Omniverso: supplementary information

Mathematics, data tables, evaluation protocol, code and computational budget

Danil Kotelnikov

12 October 2026

This document accompanies the manuscript "Omniverso: partially latent flow matching for de novo antibody and nanobody design". Section numbers beginning with S refer to this document. Section numbers without a prefix refer to the manuscript. The code package named in Section S4 is the only other file the work needs.

Contents

S1 Mathematical supplement 5
S1.1 Notation and conventions 5
S1.2 Geometry and gauges 6
S1.3 Scores, diffusion reversal and flow likelihood 10
S1.4 Transport and particle limits 15
S1.5 Decision statistics and calibration 25
S1.6 Representation floor and predictive information 31
S1.7 Decoder pullback metric and common descent 33
S1.8 Numerical error budgets 36
S1.9 Sensitivity and identifiability 40
S1.10 Standing of the results 41
S2 Data tables 42
S3 Evaluation protocol 47
S4 Code 54
S5 Computational budget and derivations 93
S6 Supplementary figures 97
References 106

List of supplementary figures

Supplementary Figure S1. Worked examples fix the notation for counts, array shapes, derivatives, decoder sensitivity and step cost. 98
Supplementary Figure S2. Chance is a share of a crowd, a moving crowd carries its density, and noise on a path spreads the paths in the middle. 99
Supplementary Figure S3. Guided sampling is a run in five moves, and three of the moves are drawn as worked examples. 100
Supplementary Figure S4. Unbalanced optimal transport turns exposure and cost into a soft contact plan, and a debiased statistic compares two measures. 101
Supplementary Figure S5. Selection scores every candidate in several plausible worlds and then picks a fixed number of designs together. 102
Supplementary Figure S6. A fixed-horizon test is fully specified before outcomes are read, and sequential evidence stays a separate research interface. 103
Supplementary Figure S7. The contacts of one deposited nanobody-lysozyme complex are typed by residue pair, and a predictive layer is specified to learn from masked interface units. 104
Supplementary Figure S8. Selected public reports of antibody and nanobody design tools are dated and described by what each one reports, and the figure ranks none of them. 105

List of supplementary tables

Supplementary Table S1. Symbols of the mathematical supplement. 6
Supplementary Table S2. Exact bias of the normalising-constant estimate, of its logarithm and of the ratio estimate in Example 4.9. 21
Supplementary Table S3. Dataset roster. 45
Supplementary Table S4. Targets and cut-off dates of published systems, as the cited sources report them. 47
Supplementary Table S5. The 53 registered gates with their acceptance rules. 52
Supplementary Table S6. Smallest fall in a success rate, from a baseline of 0.80, that a one-sided exact binomial test at level 0.05 detects with 80% power on independent Bernoulli units. 53
Supplementary Table S7. Model, data and optimiser keys of the reference configuration `reference_2gpu.json` and their values. 84
Supplementary Table S8. Parameters per sequence block and per pair update. 93
Supplementary Table S9. Parameters per module of the reference size. 94
Supplementary Table S10. Floating-point operations per complex at padded lengths 300, 400 and 500 residues. 96
Supplementary Table S11. Memory of the optimiser state at the reference size, in bytes. 97

S1 Mathematical supplement

This supplement holds the proofs and derivations behind the Methods. The results concern mathematical objects: flows, densities, particle systems, transport plans, metrics and decision rules. None of them says that a designed molecule binds, that a model is accurate, or that one model is better than another.

A result marked cited is a theorem from the literature, used as stated. A proof marked sketch gives the idea and the key step. Results are numbered by section, so that Theorem 4.3 is the third numbered result of Section S1.4. Sections S1.2 to S1.9 cover geometry, scores and flow likelihood, transport and particle limits, decision statistics, the representation floor, the decoder metric, numerical error budgets and sensitivity. Section S1.10 states which results are proved in full and which are cited. Worked examples are numerical illustrations of the statement beside them, computed from the stated formulas.

S1.1 Notation and conventions

The symbols follow the notation of the Methods. **Supplementary Table S1** lists the symbols that the proofs use and the section where each is used. A few glyphs have a different local meaning in different sections, as the table notes.

Symbol	Meaning	Section
n_e, d_x, D	Transported residue count; coordinate dimension; dimension of the packed state, $D = d_x + 8n_e$	S1.2, S1.3
$q \in \mathbb{R}^D$	Dimensionless intrinsic state	S1.2 to S1.8
$u \in [0,1]$, f_u, ψ_u, p_u	Master clock; field; its flow; density of the state at clock u	S1.2, S1.3
$T = (R, \boldsymbol{t})$	Rigid motion with rotation R and shift $\boldsymbol{t}$ (this glyph is used only in S1.2)	S1.2
$(R_c, \boldsymbol{a}_c)$, $(A_g, \boldsymbol{o}_g)$	Context frame; chart matrix and offset of coordinate profile g	S1.2
$\Xi(u)$, ζ, $\gamma(u)$	Diffusion matrix of an SDE; Gaussian noise vector; width of a noise bridge	S1.3
ξ_{H}	Hutchinson probe vector	S1.3
Γ, C, μ_R, μ_E, ρ_{OT}, $\varepsilon_{\mathrm{OT}}$	Contact plan; cost; reference masses; marginal and entropic weights	S1.4
M, K_j, R_j, Ψ_j, ω_j, N_{p}	Horizon; reference kernel; proposal kernel; potential; incremental weight; particle count	S1.4
h_j	Backward information function (this glyph also names the decoder feature map h_ϕ in S1.7)	S1.4
$\widetilde{w}_j^{(k)}$, $\bar{w}_j^{(k)}$, $\bar{\ell}_j$, $\hat{Z}_M$	Carried weight; normalised weight; step factor; estimate of the normalising constant Z_M	S1.4

Symbol	Meaning	Section
$\tilde{f}_i$, g_i, a_i, w, n_f	Scaled loss; its gradient; preconditioned gradient; common direction; number of losses	S1.7
G_ϕ, h_ϕ, J_h, W_h, τ_{ridge}, Δ	Decoder metric; feature map; its Jacobian; feature weights; ridge; trust radius (written Δ_G in the Methods)	S1.7
η, π_j, ν_j	Entropic parameter; reference and alternative weights of plausible worlds	S1.5
$\mathcal{F}_t$, e_t, E_t	Filtration; evidence factor; accumulated evidence	S1.5
ϑ, $\mathcal{O}$	Parameter vector of the sensitivity analysis; map from parameters to observables	S1.9

Supplementary Table S1. Symbols of the mathematical supplement.

Norms are Euclidean unless a subscript names another. For a symmetric positive definite matrix G, $\| v \|_G^2 = v^\top G v$. The expression x_+ means $\max(x, 0)$. The generalised relative entropy of nonnegative vectors u and v of equal length is

$$\mathrm{KL}(u \mid v) = \sum_i \left[u_i \log \frac{u_i}{v_i} - u_i + v_i \right],$$

with the conventions $0\log 0 = 0$, $0\log(0/0) = 0$, and $+\infty$ for a positive u_i against $v_i = 0$. It is nonnegative, convex in both arguments, and zero exactly when $u = v$. The last two terms make it meaningful when the total masses differ. For probability vectors it is the ordinary relative entropy.

S1.2 Geometry and gauges

This section shows that a flow with a symmetric field produces symmetric samples (Theorem 2.1). The shift is removed by a linear change of coordinates whose choice alters no density or divergence (Theorem 2.3, Proposition 2.8). A frame attached to the antigen keeps the pose of the antibody relative to the antigen in the state (Proposition 2.4). Removing rotations as well creates points at which the geometry breaks down (Proposition 2.6).

The base-model code subtracts the mean of the present atoms of each complex from the coordinates in ångströms and then multiplies them by 0.1, so that the encoder input and the flow targets are in nanometres. It stores coordinates in the padded ambient layout and computes no density, score or divergence. The charts below define the space on which densities are evaluated, and they are needed wherever a density or a divergence is computed. The anchored chart belongs to the antigen-conditioned system.

Let $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in (\mathbb{R}^3)^n$ hold the Cα coordinates of $n = n_e \geq 3$ transported residues. A rigid motion $T = (R, \boldsymbol{t})$ with $R \in \mathrm{SO}(3)$ and $\boldsymbol{t} \in \mathbb{R}^3$ acts on every point by the same rule:

$$T \cdot \boldsymbol{x} = (R\boldsymbol{x}_1 + \boldsymbol{t}, \dots, R\boldsymbol{x}_n + \boldsymbol{t}).$$

Distances, bond angles and signed volumes do not change under this action. A reflection keeps distances but reverses the sign of a signed volume, and so turns L-amino acids into D-amino acids. The symmetry group is therefore $\mathrm{SE}(3) = \mathrm{SO}(3) \ltimes \mathbb{R}^3$ and does not include reflections. A velocity is a difference of positions, so a shift does not change it and a rotation turns it by R. Write $\rho_T \boldsymbol{v} = (R\boldsymbol{v}_1, \dots, R\boldsymbol{v}_n)$ for the induced action on velocities. This is an orthogonal map.

A field $f_u(\boldsymbol{x}, c)$ that depends on a condition c (the antigen coordinates, for instance) is called equivariant if

$$f_u(T \cdot \boldsymbol{x}, T \cdot c) = \rho_T f_u(\boldsymbol{x}, c)$$

for all states, clocks u and rigid motions. A scalar U is called invariant if $U(T \cdot \boldsymbol{x}, T \cdot c) = U(\boldsymbol{x}, c)$.

Theorem 2.1 (equivariance of the flow). Fix a condition c. Let $f_u(\boldsymbol{x}, c)$ be continuous in $u \in [0,1]$, uniformly Lipschitz in $\boldsymbol{x}$, and equivariant. Let $\psi_{u,c}(\boldsymbol{x}_0)$ be the solution of $\mathrm{d}/\mathrm{d}u\, \psi_{u,c}(\boldsymbol{x}_0) = f_u(\psi_{u,c}(\boldsymbol{x}_0), c)$ with $\psi_{0,c}(\boldsymbol{x}_0) = \boldsymbol{x}_0$. Then for every rigid motion T,

$$\psi_{u,T\cdot c}(T \cdot \boldsymbol{x}_0) = T \cdot \psi_{u,c}(\boldsymbol{x}_0).$$

If in addition the base law satisfies $p_0(\cdot \mid T \cdot c) = T_{\#} p_0(\cdot \mid c)$, then the endpoint laws satisfy $p_1(\cdot \mid T \cdot c) = T_{\#} p_1(\cdot \mid c)$.

Proof. Put $\boldsymbol{y}(u) = T \cdot \psi_{u,c}(\boldsymbol{x}_0)$. The action is affine, so $\dot{\boldsymbol{y}} = R\dot{\psi} = \rho_T f_u(\psi_{u,c}(\boldsymbol{x}_0), c)$. Equivariance with the state $\psi_{u,c}(\boldsymbol{x}_0)$ gives $\rho_T f_u(\psi_{u,c}(\boldsymbol{x}_0), c) = f_u(\boldsymbol{y}(u), T \cdot c)$. Also $\boldsymbol{y}(0) = T \cdot \boldsymbol{x}_0$. So $\boldsymbol{y}$ solves the equation with condition $T \cdot c$ and start $T \cdot \boldsymbol{x}_0$. The Lipschitz condition gives uniqueness (Picard–Lindelöf), hence $\boldsymbol{y}(u) = \psi_{u,T\cdot c}(T \cdot \boldsymbol{x}_0)$. For the laws, $p_1(\cdot \mid T \cdot c) = (\psi_{1,T\cdot c})_{\#} T_{\#} p_0(\cdot \mid c) = (\psi_{1,T\cdot c} \circ T)_{\#} p_0(\cdot \mid c) = (T \circ \psi_{1,c})_{\#} p_0(\cdot \mid c) = T_{\#} p_1(\cdot \mid c)$. ■

The theorem concerns a field that is equivariant by assumption.

Lemma 2.2 (centring projector). Let $P_n = I_n - \mathbf{1}\mathbf{1}^{\top}/n$. Then $P_n = P_n^{\top} = P_n^2$, $P_n \mathbf{1} = 0$ and $\mathrm{rank} P_n = n - 1$. The matrix $P = P_n \otimes I_3$ is the orthogonal projector onto $\mathcal{X}_n = \{\boldsymbol{x} : \sum_i \boldsymbol{x}_i = 0\}$.

