ].nunique()
    )
    for a in authors
)

candidate_counter = 0

for candidate_author in authors:
    candidate_author_features = (
        features_by_author[
            candidate_author
        ]
    )

    candidate_papers = sorted(
        candidate_author_features[
            "paper_id"
        ].unique()
    )

    for candidate_paper in candidate_papers:
        candidate_counter += 1

        candidate_windows = (
            candidate_author_features[
                candidate_author_features[
                    "paper_id"
                ]
                == candidate_paper
            ]
            .drop(
                columns=["author"]
            )
            .copy()
        )

        print()
        print(
            f"[CANDIDATE {candidate_counter:03d}/{candidate_total:03d}] "
            f"{candidate_author} :: {candidate_paper[:80]}"
        )

        for profile_author in authors:
            same_author = (
                profile_author
                == candidate_author
            )

            profile_all = (
                features_by_author[
                    profile_author
                ]
                .drop(
                    columns=["author"]
                )
                .copy()
            )

            if same_author:
                # Hold out the entire candidate paper.
                reference_windows = (
                    profile_all[
                        profile_all[
                            "paper_id"
                        ]
                        != candidate_paper
                    ]
                    .copy()
                )

            typical, calibration_distances = (
                calibrate_profile(
                    reference_windows,
                    candidate_columns,

```

```

    )

else:
    # Candidate belongs to another author, therefore none of its
    # windows are in this profile.
    reference_windows = profile_all

    typical = (
        full_profile_calibration[
            profile_author
        ]["typical"]
    )

    calibration_distances = (
        full_profile_calibration[
            profile_author
        ]["distances"]
    )

    (
        raw_distance,
        window_rows,
        usable_features,
    ) = mw.score_document_raw(
        reference_windows,
        candidate_windows,
        candidate_columns,
    )

    relative_distance = (
        raw_distance
        / typical
    )

    similarity_score = (
        100.0
        * math.exp(
            -0.5
            * relative_distance ** 2
        )
    )

    candidate_window_distances = [
        float(
            row[
                "profile_distance"
            ]
        )
        for row in window_rows
    ]

    q25 = float(
        np.quantile(
            candidate_window_distances,
            0.25,
        )
    )

    q75 = float(
        np.quantile(
            candidate_window_distances,
            0.75,
        )
    )

    comparisons.append(
        {
            "candidate_author": candidate_author,
            "candidate_paper": candidate_paper,
            "profile_author": profile_author,
            "same_author": int(
                same_author
            ),
            "candidate_window_count": len(
                candidate_windows
            ),
            "reference_document_count": int(
                reference_windows[
                    "paper_id"
                ].nunique()
            ),
            "reference_window_count": len(
                reference_windows
            ),
            "usable_features": usable_features,
            "profile_distance": raw_distance,
            "typical_genuine_distance": typical,
            "relative_distance": relative_distance,
            "similarity_score": similarity_score,
            "window_distance_min": float(
                np.min(
                    candidate_window_distances
                )
            ),
            "window_distance_median": float(
                np.median(
                    candidate_window_distances

```

```

        ),
        "window_distance_max": float(
            np.max(
                candidate_window_distances
            )
        ),
        "window_distance_iqr": (
            q75 - q25
        ),
        "calibration_document_count": len(
            calibration_distances
        ),
    }
)

label = (
    "SAME"
    if same_author
    else "DIFF"
)

print(
    f"    {label:4s} vs {profile_author}: "
    f"D={raw_distance:.3f} "
    f"R={relative_distance:.3f} "
    f"score={similarity_score:.1f}"
)

results = pd.DataFrame(
    comparisons
)

y_true = results[
    "same_author"
].to_numpy(
    dtype=int
)

scores = results[
    "similarity_score"
].to_numpy(
    dtype=float
)

auc = auc_from_scores(
    y_true,
    scores,
)

threshold_summary, curve = (
    threshold_metrics(
        y_true,
        scores,
    )
)

rankings = (
    results.sort_values(
        [
            "candidate_author",
            "candidate_paper",
            "similarity_score",
        ],
        ascending=[
            True,
            True,
            False,
        ],
    )
    .copy()
)

rankings["rank"] = (
    rankings.groupby(
        [
            "candidate_author",
            "candidate_paper",
        ]
    )[
        "similarity_score"
    ]
    .rank(
        method="min",
        ascending=False,
    )
)

true_rows = rankings[
    rankings["same_author"] == 1
].copy()

top1_accuracy = float(
    (
        true_rows["rank"] == 1
    ).mean()
)

```

```

mrr = float(
    (
        1.0
        / true_rows["rank"]
    ).mean()
)