Proof. Since $\mathbf{1}^{\top}\mathbf{1} = n$, $(\mathbf{1}\mathbf{1}^{\top}/n)^2 = \mathbf{1}\mathbf{1}^{\top}/n$, which gives symmetry and idempotence. The null space of P_n is spanned by $\mathbf{1}$ and its range is $\mathbf{1}^{\perp}$. Tensoring with I_3 applies the same projection to each of the three coordinate columns. ■

Theorem 2.3 (translation-free chart). Let the columns of $B \in \mathbb{R}^{n\times(n-1)}$ be an orthonormal basis of $\mathbf{1}^{\perp}$ and put $A = B^{\top} \otimes I_3$. Then A maps $\mathcal{X}_n$ isometrically onto $\mathbb{R}^{3n-3}$ with inverse $A^{\top}$, and every $\boldsymbol{x} \in (\mathbb{R}^3)^n$ splits uniquely as

$$\boldsymbol{x} = A^{\top}A\boldsymbol{x} + \mathbf{1} \otimes \bar{\boldsymbol{x}}, \qquad \bar{\boldsymbol{x}} = \frac{1}{n}\sum_i \boldsymbol{x}_i.$$

If $\tilde{B}$ is another orthonormal basis, then $\tilde{A} = QA$ for an orthogonal $Q \in \mathrm{O}(3n-3)$, and a density p_A in the A chart becomes $p_{\tilde{A}}(\tilde{q}) = p_A(Q^{\top}\tilde{q})$, with no log-determinant term.

Proof. The basis identities are $B^{\top}B = I_{n-1}$ and $BB^{\top} = P_n$. They give $AA^{\top} = I_{3n-3}$ and $A^{\top}A = P$. For centred $\boldsymbol{x}$, $P\boldsymbol{x} = \boldsymbol{x}$, so $\| A\boldsymbol{x} \|^2 = \boldsymbol{x}^{\top}P\boldsymbol{x} = \| \boldsymbol{x} \|^2$ and $A^{\top}A\boldsymbol{x} = \boldsymbol{x}$. For any q, $A^{\top}q$ is centred and $A(A^{\top}q) = q$, so A is a bijection. Further, $P\boldsymbol{x} = \boldsymbol{x} - \mathbf{1} \otimes \bar{\boldsymbol{x}}$ gives the splitting, and its two summands are orthogonal. Two bases span the same space, so $\tilde{B} = BQ_0$ with Q_0 orthogonal, and $Q = Q_0^{\top} \otimes I_3$ is orthogonal. A change of variables by an orthogonal map has Jacobian determinant of absolute value 1. ■

For $n_e = 130$ transported residues the latent block has $8n_e = 1{,}040$ numbers. The translation-free chart then has $3 \cdot 129 = 387$ coordinate numbers and $D = 1{,}427$. The context-anchored chart below keeps the relative shift, has 390 coordinate numbers, and $D = 1{,}430$. In general $D = 11n_e - 3$ for the free chart and $D = 11n_e$ for the anchored chart.

A Gaussian vector $\ell_0 A^{\top}\xi$ with $\xi \sim \mathcal{N}(0, I_{3n-3})$ has ambient covariance $\ell_0^2 P$. This matrix is singular in $\mathbb{R}^{3n}$, so the density of such a vector exists only on the centred subspace. Any density in this supplement is therefore evaluated in the declared chart and not in the ambient space.

Proposition 2.4 (anchored coordinates). Let the condition fix an origin $\boldsymbol{a}_c \in \mathbb{R}^3$ and a rotation $R_c \in \mathrm{SO}(3)$, and define $\boldsymbol{y}_i = R_c^{\top}(\boldsymbol{x}_i - \boldsymbol{a}_c)$. A rigid motion $T = (R, \boldsymbol{t})$ acts on the frame by $\boldsymbol{a}_{T\cdot c} = R\boldsymbol{a}_c + \boldsymbol{t}$ and $R_{T\cdot c} = RR_c$. Then $\boldsymbol{y}_i(T \cdot \boldsymbol{x}, T \cdot c) = \boldsymbol{y}_i(\boldsymbol{x}, c)$, and the map $\boldsymbol{x} \mapsto \boldsymbol{y}$ keeps all $3n$ numbers of the antibody.

Proof. Substitute: $(RR_c)^{\top}((R\boldsymbol{x}_i + \boldsymbol{t}) - (R\boldsymbol{a}_c + \boldsymbol{t})) = R_c^{\top}R^{\top}R(\boldsymbol{x}_i - \boldsymbol{a}_c) = \boldsymbol{y}_i(\boldsymbol{x}, c)$. The map is affine and invertible, with inverse $\boldsymbol{x}_i = \boldsymbol{a}_c + R_c\boldsymbol{y}_i$. ■

Proposition 2.5 (error in an estimated frame). For rotations $R, \hat{R}$ and origins $\boldsymbol{a}, \hat{\boldsymbol{a}}$,

$$\| \hat{R}^{\top}(\boldsymbol{x} - \hat{\boldsymbol{a}}) - R^{\top}(\boldsymbol{x} - \boldsymbol{a}) \| \le \| \hat{R} - R \|_2 \, \| \boldsymbol{x} - \boldsymbol{a} \| + \| \hat{\boldsymbol{a}} - \boldsymbol{a} \|.$$

To first order, with $\hat{R} = R\exp([\delta\boldsymbol{\omega}]_{\times})$ and $\hat{\boldsymbol{a}} = \boldsymbol{a} + \delta\boldsymbol{a}$, the anchored coordinates change by $\delta\boldsymbol{y} = -[\delta\boldsymbol{\omega}]_{\times}\boldsymbol{y} - R^{\top}\delta\boldsymbol{a}$.

Proof. Write the difference as $\hat{R}^{\top}((\boldsymbol{x}-\hat{\boldsymbol{a}})-(\boldsymbol{x}-\boldsymbol{a}))+(\hat{R}-R)^{\top}(\boldsymbol{x}-\boldsymbol{a})$ and use $\| \hat{R} \|_2 = 1$. For the expansion, $\hat{R}^{\top}(\boldsymbol{x}-\hat{\boldsymbol{a}}) = \exp(-[\delta\boldsymbol{\omega}]_{\times})R^{\top}(\boldsymbol{x}-\boldsymbol{a}-\delta\boldsymbol{a})$, and $\exp(-[\delta\boldsymbol{\omega}]_{\times}) = I - [\delta\boldsymbol{\omega}]_{\times} + O(\| \delta\boldsymbol{\omega} \|^2)$. ■

A rotation error of the frame therefore grows with the distance from the origin, and a shift error moves every point equally.

The flow of the Methods works in the flat charts above, and shape space is used only to compare conformations.

Proposition 2.6 (orbit tangent, inertia tensor and horizontal projector). For $\boldsymbol{\omega} \in \mathbb{R}^3$ let $(K_{\boldsymbol{x}}\boldsymbol{\omega})_i = \boldsymbol{\omega} \times \boldsymbol{x}_i$. This map spans the vertical space, the tangent space of the rotation orbit through $\boldsymbol{x}$. Let $I_{\boldsymbol{x}} = \sum_i (\| \boldsymbol{x}_i \|^2 I_3 - \boldsymbol{x}_i \boldsymbol{x}_i^{\top})$. Then:

1. The adjoint of $K_{\boldsymbol{x}}$ is $K_{\boldsymbol{x}}^{*}\boldsymbol{v} = \sum_i \boldsymbol{x}_i \times \boldsymbol{v}_i$ (the angular momentum) and $K_{\boldsymbol{x}}^{*}K_{\boldsymbol{x}} = I_{\boldsymbol{x}}$.
2. $\boldsymbol{\omega}^{\top} I_{\boldsymbol{x}} \boldsymbol{\omega} = \sum_i \| \boldsymbol{\omega} \times \boldsymbol{x}_i \|^2$. Hence $I_{\boldsymbol{x}} \succcurlyeq 0$, and $I_{\boldsymbol{x}}$ is singular exactly when some nonzero axis is parallel to every nonzero $\boldsymbol{x}_i$. For a centred configuration this means that all points lie on one line through the origin.
3. If $I_{\boldsymbol{x}} \succ 0$, the map $P_{\boldsymbol{x}}^{\mathrm{H}} = I - K_{\boldsymbol{x}} I_{\boldsymbol{x}}^{-1} K_{\boldsymbol{x}}^{*}$ is the orthogonal projector onto the horizontal space, the set of velocities with zero angular momentum. It satisfies $P_{R\boldsymbol{x}}^{\mathrm{H}}(R\boldsymbol{v}) = R\, P_{\boldsymbol{x}}^{\mathrm{H}}\boldsymbol{v}$.
4. If the right-hand side $\boldsymbol{b}$ of $I_{\boldsymbol{x}}\boldsymbol{\omega} = \boldsymbol{b}$ is perturbed by $\delta\boldsymbol{b}$, the solution changes by at most $\| \delta\boldsymbol{b} \| / \lambda_{\min}(I_{\boldsymbol{x}})$.

Proof. (1) The scalar triple product gives $\langle K_{\boldsymbol{x}}\boldsymbol{\omega}, \boldsymbol{v} \rangle = \sum_i (\boldsymbol{\omega} \times \boldsymbol{x}_i)^{\top}\boldsymbol{v}_i = \boldsymbol{\omega}^{\top} \sum_i \boldsymbol{x}_i \times \boldsymbol{v}_i$. Also $\boldsymbol{x} \times (\boldsymbol{\omega} \times \boldsymbol{x}) = (\| \boldsymbol{x} \|^2 I - \boldsymbol{x}\boldsymbol{x}^{\top})\boldsymbol{\omega}$, and summing over i gives $K_{\boldsymbol{x}}^{*}K_{\boldsymbol{x}} = I_{\boldsymbol{x}}$. (2) The quadratic form is $\| K_{\boldsymbol{x}}\boldsymbol{\omega} \|^2$, which vanishes exactly when every $\boldsymbol{\omega} \times \boldsymbol{x}_i = 0$. (3) When $I_{\boldsymbol{x}} = K_{\boldsymbol{x}}^{*}K_{\boldsymbol{x}}$ is invertible, $K_{\boldsymbol{x}} I_{\boldsymbol{x}}^{-1} K_{\boldsymbol{x}}^{*}$ is the orthogonal projector onto $\mathrm{range} K_{\boldsymbol{x}}$. Therefore $K_{\boldsymbol{x}}^{*} P_{\boldsymbol{x}}^{\mathrm{H}} \boldsymbol{v} = 0$, the removed part lies in the vertical space, and the two parts are orthogonal. Cross products obey $(R\boldsymbol{x}) \times (R\boldsymbol{v}) = R(\boldsymbol{x} \times \boldsymbol{v})$ and $I_{R\boldsymbol{x}} = R I_{\boldsymbol{x}} R^{\top}$, which gives equivariance. (4) The two solutions differ by $I_{\boldsymbol{x}}^{-1}\delta\boldsymbol{b}$, and $\| I_{\boldsymbol{x}}^{-1} \|_2 = 1/\lambda_{\min}(I_{\boldsymbol{x}})$. ■

By item 2, the orbit through a collinear configuration has dimension two and not three, which is why the quotient by rotations is not a smooth manifold there, and by item 4 the horizontal solve becomes unstable as $\lambda_{\min}(I_{\boldsymbol{x}}) \to 0$.

Rotations themselves are handled by the matrix exponential. For $\Omega^{\top} = -\Omega$ and $R \in \mathrm{SO}(3)$ the update $R^{+} = R\exp(h\Omega)$ stays in $\mathrm{SO}(3)$ for every real h, because $\exp(h\Omega)^{\top} = \exp(h\Omega)^{-1}$ and $\det\exp(h\Omega) = \exp(h\mathrm{tr}\Omega) = 1$. By Rodrigues' formula, $\exp([\boldsymbol{\omega}]_{\times}) = I + \frac{\sin\theta}{\theta}[\boldsymbol{\omega}]_{\times} + \frac{1-\cos\theta}{\theta^2}[\boldsymbol{\omega}]_{\times}^2$ with $\theta = \| \boldsymbol{\omega} \|$. At $\theta = \pi$ the sine term vanishes, so a half-turn about $\boldsymbol{n}$ and about

$-\boldsymbol{n}$ coincide. The logarithm of a half-turn is therefore two-valued, and a velocity target built from it is discontinuous. The flow of the Methods transports $\mathrm{C}\alpha$ coordinates in Euclidean charts and decodes local frames afterwards, so it never uses a logarithm of a rotation.

Lemma 2.7 (gradients of invariant objects). Let ρ be an orthogonal representation acting on a chart and let $U(\rho_T q, T \cdot c) = U(q, c)$. Then $\nabla_q U(\rho_T q, T \cdot c) = \rho_T \nabla_q U(q, c)$. In particular, if a density $p(\cdot \mid c)$ is invariant, with $p(\rho_T q \mid T \cdot c) = p(q \mid c)$ and a volume that rigid motions preserve, its score $\nabla_q \log p$ is equivariant. If $G(\rho_T q, T \cdot c) = \rho_T G(q, c) \rho_T^{\top}$, then $G^{-1} \nabla U$ is equivariant.

Proof. For every perturbation $\boldsymbol{h}$, invariance gives $\nabla U(\rho_T q, T \cdot c)^{\top} \rho_T \boldsymbol{h} = \nabla U(q, c)^{\top} \boldsymbol{h}$. Hence $\rho_T^{\top} \nabla U(\rho_T q, T \cdot c) = \nabla U(q, c)$, and multiplication by ρ_T gives the claim. The statement for $\log p$ follows because $|\det R| = 1$ keeps the volume, and the statement for $G^{-1} \nabla U$ because $(\rho_T G \rho_T^{\top})^{-1} = \rho_T G^{-1} \rho_T^{\top}$. ■

Proposition 2.8 (divergence under a change of chart). Let $Q \in \mathrm{O}(d)$, $\boldsymbol{y} = Q\boldsymbol{x}$ and $\tilde{f}(\boldsymbol{y}) = Q f(Q^{\top} \boldsymbol{y})$. Then $\mathrm{div}_{\boldsymbol{y}} \tilde{f}(\boldsymbol{y}) = \mathrm{div}_{\boldsymbol{x}} f(\boldsymbol{x})$. Consequently the rate $\mathrm{d}/\mathrm{d}u \log p_u$ along a flow and the integrated log-likelihood of Theorem 3.7 do not depend on the orthonormal basis B chosen in Theorem 2.3.

Proof. The Jacobian of $\tilde{f}$ is $Q D f\, Q^{\top}$, and the trace is invariant under cyclic permutation: $\mathrm{tr}(Q D f Q^{\top}) = \mathrm{tr}(D f)$. ■

S1.3 Scores, diffusion reversal and flow likelihood

This section shows that the regression target of the flow loss is the conditional mean of the velocity (Theorem 3.2), also with independent block times (Corollary 3.3). It derives the score (Proposition 3.4) and links stochastic and deterministic samplers (Theorems 3.5 and 3.6). It gives the density of a generated sample with the errors of its numerical evaluation (Theorem 3.7, Propositions 3.8 and 3.9). **Supplementary Figure S2** gives worked examples of chance as an area, a moving crowd with its density, and the noise bridge.

Theorem 3.2 and Corollary 3.3 are the basis of the flow loss of the base model: the code draws independent uniform times for the coordinate block and the latent block and regresses the velocity heads on the difference of data and noise. The results after Corollary 3.3 support guidance, sampling with a score and likelihood evaluation, which the base-model code does not contain.

Lemma 3.1 (conditional-mean projection). Let Y be square-integrable, let X be any random element, and let $h(X)$ be square-integrable. Then

$$\mathbb{E} \parallel Y - h(X) \parallel^2 = \mathbb{E} \parallel Y - \mathbb{E}(Y \mid X) \parallel^2 + \mathbb{E} \parallel \mathbb{E}(Y \mid X) - h(X) \parallel^2.$$

Hence the conditional mean is the almost surely unique minimiser.

Proof. Put $m = \mathbb{E}(Y \mid X)$ and write $Y - h = (Y - m) + (m - h)$. The cross term has expectation $\mathbb{E}[\mathbb{E}(Y - m \mid X)^\top (m - h)] = 0$. ■

Theorem 3.2 (flow-matching target). Let $q_1 \sim p_{\text{data}}(\cdot \mid c)$ and let q_u be drawn from a conditional path $p_u(\cdot \mid q_1, c)$ with conditional velocity $b_u(q \mid q_1, c)$. Put $B = b_u(q_u \mid q_1, c)$ and assume that (q_u, B) is square-integrable. Then:

1. The minimiser over measurable fields v of $\mathbb{E} \parallel v(q_u, u, c) - B \parallel^2$ is $b_u(q, c) = \mathbb{E}[B \mid q_u = q]$. The conditional loss $\mathcal{L}_{\text{CFM}}(\theta)$ equals the marginal loss $\mathcal{L}_{\text{FM}}(\theta) = \mathbb{E} \parallel v_\theta - b_u(q_u, c) \parallel^2$ plus a constant that does not depend on θ. Hence the two losses have the same gradient.
2. If every conditional path satisfies the continuity equation with velocity $b_u(\cdot \mid q_1, c)$ in weak form, then the marginal path $p_u(q \mid c) = \int p_u(q \mid q_1, c)\, p_{\text{data}}(q_1 \mid c)\, \mathrm{d}q_1$ satisfies the continuity equation with the field $b_u(q, c)$.

Proof. (1) Apply Lemma 3.1 with $Y = B$ and $X = q_u$ (at fixed u and c). The second term, $\mathbb{E} \parallel m - B \parallel^2$, does not depend on v. For $v = v_\theta$ it is the constant in the claim, and the first term is $\mathcal{L}_{\text{FM}}$. (2) For a smooth compactly supported test function φ, the weak form for the conditional paths gives $\mathrm{d}/\mathrm{d}u\, \mathbb{E}\varphi(q_u) = \mathbb{E}[\nabla\varphi(q_u)^\top B] = \mathbb{E}[\nabla\varphi(q_u)^\top m(q_u)]$ by the tower property. This is the weak continuity equation for the marginal path with velocity m. Differentiation under the expectation needs the stated integrability and a dominating function for the derivative. ■

Corollary 3.3 (independent block times and the master clock). Let the state have blocks $b \in \{x, z\}$ with paired linear paths $q_b = (1 - t_b)q_{b,0} + t_b q_{b,1}$, where $q_0 \sim p_0$ is independent of q_1 and the times $t_x, t_z \in [0,1]$ are independent of the pair. Let $v_b(q, t_x, t_z, c) = \mathbb{E}[q_{b,1} - q_{b,0} \mid q_{t_x,t_z} = q]$. Then:

1. For each block, the minimiser of $\mathbb{E} \parallel v - (q_{b,1} - q_{b,0}) \parallel^2$ over fields that see (q, t_x, t_z, c) is v_b.
2. For any absolutely continuous nondecreasing schedule $u \mapsto (t_x(u), t_z(u))$ with $t_x(0) = t_z(0) = 0$ and $t_x(1) = t_z(1) = 1$, the laws $p_{t_x(u),t_z(u)}$ satisfy the continuity equation with the master-clock field $f_u = (t_x'(u)\, v_x,\ t_z'(u)\, v_z)$ evaluated at $(q, t_x(u), t_z(u), c)$.

Proof. (1) is Theorem 3.2 with the conditioning variable enlarged to (q, t_x, t_z, c). (2) The state q_u is a function of $(q_0, q_1, t_x(u), t_z(u))$ with $\partial q_u / \partial t_b = q_{b,1} - q_{b,0}$. By the chain rule,

$\mathrm{d}q_u/\mathrm{d}u = \big(t_x'(q_{x,1} - q_{x,0}),\, t_z'(q_{z,1} - q_{z,0})\big)$. For a test function φ, the tower property gives $\mathrm{d}/\mathrm{d}u\, \mathbb{E}\varphi(q_u) = \mathbb{E}[\nabla\varphi(q_u)^\top \mathrm{d}q_u/\mathrm{d}u] = \mathbb{E}[\nabla\varphi(q_u)^\top f_u(q_u)]$. ■

Proposition 3.4 (score from noise prediction). Let $Q = \alpha Y + LZ$, where $Z \sim \mathcal{N}(0, I_D)$ is independent of Y and the matrix L is invertible. Then:

1. $\nabla_q \log p_Q(q) = -L^{-\top}\, \mathbb{E}[Z \mid Q = q]$.
2. For a predictor $\hat{Z}(q)$ of the noise, the induced score $\hat{s} = -L^{-\top}\hat{Z}$ satisfies $\| \hat{s} - \nabla\log p_Q \| \leq \| L^{-\top} \|_2 \, \| \hat{Z} - \mathbb{E}(Z \mid Q) \|$.
3. The denoising score loss $\mathbb{E} \| s_\theta(Q) + L^{-\top}Z \|^2$ equals $\mathbb{E} \| s_\theta(Q) - \nabla\log p_Q(Q) \|^2$ plus a constant.
4. If α is invertible, then $\mathbb{E}[Y \mid Q = q] = \alpha^{-1}\big(q + LL^\top \nabla\log p_Q(q)\big)$.

For an isotropic block such as the latent coordinates, $\alpha = t_z I$ and $L_z = (1 - t_z)I$, so $\nabla_{q_z}\log p = \frac{t_z v_z - q_z}{1 - t_z}$ with $v_z = \mathbb{E}[q_{z,1} - q_{z,0} \mid q]$. Under the shared translation noise of the coordinate block, $q_{x,0} \sim \mathcal{N}(0, \Sigma_0)$ with $\Sigma_0 = (I_n + c_d^2 \mathbf{1}_n \mathbf{1}_n^\top) \otimes I_3$ and $L_x L_x^\top = (1 - t_x)^2 \Sigma_0$, so $\nabla_{q_x}\log p = (1 - t_x)^{-1}\Sigma_0^{-1}(t_x v_x - q_x)$ with $\Sigma_0^{-1} = (I_n - c_d^2/1 + n c_d^2\, \mathbf{1}_n \mathbf{1}_n^\top) \otimes I_3$, while $\lambda_{\min}(\Sigma_0) = 1$ still gives $\| L_x^{-\top} \|_2 = 1/(1 - t_x)$. For the interpolant $q = \mathcal{I} + \gamma(u)\zeta$ one gets $\nabla\log p_u = -\mathbb{E}[\zeta \mid q]/\gamma(u)$.

Proof. (1) Given y, the kernel is $k(q \mid y) = \mathcal{N}(q; \alpha y, LL^\top)$, and $\nabla_q \log k(q \mid y) = -(LL^\top)^{-1}(q - \alpha y) = -L^{-\top}z$ with $z = L^{-1}(q - \alpha y)$. Since $p_Q = \int k(\cdot \mid y)\, p_Y(\mathrm{d}y)$, differentiation under the integral (justified by dominated convergence) gives $\nabla\log p_Q(q) = \mathbb{E}[\nabla_q \log k(q \mid Y) \mid Q = q]$, which is the claim. (2) Subtract the two expressions for the score. (3) The target $T = -L^{-\top}Z$ has $\mathbb{E}[T \mid Q] = \nabla\log p_Q(Q)$ by (1), so Lemma 3.1 splits the loss into the score error and a term free of s_θ. (4) From (1), $\mathbb{E}[LZ \mid q] = -LL^\top \nabla\log p_Q(q)$, and $\mathbb{E}[LZ \mid q] = q - \alpha\mathbb{E}[Y \mid q]$. For the paired path, $\mathbb{E}[Z_b \mid q] = q_b - t_b v_b$ because $q_b = Z_b + t_b(Y_b - Z_b)$. ■

The norm $\| L^{-\top} \|_2 = 1/(1 - t_b)$ tends to infinity as $t_b \to 1$. A bounded error in the noise prediction does not bound the score error at the data end. Training can regress the noise without a singular target. A sampler then evaluates the score only for $t_b \leq 1 - \delta$ and records separately whether it integrates an analytic limit at the end or targets an endpoint smoothed at scale δ. Optimal-transport couplings between noise and data change the Gaussian kernel used in the proof, so item 1 applies to independent coupling only.

Theorem 3.5 (shared marginals of an SDE and an ODE). Let $\mathrm{d}Q_u = b(Q_u, u)\,\mathrm{d}u + \Lambda(u)\,\mathrm{d}W_u$ with W a D-dimensional Brownian motion and $\Xi(u) = \Lambda\Lambda^\top$ positive definite and

independent of the state. Let the density $p_u > 0$ be twice continuously differentiable, let $s_u = \nabla \log p_u$, and assume that the Fokker–Planck equation has a unique solution in the class containing p_u. Then:

1. With $v = b - 1/2\, \Xi s_u$, the density obeys $\partial_u p_u = -\nabla \cdot (v p_u)$. If v is locally Lipschitz in the state with linear growth, the continuity equation has a unique solution, and the ODE $\dot{q} = v(q, u)$ started from p_0 has the marginals p_u.
2. Conversely, let a field v transport the density path p_u. For every diffusion matrix $\Xi(u) \succcurlyeq 0$ independent of the state, with a dispersion $\Lambda(u)$ such that $\Lambda\Lambda^\top = \Xi$, the SDE with drift $v + 1/2\, \Xi s_u$ has the marginals p_u. For $\Xi = 2\epsilon(u) I$ this is the SDE with drift $v + \epsilon \nabla \log p_u$ and noise $\sqrt{2\epsilon}\, \mathrm{d}W$.

Proof. The Fokker–Planck equation is $\partial_u p = -\nabla \cdot (bp) + 1/2 \sum_{ij} \Xi_{ij}\, \partial_{ij} p$. Since Ξ does not depend on the state, $1/2 \sum_{ij} \Xi_{ij}\, \partial_{ij} p = \nabla \cdot (1/2\, \Xi \nabla p) = \nabla \cdot (1/2\, \Xi s\, p)$, using $\nabla p = sp$. So $\partial_u p = -\nabla \cdot ((b - 1/2\, \Xi s) p)$, which is (1). For (2), write the Fokker–Planck equation of the SDE for a candidate p': $\partial_u p' = -\nabla \cdot ((v + 1/2\, \Xi s_u) p') + \nabla \cdot (1/2\, \Xi \nabla p')$. For $p' = p_u$ the two terms with Ξ cancel because $s_u p_u = \nabla p_u$, and the equation reduces to $\partial_u p_u = -\nabla \cdot (v p_u)$, which holds. Uniqueness gives $p' = p_u$. ■

The marginals agree and the path laws do not. The SDE has positive quadratic variation and explores across barriers. The ODE has bounded variation and defines a deterministic map that can be inverted and replayed.