same_scores = results.loc[
    results["same_author"] == 1,
    "similarity_score",
]

diff_scores = results.loc[
    results["same_author"] == 0,
    "similarity_score",
]

same_relative = results.loc[
    results["same_author"] == 1,
    "relative_distance",
]

diff_relative = results.loc[
    results["same_author"] == 0,
    "relative_distance",
]

author_summary = (
    results.groupby(
        [
            "candidate_author",
            "same_author",
        ]
    )
    .agg(
        comparisons=(
            "similarity_score",
            "size",
        ),
        mean_similarity=(
            "similarity_score",
            "mean",
        ),
        median_similarity=(
            "similarity_score",
            "median",
        ),
        mean_relative_distance=(
            "relative_distance",
            "mean",
        ),
        median_relative_distance=(
            "relative_distance",
            "median",
        ),
    )
    .reset_index()
)

out_dir = Path(
    args.output_dir
)

out_dir.mkdir(
    parents=True,
    exist_ok=True,
)

results.to_csv(
    out_dir
    / "all_comparisons.csv",
    index=False,
)

rankings.to_csv(
    out_dir
    / "document_rankings.csv",
    index=False,
)

curve.to_csv(
    out_dir
    / "threshold_curve.csv",
    index=False,
)

author_summary.to_csv(
    out_dir
    / "author_summary.csv",
    index=False,
)

metrics = pd.DataFrame(
    [
        {
            "authors": len(authors),

```

```

        "candidate_documents": candidate_total,
        "comparisons": len(results),
        "same_author_comparisons": int(
            (y_true == 1).sum()
        ),
        "different_author_comparisons": int(
            (y_true == 0).sum()
        ),
        "roc_auc": auc,
        "eer": threshold_summary[
            "eer"
        ],
        "eer_threshold": threshold_summary[
            "eer_threshold"
        ],
        "best_balanced_accuracy": threshold_summary[
            "best_balanced_accuracy"
        ],
        "best_balanced_threshold": threshold_summary[
            "best_balanced_threshold"
        ],
        "top1_author_accuracy": top1_accuracy,
        "mrr": mrr,
        "same_mean_similarity": float(
            same_scores.mean()
        ),
        "different_mean_similarity": float(
            diff_scores.mean()
        ),
        "same_median_similarity": float(
            same_scores.median()
        ),
        "different_median_similarity": float(
            diff_scores.median()
        ),
        "same_mean_relative_distance": float(
            same_relative.mean()
        ),
        "different_mean_relative_distance": float(
            diff_relative.mean()
        ),
    },
]
)

metrics.to_csv(
    out_dir
    / "metrics.csv",
    index=False,
)

print()
print("=" * 82)
print(
    "MULTI-WINDOW DISCRIMINATION SUMMARY"
)
print("=" * 82)
print(
    f"Authors:                                {len(authors)}"
)
print(
    f"Candidate documents:                    {candidate_total}"
)
print(
    f"Same-author comparisons:                {(y_true == 1).sum()}"
)
print(
    f"Different-author comparisons:           {(y_true == 0).sum()}"
)
print()
print(
    f"ROC-AUC:                                {auc:.3f}"
)
print(
    f"EER:                                    {threshold_summary['eer']:.3f}"
)
print(
    f"Best balanced accuracy:                  "
    f"{threshold_summary['best_balanced_accuracy']:.3f}"
)
print(
    f"Top-1 author identification:             {top1_accuracy:.3f}"
)
print(
    f"MRR:                                    {mrr:.3f}"
)
print()
print(
    f"Same-author mean similarity:             "
    f"{same_scores.mean():.2f}"
)
print(
    f"Different-author mean sim.:             "
    f"{diff_scores.mean():.2f}"
)
print(
    f"Same-author median similarity:          "

```

```

        f"{same_scores.median():.2f}"
    )
    print(
        "Different-author median sim.: "
        f"{diff_scores.median():.2f}"
    )
    print()
    print(
        "Same-author mean relative D: "
        f"{same_relative.mean():.3f}"
    )
    print(
        "Different-author mean rel. D: "
        f"{diff_relative.mean():.3f}"
    )

    print()
    print("TRUE-AUTHOR RANK FAILURES")
    failures = true_rows[
        true_rows["rank"] != 1
    ]
    [
        [
            "candidate_author",
            "candidate_paper",
            "similarity_score",
            "rank",
        ]
    ]

    if failures.empty:
        print("None")
    else:
        print(
            failures.to_string(
                index=False
            )
        )

    print()
    print(
        f"Results: {out_dir.resolve()}"
    )

if __name__ == "__main__":
    main()

```

## Appendix D. Key baseline scoring code

The following excerpts are the core of the current group-balanced single-window scoring implementation that V1.1 builds on.