Theorem 3.6 (time reversal; the Markov property is cited). Under the conditions of Theorem 3.5, assume in addition that b is continuous in u and locally Lipschitz with linear growth, that Ξ is continuously differentiable and uniformly positive definite, and that $p_u |b|$ and $\Xi \nabla p_u$ vanish at infinity. Then $Y_r = Q_{1-r}$ is a Markov diffusion with dispersion $\Lambda(1 - r)$ and drift

$$\bar{b}(y, r) = -b(y, 1 - r) + \Xi(1 - r)\, \nabla \log p_{1-r}(y).$$

Equivalently, run backward in u, $\mathrm{d}Q_u = [b(Q_u, u) - \Xi(u) \nabla \log p_u(Q_u)]\mathrm{d}u + \Lambda(u)\, \mathrm{d}\widetilde{W}_u$.

Derivation of the drift. Write the Fokker–Planck equation as a conservation law, $\partial_u p = -\nabla \cdot j$, with probability current $j = pb - 1/2\, \Xi \nabla p = p(b - 1/2\, \Xi s)$. For $\bar{p}_r = p_{1-r}$ one has $\partial_r \bar{p}_r = \nabla \cdot j_{1-r}$. A diffusion with dispersion $\Lambda(1 - r)$ and drift $\bar{b}$ has current $\bar{\jmath} = \bar{p}\bar{b} - 1/2\, \Xi(1 - r) \nabla \bar{p}$. The reversed process carries the opposite current, $\bar{\jmath}_r = -j_{1-r}$. Dividing by the positive density gives $\bar{b} - 1/2\, \Xi s = -b + 1/2\, \Xi s$, hence $\bar{b} = -b + \Xi s$. That the time-reversed process is a Markov diffusion with these coefficients is Anderson's theorem [1], which is used as stated (cited). ■

Theorem 3.7 (density of a continuous flow). Let $f_u(q)$ be continuous in u and continuously differentiable in q, and let its flow ψ_u be a diffeomorphism on the region of interest. Then

$$\det D\psi_u(q_0) = \exp\left\{\int_0^u \nabla \cdot f_s(\psi_s(q_0))\,\mathrm{d}s\right\} > 0.$$

If f pushes p_0 to p_u, then $\log p_1(\psi_1(q_0)) = \log p_0(q_0) - \int_0^1 \nabla \cdot f_u(\psi_u(q_0))\,\mathrm{d}u$.

Proof. The Jacobian $J_u = D\psi_u(q_0)$ solves $\dot{J} = Df\,J$ with $J_0 = I$. Jacobi's formula gives $\mathrm{d}/\mathrm{d}u \log|\det J| = \mathrm{tr}(J^{-1}\dot{J}) = \mathrm{tr}Df = \nabla \cdot f$. The determinant cannot cross zero while ψ_u is a diffeomorphism, so it stays positive. Integration gives the first claim, and the change of variables $p_1(\psi_1(q_0))\det J_1 = p_0(q_0)$ gives the second. ■

Three remarks complete the theorem. First, the packed field is $f_u = (t_x' v^x,\ t_z' v^z)$, and its divergence is $t_x' \mathrm{tr}(\partial v^x / \partial y_x) + t_z' \mathrm{tr}(\partial v^z / \partial z_e)$, because the cross blocks lie off the diagonal and add nothing to the trace. Second, the value is a density in the dimensionless chart. Rescaling the coordinates by ℓ_0 and each latent channel by $\sigma_{z,r}$ multiplies the density by a constant Jacobian, so that

$$\log p^{\mathrm{phys}}(y_x, z_e) = \log p(q) - d_x \log \ell_0 - n_e \sum_{r=1}^{8} \log \sigma_{z,r}.$$

Third, a chart that changes with the clock, $y = \Phi_u(q)$, has the velocity $D\Phi_u\, f + \partial_u \Phi_u$ and a time-dependent Jacobian. The Methods fix the chart during one solve, so that no term $\partial_u \Phi_u$ is omitted.

Proposition 3.8 (variance of trace probes). Let $J \in \mathbb{R}^{D\times D}$, $S = 1/2\,(J + J^{\top})$, and let ξ_{H} have mean zero and identity covariance. Then $\mathbb{E}[\xi_{\mathrm{H}}^{\top} J \xi_{\mathrm{H}}] = \mathrm{tr}J$. For a standard Gaussian probe, $\mathrm{Var}(\xi_{\mathrm{H}}^{\top} J \xi_{\mathrm{H}}) = 2 \parallel S \parallel_{\mathrm{F}}^2$. For a Rademacher probe, $\mathrm{Var}(\xi_{\mathrm{H}}^{\top} J \xi_{\mathrm{H}}) = 2\sum_{i\neq j} S_{ij}^2$. The mean of K independent probes has $1/K$ of these variances.

Proof. The expectation is $\mathrm{tr}(J\,\mathbb{E}[\xi_{\mathrm{H}}\xi_{\mathrm{H}}^{\top}]) = \mathrm{tr}J$. Since $\xi^{\top}(J - J^{\top})\xi = 0$, the quadratic form equals $\xi^{\top} S \xi$. For a Gaussian probe, Isserlis' identity gives $\mathbb{E}(\xi^{\top} S \xi)^2 = (\mathrm{tr}S)^2 + 2\mathrm{tr}(S^2)$, and subtracting $(\mathrm{tr}S)^2$ leaves $2 \parallel S \parallel_{\mathrm{F}}^2$. For a Rademacher probe, $\xi_i^2 = 1$, so the diagonal terms are constant. The off-diagonal terms $\sum_{i\neq j} S_{ij}\,\xi_i \xi_j$ are pairwise uncorrelated with variance $2\sum_{i\neq j} S_{ij}^2$. Independence divides the variance by K. ■

For a diagonal J one Rademacher probe returns $\mathrm{tr}J$ exactly, while a Gaussian probe has variance $2\sum_i \lambda_i^2$.

Proposition 3.9 (error of a computed log density). Let f be L_f-Lipschitz in the state and let $d = \nabla \cdot f$ be L_d-Lipschitz, both uniformly in u. Let a numerical trajectory $\hat{q}(u)$ with $\hat{q}(0) = q_0$ satisfy $\| \dot{\hat{q}} - f(\hat{q}, u) \| \leq \varepsilon_f$, and let a divergence estimate satisfy $|\hat{d} - d| \leq \varepsilon_d$. Then

$$\left| \int_0^1 \hat{d}\,(\hat{q}(u), u)\,\mathrm{d}u - \int_0^1 d\,(q(u), u)\,\mathrm{d}u \right| \leq \varepsilon_d + L_d\,\varepsilon_f\,\Phi(L_f),$$

where $\Phi(L_f) = (e^{L_f} - 1 - L_f)/L_f^2$ for $L_f > 0$ and $\Phi(0) = 1/2$.

Proof. Let $e = \hat{q} - q$ and $r = \dot{\hat{q}} - f(\hat{q}, u)$. Then $\| r \| \leq \varepsilon_f$ and $D^+ \| e \| \leq L_f \| e \| + \varepsilon_f$, so Grönwall's inequality gives $\| e(u) \| \leq \varepsilon_f (e^{L_f u} - 1)/L_f$. For each u, $|\hat{d}(\hat{q}) - d(q)| \leq |\hat{d}(\hat{q}) - d(\hat{q})| + |d(\hat{q}) - d(q)| \leq \varepsilon_d + L_d \| e \|$. Integrate over $[0,1]$, using $\int_0^1 (e^{L_f u} - 1)\,\mathrm{d}u = (e^{L_f} - 1 - L_f)/L_f$. The case $L_f = 0$ integrates u to $1/2$. ■

A guided field $\tilde{f} = f_\theta + \kappa\, G_\phi^{-1} \nabla U$ defines its own flow, and Theorem 3.7 applies to it with the divergence of the complete assembled field, including the guidance term. The endpoint law is the pushforward of p_0 under that flow. It does not equal the base law tilted by e^U unless the guidance term is derived from an exact Doob transform of a stated path measure. No statement in this supplement says that it does. Raw log density values of guided and unguided runs therefore need the same reference measure before they are compared.

S1.4 Transport and particle limits

This section treats two tools. The first is a static soft assignment of contact mass between antibody and epitope residues, a convex problem with a unique solution (Theorems 4.2 and 4.3, Propositions 4.4 to 4.6). The second is a particle method that steers a stochastic sampler while it keeps an account of the weights (Theorems 4.7, 4.10 and 4.14 and the results between them). A contact plan is not a generative law, and a high effective sample size does not show that the contact optimisation has converged. **Supplementary Figure S4** and **Supplementary Figure S3** illustrate the contact plan and the particle weights.

Contact plans from unbalanced transport. Let $\mu_R \in \mathbb{R}_{>0}^{n_R}$ and $\mu_E \in \mathbb{R}_{>0}^{n_E}$ be the reference masses of the antibody residues and the epitope residues, and let C be a finite cost matrix. Write $\rho = \rho_{\mathrm{OT}}$ and $\varepsilon = \varepsilon_{\mathrm{OT}}$, both positive. For a plan $\Gamma \geq 0$ with marginals $\Gamma\mathbf{1}$ and $\Gamma^\top\mathbf{1}$ the objective is

$$\mathcal{F}(\Gamma) = \langle C, \Gamma \rangle + \rho \mathrm{KL}(\Gamma\mathbf{1} \mid \mu_R) + \rho \mathrm{KL}(\Gamma^\top\mathbf{1} \mid \mu_E) + \varepsilon \mathrm{KL}(\Gamma \mid \mu_R \otimes \mu_E),$$

and $\mathrm{OT}_{\varepsilon,\rho} = \min_{\Gamma \geq 0} \mathcal{F}(\Gamma)$. The marginal terms let a residue that has no good partner send out less mass than its reference, at a price set by ρ. The last term spreads mass over near-equal poses and makes the problem strictly convex.

Debiased divergence. For two measures α and β on one space, with one shared cost, the debiased divergence is

$$S_{\varepsilon,\rho}(\alpha, \beta) = \mathrm{OT}_{\varepsilon,\rho}(\alpha, \beta) - 1/2\ \mathrm{OT}_{\varepsilon,\rho}(\alpha, \alpha) - 1/2\ \mathrm{OT}_{\varepsilon,\rho}(\beta, \beta) + \varepsilon/2\ (m(\alpha) - m(\beta))^2,$$

where m is the total mass [2]. The last term vanishes when the two masses are equal.

Lemma 4.1 (two elementary facts). (a) For $u \geq 0$ with sum s and $v > 0$ with sum m, $\mathrm{KL}(u \mid v) \geq s\log(s/m) - s + m$. (b) For $y > 0$, $\lambda > 0$ and $\phi_y(x) = x\log(x/y) - x + y$ on $x \geq 0$, $\inf_{x \geq 0}\{\lambda \phi_y(x) - \sigma x\} = -\lambda y\ (e^{\sigma/\lambda} - 1)$, with the unique minimiser $x^\star = y e^{\sigma/\lambda}$.

Proof. (a) Jensen's inequality for the convex function $x\log x$ with weights v_i/m gives $\sum_i v_i\ \phi(u_i/v_i) \geq m\, \phi(s/m)$ with $\phi(x) = x\log x$. Rearranging gives the claim. (b) The function is strictly convex with derivative $\lambda\log(x/y) - \sigma$, which tends to $-\infty$ at $x = 0$. So the minimiser is interior and solves $\lambda\log(x/y) = \sigma$. Substitution gives $\lambda(y - x^\star) = -\lambda y(e^{\sigma/\lambda} - 1)$. ■

Theorem 4.2 (existence, uniqueness and positivity). The problem $\min_{\Gamma \geq 0} \mathcal{F}(\Gamma)$ has a unique minimiser $\Gamma^\star$, and every entry of $\Gamma^\star$ is strictly positive.

Proof. The objective is continuous on the closed orthant under the convention $0\log 0 = 0$. It is strictly convex, because the entropic term is strictly convex when every reference entry $\mu_{R,i}\mu_{E,j}$ is positive, and the other terms are convex. For coercivity put $s = \sum_{ij} \Gamma_{ij}$ and $m_R = \sum_i \mu_{R,i}$. By Lemma 4.1(a) applied to the row marginal, $\mathcal{F}(\Gamma) \geq (\min_{ij} C_{ij})\, s + \rho\, [s\log(s/m_R) - s + m_R]$, which tends to $+\infty$ with s because the term $\rho s\log s$ beats any linear growth. Hence a minimiser exists, and strict convexity makes it unique.

For positivity, suppose $\Gamma_{ij} = 0$ and raise that entry by $t > 0$. Let r_i and c_j be its row and column masses. The change of $\mathcal{F}$ is $[\varepsilon + \rho\, \mathbf{1}\{r_i = 0\} + \rho\, \mathbf{1}\{c_j = 0\}]\, t\log t + O(t)$ as $t \downarrow 0$. The bracket is positive and $t\log t < 0$, so the change is negative for small t. This contradicts minimality. ■

Strict positivity says that the optimal plan is dense. It does not say that every antibody residue touches every epitope residue. A discrete contact map is extracted by a threshold or a matching,

and a contact claim must report how it depends on the threshold and on the retained mass $\sum_{ij} \Gamma^{\star}_{ij}$.

Theorem 4.3 (dual problem and scaling equations). With $\tau_{\mathrm{OT}} = \rho/(\rho + \varepsilon)$, strong duality holds:

$$\mathrm{OT}_{\varepsilon,\rho} = \max_{\boldsymbol{a},\boldsymbol{b}} \Bigg\{ -\rho \sum_i \mu_{R,i} \left(e^{-a_i/\rho} - 1\right) - \rho \sum_j \mu_{E,j} \left(e^{-b_j/\rho} - 1\right) - \varepsilon \sum_{ij} \mu_{R,i}\, \mu_{E,j} \left(e^{(a_i + b_j - C_{ij})/\varepsilon} - 1\right) \Bigg\}.$$

The dual objective is strictly concave, and at its maximiser $(\boldsymbol{a}^{\star}, \boldsymbol{b}^{\star})$ the primal plan is $\Gamma^{\star}_{ij} = \mu_{R,i}\mu_{E,j} \exp\{(a_i^{\star} + b_j^{\star} - C_{ij})/\varepsilon\}$. The maximiser satisfies

$$a_i = -\tau_{\mathrm{OT}} \varepsilon \log \sum_j \mu_{E,j} \exp\left\{\frac{b_j - C_{ij}}{\varepsilon}\right\}, \qquad b_j = -\tau_{\mathrm{OT}} \varepsilon \log \sum_i \mu_{R,i} \exp\left\{\frac{a_i - C_{ij}}{\varepsilon}\right\}.$$

Proof. Introduce free marginal variables $r = \Gamma \mathbf{1}$ and $c = \Gamma^{\top} \mathbf{1}$ with multipliers $\boldsymbol{a}$ and $\boldsymbol{b}$, and form the Lagrangian $\mathcal{F}$ with the terms $\boldsymbol{a}^{\top}(r - \Gamma \mathbf{1}) + \boldsymbol{b}^{\top}(c - \Gamma^{\top} \mathbf{1})$ added. Minimise over each r_i, each c_j and each Γ_{ij} separately, using Lemma 4.1(b) with $(\lambda, \sigma) = (\rho, -a_i)$, $(\rho, -b_j)$ and $(\varepsilon, a_i + b_j - C_{ij})$. This gives the dual objective, and the minimiser over Γ_{ij} gives the plan. The primal problem is convex with finite value and a feasible point in the interior of its domain, so there is no duality gap. Each dual term is strictly concave in its own variable, so the objective is strictly concave. Setting $\partial / \partial a_i$ to zero gives $\mu_{R,i} e^{-a_i/\rho} = \sum_j \mu_{R,i}\, \mu_{E,j} e^{(a_i + b_j - C_{ij})/\varepsilon}$. Take logarithms and use $1/\rho + 1/\varepsilon = 1/(\tau_{\mathrm{OT}} \varepsilon)$. The equation for b_j is identical. ■

The two equations are the log-domain form of generalised Sinkhorn scaling. The sums are evaluated with the shifted log-sum-exp of Section S1.8, and convergence is certified by the gap between primal and dual values. For one site, $n_R = n_E = 1$, stationarity reads $C + \rho \log(\gamma/\mu_R) + \rho \log(\gamma/\mu_E) + \varepsilon \log\{\gamma/(\mu_R \mu_E)\} = 0$, so $\gamma^{\star} = \exp\{[(\rho + \varepsilon)(\log \mu_R + \log \mu_E) - C]/(2\rho + \varepsilon)\}$. For $\mu_R = \mu_E = 1$ the optimal value is $(2\rho + \varepsilon)(1 - \gamma^{\star})$.

Proposition 4.4 (envelope gradient). Let $C(\vartheta)$, $\mu_R(\vartheta)$ and $\mu_E(\vartheta)$ be differentiable, with the assumptions of Theorem 4.2 on a neighbourhood. Let $r = \Gamma^{\star} \mathbf{1}$, $c = \Gamma^{\star \top} \mathbf{1}$, $m_R = \sum_i \mu_{R,i}$ and $m_E = \sum_j \mu_{E,j}$. Then $T(\vartheta) = \mathrm{OT}_{\varepsilon,\rho}$ is differentiable and

$$\nabla_\vartheta T = \langle \Gamma^\star, \nabla_\vartheta C \rangle + \sum_i \left[\rho \left(1 - \frac{r_i}{\mu_{R,i}} \right) + \varepsilon \left(m_E - \frac{r_i}{\mu_{R,i}} \right) \right] \nabla_\vartheta \mu_{R,i}$$
$$+ \sum_j \left[\rho \left(1 - \frac{c_j}{\mu_{E,j}} \right) + \varepsilon \left(m_R - \frac{c_j}{\mu_{E,j}} \right) \right] \nabla_\vartheta \mu_{E,j}.$$

For constant reference masses, $\nabla_\vartheta T = \langle \Gamma^\star, \nabla_\vartheta C \rangle$. Note that $r_i/\mu_{R,i} = e^{-a_i^\star/\rho}$.

Proof. The value is $T(\vartheta) = \min_\Gamma \mathcal{F}(\Gamma; \vartheta)$. The minimiser is unique and, by strict convexity and local uniform coercivity, depends continuously on ϑ. Danskin's theorem therefore gives $\nabla_\vartheta T = \partial_\vartheta \mathcal{F}(\Gamma^\star; \vartheta)$. For fixed Γ: $\partial \mathcal{F} / \partial C_{ij} = \Gamma_{ij}$; $\partial \mathrm{KL}(r \mid \mu_R) / \partial \mu_{R,i} = 1 - r_i/\mu_{R,i}$; and $\partial \mathrm{KL}(\Gamma \mid \mu_R \otimes \mu_E) / \partial \mu_{R,i} = \sum_j (\mu_{E,j} - \Gamma_{ij}/\mu_{R,i}) = m_E - r_i/\mu_{R,i}$. The same holds for μ_E, and the second marginal term does not involve μ_R. ■

Proposition 4.5 (regularisation gap). Let $H(\Gamma) = \mathrm{KL}(\Gamma \mid \mu_R \otimes \mu_E)$, let Γ_ε minimise $\mathcal{F}$, and let Γ_0 be any minimiser of $\mathcal{F} - \varepsilon H$ (it exists by the coercivity argument of Theorem 4.2). Then

$$0 \le \varepsilon H(\Gamma_\varepsilon) \le \mathrm{OT}_{\varepsilon,\rho} - \mathrm{OT}_{0,\rho} \le \varepsilon H(\Gamma_0).$$

Proof. Put $\mathcal{F}_0 = \mathcal{F} - \varepsilon H$, so that $\mathcal{F} = \mathcal{F}_0 + \varepsilon H$ and $H \ge 0$. Optimality of Γ_0 gives $\mathcal{F}_0(\Gamma_\varepsilon) \ge \mathrm{OT}_{0,\rho}$, so $\mathrm{OT}_{\varepsilon,\rho} = \mathcal{F}_0(\Gamma_\varepsilon) + \varepsilon H(\Gamma_\varepsilon) \ge \mathrm{OT}_{0,\rho} + \varepsilon H(\Gamma_\varepsilon)$. Optimality of Γ_ε gives $\mathrm{OT}_{\varepsilon,\rho} \le \mathcal{F}(\Gamma_0) = \mathrm{OT}_{0,\rho} + \varepsilon H(\Gamma_0)$. $H(\Gamma_0)$ is finite because Γ_0 is bounded and the reference is positive. ■

The bound grows with the number of reference pairs, so ε must shrink as the interface grows to hold a given gap.

Proposition 4.6 (soft plans and integer contacts). (a) A binary contact map with given integer valences for each residue and exactly k contacts that minimises a linear cost exists whenever one is feasible, because it is a minimum-cost flow problem, and flow problems with integer capacities have integer optima (standard). This controls the linear part of the objective. (b) No bound on the distance between $\Gamma^\star$ and a binary map follows from ε, ρ and $\| C \|_\infty$ alone: take $n_R = n_E = n$, $\mu_R = \mu_E = \mathbf{1}/n$ and $C = 0$. Then $\Gamma^\star_{ij} = 1/n^2$ for all $\rho, \varepsilon > 0$, and for every $S \in \{0,1\}^{n \times n}$ with $|S|$ unit entries,

$$\| \Gamma^\star - S \|_1 = |S| + 1 - \frac{2|S|}{n^2} \ge 1 = \| \Gamma^\star \|_1.$$

A permutation matrix gives $n + 1 - 2/n$. A permutation scaled to the plan's mass, $1/n\, P$, gives $2(1 - 1/n)$, which tends to the largest possible value 2.

Proof. (b) The first-order condition of $\mathcal{F}$ is $C_{ij} + \varepsilon \log \frac{\Gamma_{ij}}{\mu_{R,i}\mu_{E,j}} + \rho \log \frac{r_i}{\mu_{R,i}} + \rho \log \frac{c_j}{\mu_{E,j}} = 0$. At $\Gamma_{ij} = 1/n^2$ every ratio equals 1, so all logarithms vanish and $C = 0$ holds. By uniqueness this is the minimiser. Entries with $S_{ij} = 1$ contribute $1 - 1/n^2$ each, and the other $n^2 - |S|$ entries contribute $1/n^2$ each, which gives the formula. The bound $|S|(1 - 2/n^2) \geq 0$ holds for $n \geq 2$. For $1/n\,P$ the n unit positions contribute $(n-1)/n^2$ each and the other $n^2 - n$ entries $1/n^2$ each. ■

Particle systems for a guided stochastic sampler. Fix a horizon M and a state space $\mathcal{X}$. Let p_0 be the initial law and let $\mathsf{K}_j(q_{j-1}, \mathrm{d}q_j)$ be the reference transition kernels, $j = 1, \ldots, M$. Let $\Psi_0 \colon \mathcal{X} \to (0, C_\Psi]$ and $\Psi_j \colon \mathcal{X}^2 \to (0, C_\Psi]$ be bounded positive potentials. The unnormalised path measure, its total mass and its normalised form are

$$\gamma_M(\mathrm{d}q_{0:M}) = p_0(\mathrm{d}q_0)\Psi_0(q_0) \prod_{j=1}^{M} \mathsf{K}_j\,(q_{j-1}, \mathrm{d}q_j)\Psi_j(q_{j-1}, q_j), \qquad Z_M = \gamma_M(1), \qquad \eta_M = \frac{\gamma_M}{Z_M}.$$

The default of the Methods is $\Psi_0 \equiv 1$, which this supplement keeps as a special case. A proposal law R_0 and proposal kernels R_j replace the reference laws in the simulation. They must dominate: $p_0 \ll \mathsf{R}_0$ and $\mathsf{K}_j(q,\cdot) \ll \mathsf{R}_j(q,\cdot)$. The incremental weights are

$$\omega_0 = \Psi_0 \, \frac{\mathrm{d}p_0}{\mathrm{d}\mathsf{R}_0}(q_0), \qquad \omega_j = \Psi_j(q_{j-1}, q_j) \, \frac{\mathrm{d}\mathsf{K}_j(q_{j-1},\cdot)}{\mathrm{d}\mathsf{R}_j(q_{j-1},\cdot)}(q_j).$$

If a proposal assigns zero probability to a measurable set with positive reference probability, the domination condition fails and importance weighting cannot recover the missing mass. Proposal draws outside the reference support receive zero weight.