```
EXCLUDED_FROM_DISTANCE = {
    "word_count",
    "sentence_count",
    "paragraph_count",
    "character_count",
    "character_count_no_spaces",
    "unique_word_count",
    "sentence_length_min",
    "sentence_length_max",
    "avg_paragraph_length",
    "paragraph_length_std",
}

FEATURE_GROUPS = {
    "sentence_structure": [
        "avg_sentence_length",
        "sentence_length_std",
        "long_sentences_ratio",
        "short_sentences_ratio",
        "unique_sentence_opening_ratio",
        "repeated_sentence_opening_ratio",
    ],
    "vocabulary": [
        "avg_word_length",
        "word_length_std",
        "type_token_ratio",
        "root_type_token_ratio",
        "corrected_type_token_ratio",
        "hapax_ratio",
        "hapax_unique_ratio",
        "dislegomena_ratio",
        "repeated_word_ratio",
        "long_words_ratio",
        "very_long_words_ratio",
    ],
    "readability": [
        "avg_syllables_per_word",
        "complex_word_ratio",
        "flesch_reading_ease",
        "flesch_kincaid_grade",
        "gunning_fog_index",
    ],
    "punctuation": [
        "commas_per_100_words",
        "semicolons_per_100_words",
        "colons_per_100_words",
        "question_marks_per_100_words",
        "exclamation_marks_per_100_words",
        "parentheses_per_100_words",
        "hyphens_per_100_words",
    ],
    "pronouns_and_stance": [
        "first_person_per_100_words",
        "second_person_per_100_words",
        "third_person_per_100_words",
        "contractions_per_100_words",
    ],
}

def assign_group(feature: str) -> str:
    for group, features in FEATURE_GROUPS.items():
        if feature in features:
            return group

    if feature.startswith("function_"):
        return "function_words"

    if feature.startswith("transition_"):
        return "transitions"

    if feature.startswith("sentence_opening_"):
        return "sentence_openings"

    return "other"

def numeric_feature_columns(profile: pd.DataFrame) -> list[str]:
    return [
        column
        for column in profile.columns
        if column != "essay"
        and column not in EXCLUDED_FROM_DISTANCE
        and pd.api.types.is_numeric_dtype(profile[column])
    ]

def usable_features(
    profile: pd.DataFrame,
```

```

    columns: list[str],
) -> list[str]:
    """
    Keep only features with enough variation to be meaningful.

    With very small author profiles, nearly constant features can create
    absurd z-scores because the standard deviation is close to zero.
    """
    stds = profile[columns].std(ddof=1)
    means = profile[columns].mean()

    usable: list[str] = []

    for column in columns:
        std = stds[column]
        mean = means[column]

        if pd.isna(std) or std <= 0:
            continue

        absolute_floor = 0.02
        relative_floor = max(abs(mean) * 0.05, absolute_floor)

        if std < relative_floor:
            continue

        usable.append(column)

    return usable

def clipped_z_scores(
    reference: pd.DataFrame,
    sample: pd.Series,
    columns: list[str],
) -> tuple[pd.Series, pd.Series, pd.Series]:
    means = reference[columns].mean()
    stds = reference[columns].std(ddof=1)

    z_scores = (sample[columns] - means) / stds
    z_scores = z_scores.clip(lower=-4.0, upper=4.0)

    return means, stds, z_scores

def group_balanced_distance(
    z_scores: pd.Series,
) -> tuple[float, dict[str, float]]:
    """
    Calculate one distance per conceptual feature group, then average
    those group distances.

    This prevents large groups, such as function words or transitions,
    from dominating simply because they contain more columns.
    """
    group_distances: dict[str, float] = {}

    groups = sorted({assign_group(feature) for feature in z_scores.index})

    for group in groups:
        group_features = [
            feature
            for feature in z_scores.index
            if assign_group(feature) == group
        ]

        values = (
            z_scores[group_features]
            .replace([np.inf, -np.inf], np.nan)
            .dropna()
        )

        if values.empty:
            continue

        distance = float(
            np.sqrt(np.mean(np.square(values)))
        )

        group_distances[group] = distance

    if not group_distances:
        raise ValueError("No valid feature groups were available.")

    overall_distance = float(np.mean(list(group_distances.values())))

    return overall_distance, group_distances

def standardised_distance(
    reference: pd.DataFrame,
    sample: pd.Series,
    columns: list[str],
) -> tuple[
    float,
    pd.Series,
    pd.Series,

```

```

    pd.Series,
    dict[str, float],
]:
    means, stds, z_scores = clipped_z_scores(
        reference,
        sample,
        columns,
    )

    distance, group_distances = group_balanced_distance(z_scores)

    return distance, means, stds, z_scores, group_distances

def leave_one_out_distances(
    profile: pd.DataFrame,
    candidate_columns: list[str],
) -> np.ndarray:
    distances: list[float] = []

    for index in profile.index:
        training = profile.drop(index=index)
        held_out = profile.loc[index]

        columns = usable_features(training, candidate_columns)

        if len(columns) < 3:
            continue

        _, _, z_scores = clipped_z_scores(
            training,
            held_out,
            columns,
        )

        try:
            distance, _ = group_balanced_distance(z_scores)
        except ValueError:
            continue

        distances.append(distance)

    return np.asarray(distances, dtype=float)

def similarity_score(
    new_distance: float,
    reference_distances: np.ndarray,
) -> tuple[float, float]:
    if len(reference_distances) == 0:
        return float("nan"), float("nan")

    median_distance = float(np.median(reference_distances))
    median_distance = max(median_distance, 1e-8)

    relative_distance = new_distance / median_distance

    score = 100.0 * math.exp(
        -0.5 * relative_distance ** 2
    )

    return (
        max(0.0, min(100.0, score)),
        relative_distance,
    )

```

## Appendix E. Calibration-comparison prototype for tomorrow

This is a starting implementation sketch, not a validated final model. It is intentionally separated from today's frozen production baseline.

```
from __future__ import annotations

import argparse
import math
from pathlib import Path

import numpy as np
import pandas as pd

def auc_rank(y_true, values, *, lower_is_better):
    y = np.asarray(y_true, dtype=int)
    x = np.asarray(values, dtype=float)

    # Convert to "higher = more same-author" for rank AUC.
    s = -x if lower_is_better else x

    ranks = pd.Series(s).rank(method="average").to_numpy()
    pos = y == 1
    neg = y == 0

    n_pos = int(pos.sum())
    n_neg = int(neg.sum())

    return float(
        (
            ranks[pos].sum()
            - n_pos * (n_pos + 1) / 2.0
        )
        / (n_pos * n_neg)
    )

def threshold_metrics(y_true, values, *, lower_is_better):
    y = np.asarray(y_true, dtype=int)
    x = np.asarray(values, dtype=float)

    thresholds = np.unique(
        np.concatenate(
            [x, [x.min() - 1e-9], [x.max() + 1e-9]]
        )
    )

    rows = []

    for threshold in thresholds:
        if lower_is_better:
            pred = (x <= threshold).astype(int)
        else:
            pred = (x >= threshold).astype(int)

        tp = ((pred == 1) & (y == 1)).sum()
        fn = ((pred == 0) & (y == 1)).sum()
        fp = ((pred == 1) & (y == 0)).sum()
        tn = ((pred == 0) & (y == 0)).sum()

        tpr = tp / max(1, (y == 1).sum())
        fnr = fn / max(1, (y == 1).sum())
        fpr = fp / max(1, (y == 0).sum())
        tnr = tn / max(1, (y == 0).sum())

        rows.append(
            {
                "threshold": threshold,
                "fnr": fnr,
                "fpr": fpr,
                "balanced_accuracy": (tpr + tnr) / 2.0,
                "eer_gap": abs(fpr - fnr),
            }
        )

    curve = pd.DataFrame(rows)
    eer_row = curve.sort_values("eer_gap").iloc[0]
    best = curve.sort_values(
        "balanced_accuracy",
        ascending=False,
    ).iloc[0]

    return {
        "eer": float((eer_row.fnr + eer_row.fpr) / 2.0),
        "best_balanced_accuracy": float(best.balanced_accuracy),
    }

def shrunk_typical(
    typical_genuine_distance,
```

```

reference_document_count,
global_typical,
k,
):
    n = float(reference_document_count)
    weight = n / (n + float(k))

    return (
        weight * typical_genuine_distance
        + (1.0 - weight) * global_typical
    )

def evaluate_metric(df, column, *, lower_is_better):
    y = df["same_author"].astype(int).to_numpy()
    x = df[column].astype(float).to_numpy()

    return {
        "auc": auc_rank(
            y,
            x,
            lower_is_better=lower_is_better,
        ),
        **threshold_metrics(
            y,
            x,
            lower_is_better=lower_is_better,
        ),
    }