The system has N_p particles. Initial particles $q_0^{(k)}$ are drawn independently from R_0 with carried weights $\widetilde{w}_{-1}^{(k)} = 1/N_\mathrm{p}$. Before each step $j \geq 1$ the system either resamples or holds, and this choice may depend on everything observed up to step $j-1$. After a resampling step the ancestors a_k have offspring counts N_i with $\mathbb{E}[N_i \mid \mathcal{G}_{j-1}] = N_\mathrm{p} \bar{w}_{j-1}^{(i)}$ (conditional unbiasedness), and the carried weights are $\widetilde{w}_{j-1}^{(k)} = 1/N_\mathrm{p}$. After a held step, $a_k = k$ and $\widetilde{w}_{j-1}^{(k)} = \bar{w}_{j-1}^{(k)}$, the normalised weight. Then $q_j^{(k)} \sim \mathsf{R}_j(q_{j-1}^{(a_k)}, \cdot)$ and

$$w_j^{(k)} = \widetilde{w}_{j-1}^{(k)}\, \omega_j(q_{j-1}^{(a_k)}, q_j^{(k)}), \qquad \bar{\ell}_j = \sum_{k=1}^{N_\mathrm{p}} w_j^{(k)}, \qquad \bar{w}_j^{(k)} = \frac{w_j^{(k)}}{\bar{\ell}_j}, \qquad \hat{Z}_M = \prod_{j=0}^{M} \bar{\ell}_j.$$

For $j = 0$ there is no ancestor and ω_0 depends on $q_0^{(k)}$ only. If some $\bar{\ell}_j = 0$, put $\hat{Z}_M = 0$. The factor $\bar{\ell}_j$ carries the previous weights. After a resampling step it reduces to the plain mean $N_\mathrm{p}^{-1} \sum_k \omega_j$.

One proposal step followed by one resampling step leaves the expected weighted measure unchanged, and the proof is an induction on this fact.

Theorem 4.7 (unbiasedness). Under the assumptions above, for every $N_\mathrm{p} \geq 1$, every adaptive resampling schedule and every bounded measurable path function φ,

$$\mathbb{E}\left[\hat{Z}_M \sum_{k=1}^{N_\mathrm{p}} \bar{w}_M^{(k)}\ \varphi\left(q_{0:M}^{(k)}\right)\right] = \gamma_M(\varphi),$$

and in particular,

$$\mathbb{E}\left[\hat{Z}_M\right] = Z_M.$$

Here $q_{0:M}^{(k)}$ is the ancestral path of particle k. Negative association of the offspring counts is not needed.

Proof. Put $\hat{\gamma}_j(\varphi) = \hat{Z}_j \sum_k \bar{w}_j^{(k)}\, \varphi(q_{0:j}^{(k)})$ and take $\varphi \geq 0$ first. For $j = 0$, $\mathbb{E}\hat{\gamma}_0(\varphi) = \int \mathsf{R}_0(\mathrm{d}q_0)\, \omega_0\, \varphi = \int p_0(\mathrm{d}q_0) \Psi_0 \varphi = \gamma_0(\varphi)$ by the Radon–Nikodym theorem. For $j \geq 1$ note that $\hat{Z}_j \bar{w}_j^{(k)} = \hat{Z}_{j-1} w_j^{(k)}$, so

$$\hat{\gamma}_j(\varphi) = \hat{Z}_{j-1} \sum_k \widetilde{w}_{j-1}^{(k)}\ \omega_j\left(q_{j-1}^{(a_k)}, q_j^{(k)}\right) \varphi\left(q_{0:j-1}^{(a_k)}, q_j^{(k)}\right).$$

Given the history and the ancestors, $q_j^{(k)} \sim \mathsf{R}_j$, and the change of measure gives $\int \mathsf{R}_j(q, \mathrm{d}q')\, \omega_j\, \varphi = \int \mathsf{K}_j(q, \mathrm{d}q') \Psi_j(q, q') \varphi =: (\mathsf{K}_j^{\Psi}\varphi)(q)$. So the conditional expectation is $\hat{Z}_{j-1} \sum_k \widetilde{w}_{j-1}^{(k)}\, (\mathsf{K}_j^{\Psi}\varphi)(q_{0:j-1}^{(a_k)})$. If the step is held, this equals $\hat{\gamma}_{j-1}(\mathsf{K}_j^{\Psi}\varphi)$. If the step resamples, the average over the offspring counts is $N_\mathrm{p}^{-1} \sum_i \mathbb{E}\,[N_i]\, F(q^{(i)}) = \sum_i \bar{w}_{j-1}^{(i)}\, F(q^{(i)})$ with $F = \mathsf{K}_j^{\Psi}\varphi$, which is again $\hat{\gamma}_{j-1}(\mathsf{K}_j^{\Psi}\varphi)$. The schedule is a function of the history, so the two cases combine. The induction hypothesis gives $\mathbb{E}\hat{\gamma}_{j-1}(\mathsf{K}_j^{\Psi}\varphi) = \gamma_{j-1}(\mathsf{K}_j^{\Psi}\varphi) = \gamma_j(\varphi)$. For $\varphi \geq 0$, Tonelli's theorem justifies every exchange of sum and integral even before finiteness is known,

and $\gamma_j(1) \le C_\Psi^{j+1}$ is finite. A signed bounded φ is split into positive and negative parts. The choice $\varphi = 1$ gives the last claim, with the convention $\hat{Z}_M = 0$ whenever a step factor vanishes. ■

The logarithm of $\hat{Z}_M$ is biased low, by Jensen's inequality, and a ratio such as $\hat{\eta}_M(\varphi)$ is biased at finite N_p. The next two examples show how large the effects can be and why the carried weights matter.

Example 4.8 (carried weights). Let p_0 be uniform on $\{0,1\}$, let the kernels and proposals be the identity map, let $\Psi_0 \equiv 1$ and $\Psi_1 = \Psi_2 = (1,2)$, and let $N_\mathrm{p} = 2$ with no resampling. The four equally likely initial clouds give products of plain means $1,\ 2.25,\ 2.25,\ 4$, with mean 2.375. The target is $\mathbb{E}[\Psi_1\Psi_2] = 2.5$. The carried-weight estimator returns $1,\ 2.5,\ 2.5,\ 4$ with mean 2.5.

Example 4.9 (exact finite-particle bias). Let $X \in \{0,1\}$ be equally likely, with potential $\Psi(0) = 1, \Psi(1) = 3$ and the reference kernel as proposal. With n independent particles and $K \sim \mathrm{Binomial}(n, 1/2)$ the number with $X = 1$, the estimates are $\hat{Z} = 1 + 2K/n$ and $\hat{\pi}(1) = 3K/(n + 2K)$. The truth is $Z = 2$ and $\pi(1) = 3/4$. Exact summation over the $n + 1$ outcomes gives **Supplementary Table S2**. The estimate $\hat{Z}$ is unbiased, while its logarithm and the ratio are biased low, and both biases shrink about like $1/n$. Averaging independent runs leaves these biases unchanged, so a particle budget cannot be split into small runs without cost.

n	$\mathbb{E}\hat{Z}$	$\log 2 - \mathbb{E}\log\hat{Z}$	$3/4 - \mathbb{E}\hat{\pi}(1)$
1	2	0.143841	0.250000
4	2	0.034115	0.056250
16	2	0.007997	0.012287
64	2	0.001965	0.002964

Supplementary Table S2. Exact bias of the normalising-constant estimate, of its logarithm and of the ratio estimate in Example 4.9.

The values come from exact summation over the outcomes.

Define the backward information function by $h_M \equiv 1$ and $h_{j-1}(q) = \int \mathsf{K}_j(q, \mathrm{d}q')\Psi_j(q, q')h_j(q')$ for $j = M, \ldots, 1$. Since the potentials are positive and bounded, $0 < h_j \le C_\Psi^{M-j}$. The twisted kernels and potentials are

$$\mathsf{K}_j^h(q, \mathrm{d}q') = \frac{\mathsf{K}_j(q, \mathrm{d}q')\Psi_j(q, q')h_j(q')}{h_{j-1}(q)}, \qquad \Psi_0^h = \Psi_0 h_0, \qquad \Psi_j^h(q, q') = \Psi_j(q, q')\frac{h_j(q')}{h_{j-1}(q)}.$$

Theorem 4.10 (zero variance under perfect twisting). With these definitions, $Z_M = \int p_0(\mathrm{d}q_0)\Psi_0(q_0)h_0(q_0)$, and $\Psi_0^h \prod_{j=1}^M \Psi_j^h = \Psi_0 \prod_{j=1}^M \Psi_j$ along every path, so γ_M and Z_M are unchanged. Run the sampler for the potentials Ψ^h with the initial proposal $\mathsf{R}_0(\mathrm{d}q_0) = p_0(\mathrm{d}q_0)\Psi_0(q_0)h_0(q_0)/Z_M$ and the proposals $\mathsf{R}_j = \mathsf{K}_j^h$. The weights are constant:

$$\omega_0(q_0^{(k)}) \equiv Z_M, \qquad \omega_j(q_{j-1}^{(k)}, q_j^{(k)}) \equiv 1 \qquad (j = 1, \dots, M).$$

Consequently, for every $N_\mathrm{p} \geq 1$, every resampling rule and every schedule, $\hat{Z}_M = Z_M$ on every realisation, and $\mathrm{Var}(\hat{Z}_M) = 0$. All normalised weights equal $1/N_\mathrm{p}$. The equality holds for the explicit versions of h_j and of the Radon–Nikodym derivatives; for other versions it holds almost surely.

Proof. The recursion for h_j and positivity give finite positive ratios h_j/h_{j-1}. *Normalisation.* By the recursion, $\int \mathsf{K}_j(q, \mathrm{d}q')\Psi_j(q, q')h_j(q') = h_{j-1}(q)$, so $\mathsf{K}_j^h(q,\cdot)$ has mass 1. Integrating γ_M over $q_M, q_{M-1}, \dots, q_1$ in turn with the same recursion gives $Z_M = \int p_0\Psi_0h_0$, so R_0 is a probability law. *Telescoping.* The factors h_j cancel in the product of twisted potentials, and $h_M \equiv 1$ removes the last one. *Derivatives.* $\mathsf{K}_j^h(q,\cdot)$ has the positive density $\Psi_j h_j/h_{j-1}$ with respect to $\mathsf{K}_j(q,\cdot)$, and R_0 has the positive density $\Psi_0 h_0/Z_M$ with respect to p_0. So the measures are mutually absolutely continuous, with $\mathrm{d}\mathsf{K}_j/\mathrm{d}\mathsf{K}_j^h = h_{j-1}/(\Psi_j h_j)$ and $\mathrm{d}p_0/\mathrm{d}\mathsf{R}_0 = Z_M/(\Psi_0 h_0)$. *Weights.* $\omega_0 = \Psi_0 h_0 \cdot Z_M/(\Psi_0 h_0) = Z_M$ and $\omega_j = (\Psi_j h_j/h_{j-1}) \cdot h_{j-1}/(\Psi_j h_j) = 1$. The carried weights sum to one, so $\bar{\ell}_0 = Z_M$ and $\bar{\ell}_j = 1$ for $j \geq 1$ whatever the schedule, and $\hat{Z}_M = Z_M$. ■

If the original potentials are used with the same proposals, the weights are not constant. The initial weight is $Z_M/h_0(q_0)$ and the step weights are $h_{j-1}(q_{j-1})/h_j(q_j)$. Their product along any path equals Z_M, so a sampler without resampling has equal cumulative weights. Resampling by a single factor adds randomness, and the variance of $\hat{Z}_M$ is then positive in general (Example 4.11). Zero variance is a property of $\hat{Z}_M$ alone. Without resampling the particles are an independent sample from η_M, and $\eta^N(\varphi)$ keeps the variance $\mathrm{Var}_{\eta_M}(\varphi)/N_\mathrm{p}$ for $\varphi \in L^2(\eta_M)$.

In practice h_j is out of reach, and a learned $\hat{h}_j > 0$ with $\hat{h}_M \equiv 1$ replaces it. The twisted potentials built from $\hat{h}$ have the same product along every path, so the target is unchanged. Theorem 4.7 applies to them, and $\hat{Z}_M$ stays unbiased for any $\hat{h}$. The weights equal one only if

$\hat{h} = h$ and the proposals are K_j^h. That a closer $\hat{h}$ gives a smaller variance is a tendency and not a theorem.

Example 4.11 (resampling by the untwisted weights). Let $\mathcal{X} = \{a, b\}$, $M = 1$, p_0 uniform, $\Psi_0 \equiv 1$, $\mathsf{K}_1(q,\cdot) = \delta_q$, and $\Psi_1(q, q') = g(q)$ with $g(a) = 1$ and $g(b) = 3$. Then $h_1 \equiv 1$, $h_0 = g$, $Z_1 = 2$, $\mathsf{R}_0 = (1/4, 3/4)$ and $\mathsf{K}_1^h = \mathsf{K}_1$. The untwisted weights are $\omega_0 = Z_1/g = (2, 2/3)$ and $\omega_1 = g = (1,3)$, with product 2 on both paths. Take $N_\mathrm{p} = 2$ and resample by multinomial resampling before step 1. The clouds (a, a) and (b, b) have probabilities $1/16$ and $9/16$, and in both $\hat{Z} = 2$. The mixed cloud has probability $6/16$, $\bar{\ell}_0 = 4/3$ and normalised weights $(3/4, 1/4)$. Its two offspring are (a, a), (a, b) or (b, b) with probabilities $9/16$, $6/16$, $1/16$, so $\bar{\ell}_1 = 1{,}2{,}3$ and $\hat{Z} = 4/3, 8/3, 4$. Hence $\mathbb{E}\hat{Z} = 2$, $\mathbb{E}\hat{Z}^2 = 17/4$ and $\mathrm{Var}\hat{Z} = 1/4$. With the twisted weights $\omega_0 = 2$ and $\omega_1 = 1$, the estimate equals 2 in every case.