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--comparisons", required=True)
    parser.add_argument(
        "--k",
        type=float,
        nargs="*",
        default=[2, 5, 10, 20],
    )
    args = parser.parse_args()

    d = pd.read_csv(args.comparisons)

    # Control A: raw distance
    print("RAW D")
    print(
        evaluate_metric(
            d,
            "profile_distance",
            lower_is_better=True,
        )
    )

    # Control B: current relative distance
    print("CURRENT R")
    print(
        evaluate_metric(
            d,
            "relative_distance",
            lower_is_better=True,
        )
    )

    # Exploratory global centre.
    # Better final validation should avoid selecting k on the same test.
    author_calibration = (
        d[d["same_author"] == 1]
        .groupby("profile_author")[
            "typical_genuine_distance"
        ]
        .median()
    )

    global_typical = float(
        author_calibration.median()
    )

    for k in args.k:
        out = d.copy()

        out["shrunk_typical"] = out.apply(
            lambda row: shrunk_typical(
                row["typical_genuine_distance"],
                row["reference_document_count"],
                global_typical,
                k,
            ),
            axis=1,
        )

        out["shrunk_R"] = (
            out["profile_distance"]
            / out["shrunk_typical"]
        )

        print(f"SHRUNK R k={k}")

```

```
        print(
            evaluate_metric(
                out,
                "shrunk_R",
                lower_is_better=True,
            )
        )

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

## Appendix F. Reproducibility commands and decision log

### F.1 Core commands used in the multi-window stage

```
# Build three ~550-word windows
py build_multiwindow_corpus_v1_fixed.py \
  --author-folder data/human_essays/author_007 \
  --author-alias author_007 \
  --windows 3 \
  --window-words 550 \
  --clear

# Single-author multi-window evaluation
py evaluate_multiwindow_authorship_v1.py \
  --manifest data/multiwindow_manifests_v1/author_007_multiwindow_v1.csv \
  --author author_007 \
  --output-dir evaluation_results_author007_multiwindow_v1

# Five-author discrimination
py evaluate_multiwindow_discrimination_v1.py \
  --manifest data/multiwindow_manifests_v1/author_004_multiwindow_v1_qc.csv \
  --manifest data/multiwindow_manifests_v1/author_007_multiwindow_v1_qc.csv \
  --manifest data/multiwindow_manifests_v1/author_008_multiwindow_v1_qc.csv \
  --manifest data/multiwindow_manifests_v1/author_009_multiwindow_v1_qc.csv \
  --manifest data/multiwindow_manifests_v1/author_010_multiwindow_v1_qc.csv \
  --output-dir evaluation_results_multiwindow_5author_v1
```

### F.2 Clean-word QC rule

```
# Keep only papers with enough clean text for 3 x 550 design
good = (
    d.loc[d["status"].eq("ok")]
    .groupby("paper_id")["clean_words"]
    .first()
)

good = good[good >= 1650].index
```

### F.3 Decision log

| Decision                           | Status                                |
|------------------------------------|---------------------------------------|
| AI detector work                   | Paused; consistency pipeline first    |
| Mixture models                     | Future R&D                            |
| Single arbitrary 1,500-word sample | Rejected as too sampling-sensitive    |
| Multi-window representation        | Provisionally adopted                 |
| Windows                            | 3 distributed windows                 |
| Target window length               | ~550 words                            |
| Minimum clean text                 | 1,650 words                           |
| Candidate aggregation              | Median raw window distance            |
| Leakage control                    | Hold out whole document               |
| Feature/scoring mechanics          | Frozen during validation              |
| Current calibration                | Under review tomorrow                 |
| Repo refactor                      | Deferred until architecture is frozen |

### F.4 Source artefacts from today's session

- Pasted text(8).txt - controlled light-clean experiment output.
- Pasted text(9).txt - Author 007 multi-window consistency evaluator output.
- Pasted text(10).txt - two-author (004 vs 007) discrimination run.
- Pasted text(20260810-160148).txt - QC counts and complete five-author discrimination run.
- System diagram image - broader Viskopic architecture discussion.
- build\_multiwindow\_corpus\_v1\_fixed.py, evaluate\_multiwindow\_authorship\_v1.py, evaluate\_multiwindow\_discrimination\_v1.py - code written today.

### End-of-day checkpoint

**CHECKPOINT:** The strongest conclusion today is not that the baseline is finished. It is that the representation now looks coherent enough to isolate the next weakness. Multi-window stylometry preserves genuine signal, but the current per-author calibration can make broad/small profiles too permissive. Tomorrow starts exactly there.