Proposition 4.12 (mean-unbiased resampling does not give consistency). Let $\mathcal{X} = \{0{,}1\}$, $M = 2$, p_0 uniform, $\Psi_0 \equiv \Psi_1 \equiv 1$, $\Psi_2(q, q') = g(q)$ with $g(0) = 1/3$ and $g(1) = 1$, and let all kernels and proposals be the identity. Before every step the system draws one index I with probabilities $\bar{w}$ and gives all N_p offspring to it. Then, given the past, $\mathbb{E}[N_i] = N_\mathrm{p}\bar{w}^{(i)}$, $\mathbb{E}\hat{Z}_2 = Z_2 = 2/3$, and $\hat{\eta}_2(\mathrm{id})$ is 0 or 1 with probability $1/2$ each. The target value is $\eta_2(\mathrm{id}) = 3/4$, so the bias of the normalised estimate is exactly $-1/4$ for every N_p. The effective sample size equals N_p after every step.

Proof. The offspring means hold by construction, and the claim on $\hat{Z}_2$ is Theorem 4.7. After step 1 all particles equal $q_0^{(I)}$, where I is uniform because the weights are equal, so $q_0^{(I)}$ is 0 or 1 with probability $1/2$. At step 2 all particles carry the same weight $g(q_0^{(I)})$, so the normalised weights are uniform and $\hat{\eta}_2(\mathrm{id}) = q_0^{(I)}$. Hence $\mathbb{E}\hat{\eta}_2 = 1/2$. The target is $\eta_2(\mathrm{id}) = 1/2 \cdot 1/(1/2 \cdot 1/3 + 1/2 \cdot 1) = 3/4$. All weights are equal at every step, so the effective sample size is N_p throughout. ■

The example shows that conditional unbiasedness is too weak to support a claim about the weighted cloud, and that the effective sample size cannot detect such a failure.

Proposition 4.13 (systematic resampling is not negatively associated). Let $N_\mathrm{p} = 4$ and $\bar{w} = (1{,}1{,}1{,}5)/8$. Systematic resampling draws $U \sim \mathcal{U}[0, 1/4)$ and uses the points $U + (i-1)/4$. Then $N_1 = N_3 = \mathbf{1}\{U < 1/8\}$, $N_2 = \mathbf{1}\{U \geq 1/8\}$ and $N_4 = 2 + \mathbf{1}\{U \geq 1/8\}$. So $\mathbb{P}(N_1 = 1, N_3 = 1) = 1/2$ while $\mathbb{P}(N_1 = 1)\,\mathbb{P}(N_3 = 1) = 1/4$, and $\mathrm{Cov}(N_1, N_3) = 1/4 > 0$.

Proof. The cumulative weights are $(1/8, 2/8, 3/8, 1)$. The point U lies in $[0, 1/8)$ exactly when $U < 1/8$ (particle 1) and otherwise in $[1/8, 1/4)$ (particle 2). The point $U + 1/4 \in [1/4, 1/2)$ lies in $[2/8, 3/8)$ (particle 3) exactly when $U < 1/8$ and otherwise in $[3/8, 1/2)$ (particle 4). The points $U + 1/2$ and $U + 3/4$ lie in $[1/2, 1)$ (particle 4). ■

The expected counts are right ($\mathbb{E}N_1 = 1/2 = 4\bar{w}_1$), so systematic resampling is conditionally unbiased, but its counts move together, and the consistency theorem below does not cover it for unordered particles. Consistency is recovered when the particles are ordered along a coordinate [3]. The Methods use no such ordering for the packed state, so a run that uses systematic resampling treats its weighted cloud as a diagnostic. The resampling laws with proved conditions are multinomial, residual and stratified resampling.

Theorem 4.14 (consistency; sketch, with the resampling step cited). Let the assumptions of Theorem 4.7 hold, let the Radon–Nikodym derivatives $\mathrm{d}p_0/\mathrm{dR}_0$ and $\mathrm{dK}_j/\mathrm{dR}_j$ be bounded, and let $\mathcal{X}$ be a cubifiable subset of $\mathbb{R}^d$ (the condition on the state space in the cited theorem). Let the system resample before every step with a scheme that meets the conditions of Theorem 1 of [3] (multinomial, residual and stratified resampling do). Then for every bounded measurable φ, $\hat{\eta}_M(\varphi) = \sum_k \bar{w}_M^{(k)}\, \varphi(q_M^{(k)}) \to \eta_M(\varphi)$ in probability as $N_\mathrm{p} \to \infty$.

Sketch. At $j = 0$ the law of large numbers gives convergence of the weighted empirical measure for bounded functions. Assume convergence at $j - 1$. Selection replaces the empirical measure by its weighted version. The map from a measure η to $\eta(G\,\mathrm{K}\varphi)/\eta(G)$ is a ratio whose denominator converges to a positive constant, so the continuous mapping theorem gives convergence of the selected measure. The cited theorem gives that resampling preserves the limit. Given the ancestors, the mutations are conditionally independent with kernels K_j and weights ω_j, and the conditional law of large numbers gives the update. A finite induction reaches step M. The normalising constant factors as a finite product of integrals under the previous laws, so each factor of $\hat{Z}_M$ converges and so does the product. ■

No central limit theorem and no explicit constant for the bias of order $1/N_\mathrm{p}$ are given. A bias bound of that order needs a bounded weight, which fails for most guided proposals, and Proposition 4.12 shows that a deviation condition on the offspring cannot be dropped.

Proposition 4.15 (effective sample size). For positive weights w_k with mean $\bar{w}$ and normalised weights $\bar{w}_k = w_k / \sum_l w_l$, let $\mathrm{ESS} = 1/\sum_k \bar{w}_k^2$ and $\mathrm{CV}^2 = N^{-1} \sum_k (w_k - \bar{w})^2/\bar{w}^2$. Then:

1. $\mathrm{ESS} = N/(1 + \mathrm{CV}^2)$.
2. $1 \le \mathrm{ESS} \le N$ and $1/\mathrm{ESS} \le \max_k \bar{w}_k \le 1/\sqrt{\mathrm{ESS}}$.
3. If $w_1, \ldots, w_N$ are independent with $\log w_k \sim \mathcal{N}(\mu, s^2)$, then $\mathrm{ESS}/N \to e^{-s^2}$ in probability as $N \to \infty$. This is an asymptotic statement for an idealised model and not an identity at finite N.
4. Immediately after resampling the weights are all $1/N$ and $\mathrm{ESS} = N$, whatever the number of distinct ancestors. Resampling that copies one particle N times leaves $\mathrm{ESS} = N$ with one ancestor.

Proof. (1) Expanding the variance gives $N^{-1}\sum w_k^2 = \bar{w}^2(1 + \mathrm{CV}^2)$, and $\sum_l w_l = N\bar{w}$, so $\sum_k \bar{w}_k^2 = (1 + \mathrm{CV}^2)/N$. (2) By Cauchy–Schwarz, $1 = (\sum_k \bar{w}_k)^2 \le N \sum_k \bar{w}_k^2$, so $\mathrm{ESS} \le N$. Also $\sum_k \bar{w}_k^2 \le \max_k \bar{w}_k \sum_k \bar{w}_k = \max_k \bar{w}_k \le 1$, which gives $\mathrm{ESS} \ge 1$ and $1/\mathrm{ESS} \le \max_k \bar{w}_k$. Finally $\max_k \bar{w}_k^2 \le \sum_k \bar{w}_k^2$. (3) The law of large numbers gives $N^{-1}\sum_k w_k \to e^{\mu + s^2/2}$ and $N^{-1}\sum_k w_k^2 \to e^{2\mu + 2s^2}$, and ESS/N is the ratio of the square of the first to the second. (4) is the statement of Proposition 4.12 for the effective sample size. ■

Because of item 4, the number of distinct ancestral lineages that survive from step 0 is recorded next to the effective sample size, together with the spread of the decoded structures and the variance of $\log \hat{Z}_M$ across independent complete runs. These are diagnostics, and no theorem here turns them into an error bound. Independent complete runs give an empirical standard error that does not depend on the genealogy. The Methods plan at least 32 independent complete runs.

No proposition governs a deterministic guided flow. Distinct deterministic kernels are point masses on different updates and are generally mutually singular, so the domination condition of Theorem 4.7 fails. The particle results above apply to a stochastic transition profile only.

S1.5 Decision statistics and calibration

This section states the guarantees that the Methods use for ambiguity over plausible worlds, coverage, conformal calibration, exact count bounds and sequential evidence. Each guarantee holds under named conditions, which the text gives beside it. The results concern finite sets of numbers, weights and probabilities. **Supplementary Figure S5** and **Supplementary Figure S6** illustrate the cautious score under ambiguity and the exact-bound examples.

Ambiguity over plausible worlds. Let the worlds $j = 1, \ldots, M$ be surrogate checkpoints or assay mappings, with reference weights $\pi_j > 0$ that sum to one. Each world gives a dimensionless utility U_j of one design. The entropic pessimistic utility is $U_{\mathrm{ent}} = -\eta^{-1} \log \sum_j$

$\pi_j\, e^{-\eta U_j}$ for $\eta > 0$. Write Δ_M for the probability vectors of length M and $\mathrm{KL}(\nu \mid \pi) = \sum_j \nu_j \log(\nu_j/\pi_j)$.

Proposition 5.1 (entropic identity). Put $Z = \sum_j \pi_j\, e^{-\eta U_j}$ and $\nu_j^\star = \pi_j e^{-\eta U_j}/Z$. For every $\nu \in \Delta_M$,

$$\sum_j \nu_j\, U_j + \frac{1}{\eta}\mathrm{KL}(\nu \mid \pi) = U_{\mathrm{ent}} + \frac{1}{\eta}\mathrm{KL}(\nu \mid \nu^\star).$$

Hence $U_{\mathrm{ent}} = \min_{\nu\in\Delta_M}\{\sum_j \nu_j\, U_j + \eta^{-1}\mathrm{KL}(\nu \mid \pi)\}$, the minimum is attained only at $\nu^\star$, and $\min_j U_j \le U_{\mathrm{ent}} \le \sum_j \pi_j\, U_j$.

Proof. Expand $\mathrm{KL}(\nu \mid \nu^\star) = \sum_j \nu_j \log(\nu_j/\pi_j) + \eta \sum_j \nu_j\, U_j + \log Z$ and rearrange, using $U_{\mathrm{ent}} = -\eta^{-1}\log Z$. Since $\mathrm{KL}(\nu \mid \nu^\star) \ge 0$ with equality only at $\nu = \nu^\star$, the minimum follows. The choice $\nu = \pi$ gives the upper bound. Also $Z \le e^{-\eta \min_j U_j}$, which gives the lower bound. ■

Numerical illustration: for $\pi = (0.5,0.5)$, $U = (0.9,0.3)$ and $\eta = 2$, the formula gives $U_{\mathrm{ent}} = 0.5149$ and $\nu^\star = (0.2315,0.7685)$. The weighted mean utility is 0.4389 and the penalty term is 0.0760, which add to U_{ent}. The plain mean is 0.6.

Proposition 5.2 (hard KL ball). For $\delta > 0$,

$$\min_{\nu\in\Delta_M,\ \mathrm{KL}(\nu|\pi)\le\delta} \sum_j \nu_j\, U_j = \sup_{\lambda\ge 0}\left\{-\lambda \log \sum_j \pi_j\, e^{-U_j/\lambda} - \lambda\delta\right\},$$

where the term at $\lambda \downarrow 0$ is read as $\min_j U_j$.

Proof. The point $\nu = \pi$ has $\mathrm{KL} = 0 < \delta$, so the convex problem has a strictly feasible point and no duality gap. The Lagrangian is $\sum_j \nu_j\, U_j + \lambda(\mathrm{KL}(\nu \mid \pi) - \delta)$. For $\lambda > 0$, Proposition 5.1 with $\eta = 1/\lambda$ gives its minimum over ν as $-\lambda \log \sum_j \pi_j\, e^{-U_j/\lambda} - \lambda\delta$. As $\lambda \downarrow 0$ the first term tends to $\min_j U_j$. ■

The penalised form fixes the multiplier at $1/\eta$. The ball form optimises it, so the two describe different families of adversaries. A Wasserstein ball around a nominal law has a duality theorem only under conditions on the cost and the utility: lower semicontinuity, integrability and a growth or compactness bound [4]. That theorem is cited and is not proved here. Each utility must be checked against its conditions before the dual form is used.

Coverage and greedy selection. A portfolio of N designs should cover several targets without spending effort on a target that is already covered. With target weights $w_j \geq 0$ and binding inputs $p_j(X) \geq 0$, the coverage of a set $\mathcal{B}$ is $F_{\mathrm{cov}}(\mathcal{B}) = \sum_j w_j \log(1 + S_j(\mathcal{B}))$, where $S_j(\mathcal{B}) = \sum_{X \in \mathcal{B}} p_j(X)$. The gain from adding X is $\Delta_X F_{\mathrm{cov}}(\mathcal{B}) = F_{\mathrm{cov}}(\mathcal{B} \cup \{X\}) - F_{\mathrm{cov}}(\mathcal{B})$.

Proposition 5.3 (diminishing returns). $F_{\mathrm{cov}}(\varnothing) = 0$, F_{cov} is monotone, and for $\mathcal{A} \subseteq \mathcal{B}$ and $X \notin \mathcal{B}$,

$$\Delta_X F_{\mathrm{cov}}(\mathcal{A}) \;\geq\; \Delta_X F_{\mathrm{cov}}(\mathcal{B}) \;\geq\; 0.$$

Proof. The gain is $\sum_j w_j \log\big(1 + p_j(X)/(1 + S_j(\mathcal{B}))\big)$, which is nonnegative. For $\mathcal{A} \subseteq \mathcal{B}$ every $S_j(\mathcal{A}) \leq S_j(\mathcal{B})$, so each denominator grows and each logarithm falls. ■

Proposition 5.4 (greedy bound, cardinality only). Let the pool be finite and the only constraint be $|\mathcal{B}| = N$. Let S_t be the set after t greedy steps, each adding the element of largest gain, and let $S^\star$ be a best set of size N. Then

$$F_{\mathrm{cov}}(S_N) \;\geq\; [1 - (1 - 1/N)^N] F_{\mathrm{cov}}(S^\star) \;\geq\; (1 - e^{-1})\, F_{\mathrm{cov}}(S^\star).$$

Proof. By monotonicity and submodularity, $F(S^\star) \leq F(S_t \cup S^\star) \leq F(S_t) + \sum_{X \in S^\star} \Delta_X F(S_t) \leq F(S_t) + N\,[F(S_{t+1}) - F(S_t)]$, because greedy takes the largest gain. Rearranged, $F(S^\star) - F(S_{t+1}) \leq (1 - 1/N)[F(S^\star) - F(S_t)]$. Iterating from $F(S_0) = 0$ gives the first inequality, and $(1 - 1/N)^N \leq e^{-1}$ gives the second [5]. ■

The proof uses unrestricted choice at each step. Subtracting a nonnegative symmetric zero-diagonal pairwise similarity penalty preserves submodularity while breaking monotonicity, and the collision limit and format constraints of the Methods remove unrestricted choice, so the $(1 - e^{-1})$ bound is not claimed for the constrained portfolio.

Conformal calibration and risk control. A score can be turned into an interval or a threshold with a guarantee that needs no model of the scores. The one assumption is exchangeability: the calibration items and the new item are drawn so that their order carries no information. The guarantee is an average over calibration sets and new items. It does not hold for one calibration set or for one new item.

Theorem 5.5 (split-conformal rank bound). Let $S_1, \ldots, S_{n+1}$ be exchangeable real scores. Let $\alpha \in (0,1)$ and put $k = \lceil (n+1)(1-\alpha) \rceil$. Let $\hat{q}$ be the k-th smallest of $S_1, \ldots, S_n$ if $k \leq n$, and $+\infty$ if $k = n + 1$. Then $\mathbb{P}(S_{n+1} \leq \hat{q}) \geq 1 - \alpha$. If the scores are almost surely distinct and $k \leq n$, the probability equals $k/(n+1)$, which is at most $1 - \alpha + 1/(n+1)$.

Proof. If the scores are distinct, the rank of S_{n+1} among all $n+1$ scores is uniform on $\{1, \dots, n+1\}$, by exchangeability. The event $S_{n+1} \le \hat{q}$ is the event that this rank is at most k, which has probability $k/(n+1) \ge 1-\alpha$. With ties, add independent random tie breakers to obtain a uniform rank. Removing the tie breakers while keeping equality as inclusion can only enlarge the event. The case $k = n+1$ gives probability one. ■

For scores $S_i = |Y_i - \hat{\mu}(X_i)|/\hat{\sigma}(X_i)$ the interval $\hat{\mu}(X_{n+1}) \pm \hat{q}\,\hat{\sigma}(X_{n+1})$ therefore has marginal coverage at least $1-\alpha$. The smallest attainable miscoverage step is $1/(n+1)$. A level $\alpha < 1/(n+1)$ forces the infinite threshold.

Theorem 5.6 (conformal risk control). Let $L_1, \dots, L_{n+1}$ be exchangeable random functions of λ in a compact interval Λ with largest element $\lambda_{\max}$. Assume each L_i is nonincreasing, right-continuous, with values in $[0, B]$, and $L_i(\lambda_{\max}) \le \alpha$ almost surely. Put

$$\hat{\lambda} = \inf\left\{\lambda \in \Lambda\colon \frac{1}{n+1}\left[\sum_{i=1}^{n} L_i(\lambda) + B\right] \le \alpha\right\},$$

with $\hat{\lambda} = \lambda_{\max}$ when the set is empty. Then $\mathbb{E}\, L_{n+1}(\hat{\lambda}) \le \alpha$.

Proof. Define the look-ahead threshold $\lambda^* = \inf\{\lambda\colon (n+1)^{-1}\sum_{i=1}^{n+1} L_i(\lambda) \le \alpha\}$. The set contains $\lambda_{\max}$, and right-continuity makes the infimum attained. Because $L_{n+1} \le B$, every λ feasible for $\hat{\lambda}$ is feasible for λ^*, so $\hat{\lambda} \ge \lambda^*$, including the fallback. Monotonicity gives $L_{n+1}(\hat{\lambda}) \le L_{n+1}(\lambda^*)$. The threshold λ^* is a symmetric function of the $n+1$ losses, so exchangeability gives $\mathbb{E}L_{n+1}(\lambda^*) = \mathbb{E}[(n+1)^{-1}\sum_i L_i(\lambda^*)] \le \alpha$. ■

Four conditions decide whether the theorem applies. First, the loss must be monotone. The joint loss "admitted and bad" fits. The share of bad designs among the admitted does not, because numerator and denominator both change. Second, a rule with two thresholds needs a nested scalar family, either one summary score or a monotone curve in the plane of the two thresholds, because independently tuned thresholds can make the empirical loss rise and void the bound. Third, the admitted set is $\{s > \lambda\}$ with strict inequality, so that the loss is right-continuous. Fourth, an empty feasible set returns $\lambda_{\max}$ and admits nothing, which is recorded as a typed outcome.

The guarantee is marginal. It bounds $\mathbb{P}(\text{admitted and bad}) \le \alpha$ and not the share of bad designs among those admitted. If 1 percent of candidates are admitted and every one is bad, then the bound holds at $\alpha = 0.10$, yet every admitted design is bad. A claim about the admitted set needs

a different tool, such as a multiple-testing procedure on conformal p-values. The rule is the theorem of the cited source [6], proved here at a fixed horizon.

Proposition 5.7 (transfer under a bounded density ratio). Let the new observation have a fixed law Q with $Q \ll P$ and $\mathrm{d}Q/\mathrm{d}P \le W$. If a calibration procedure gives $\mathbb{E}_{\mathrm{cal},P} L \le \alpha$ and $0 \le L \le B$, then $\mathbb{E}_{\mathrm{cal},Q} L \le \min(W\alpha, B)$.

Proof. Given the calibration outcome, change measure: $\mathbb{E}_Q L = \mathbb{E}_P[(\mathrm{d}Q/\mathrm{d}P)L] \le W\,\mathbb{E}_P L$. Average over calibration and use $L \le B$. ■

Generated candidates are optimised against a predictor, so their law Q can differ from the calibration law P. The bound degrades by the factor W and is vacuous when $W\alpha \ge B$. A law Q that is chosen after the calibration outcomes are seen is not covered by this proof. Particles from one sequential Monte Carlo run share resampling ancestry and fail exchangeability, so split-conformal calibration and risk control draw at most one candidate or cloud summary per independent seed.

Exact count bounds, power and clustered designs. When a rate is estimated from a count of independent units, the exact binomial distribution gives a bound with no normal approximation. The bound is conservative because the count is discrete. Its units must be independent: replicate wells for one construct measure repeatability and do not supply independent constructs.

Proposition 5.8 (one-sided exact bound and the zero-failure rule). Let $K \sim \mathrm{Bin}(n, p)$ and $\alpha \in (0,1)$. Let $U(k)$ be the $(1-\alpha)$ quantile of $\mathrm{Beta}(k+1, n-k)$ for $k < n$, and $U(n) = 1$. Then $\mathbb{P}_p\{p \le U(K)\} \ge 1-\alpha$ for every p. For $K = 0$, $U(0) = 1 - \alpha^{1/n}$, and $U(0) \le p_0$ if and only if $n \ge \log\alpha/\log(1-p_0)$.

Proof. The identity $\mathbb{P}_p(K \le k) = \mathbb{P}(\mathrm{Beta}(k+1, n-k) > p)$ shows that $U(k)$ is the value of p at which $\mathbb{P}_p(K \le k) = \alpha$, and that $\mathbb{P}_p(K \le k)$ falls as p grows. For fixed p, let k_p be the largest k with $\mathbb{P}_p(K \le k) \le \alpha$. The bound fails only when $K \le k_p$, which has probability at most α. At $k = 0$ the endpoint solves $(1-U)^n = \alpha$. Both logarithms are negative, which gives the condition on n. ■

Numerical illustration: with 18 events in 200 independent units and $\alpha = 0.05$, the upper endpoint is 0.1305, so an observed fraction of 0.09 does not clear an upper-risk limit of 0.10. For zero events, the required n is 14, 29, 59, 149 and 299 for $p_0 = 0.20$, 0.10, 0.05, 0.02 and 0.01. A one-sided exact test of $p \ge 0.8$ with $n = 200$ and size at most 0.05 rejects for $K \le$

150 (size 0.0494), and its power reaches 0.8 only for alternatives $p \leq 0.7258$. A sample-size label alone therefore does not show power against a smaller drop.

Proposition 5.9 (design effect). Let m observations in one cluster have common variance σ^2 and common pairwise correlation ρ_c. Then $\mathrm{Var}(\bar{Y}) = (\sigma^2/m)\,[1 + (m-1)\rho_c]$. For C clusters, the effective sample size is Cm/D with $D = 1 + (m-1)\rho_c$.

Proof. $\mathrm{Var}(\bar{Y}) = m^{-2}\{m\sigma^2 + m(m-1)\rho_c\sigma^2\}$. ■

The number of clusters, and not the number of wells, sets the large-sample degrees of freedom. A balanced model with C clusters, m constructs per cluster and r technical repeats per construct has grand-mean variance $\sigma_c^2/C + \sigma_v^2/(Cm) + \sigma_e^2/(Cmr)$, because the cluster, construct and repeat effects are independent and their means have these variances. More repeats change only the last term, which is why they cannot replace more clusters.

Exploration weights and sequential evidence. A campaign that picks designs partly at random with known probabilities can weight each outcome by the inverse of its selection probability, and a running product of fair-bet factors gives a test that can be watched at any time.

Proposition 5.10 (predictable inverse-probability weights). Given the past $\mathcal{F}_{j-1}$, let u_j, p_j and a bounded loss ℓ_j be fixed on a finite pool, with $p_j > 0$ wherever $u_j > 0$. For $X_j \sim p_j$, $\mathbb{E}[(u_j/p_j)(X_j)\,\ell_j(X_j) \mid \mathcal{F}_{j-1}] = \sum_x u_j(x)\ell_j(x)$.

Proof. Expand the conditional expectation as a finite sum and cancel $p_j(x)$. ■

The identity needs selection to come before the observed outcome. A final predictor fitted to the sampled outcomes is not fixed in this argument. If $p_j = \epsilon u_j + (1-\epsilon)q_j$ then $u_j/p_j \leq 1/\epsilon$.

Theorem 5.11 (Ville's inequality). Let $(E_t)_{t\geq 0}$ be a nonnegative supermartingale with $\mathbb{E}E_0 \leq 1$. For every $\alpha \in (0,1)$, $\mathbb{P}(\sup_t E_t \geq 1/\alpha) \leq \alpha$.

Proof. For a horizon T let $\tau = \inf\{t \leq T : E_t \geq 1/\alpha\}$, with $\tau = T$ if there is no crossing. Optional sampling gives $\mathbb{E}E_\tau \leq \mathbb{E}E_0 \leq 1$. On the crossing event $E_\tau \geq 1/\alpha$, so $1 \geq \alpha^{-1}\mathbb{P}(\max_{t\leq T} E_t \geq 1/\alpha)$. Let $T \to \infty$ and use monotone convergence of the events. ■

A product $E_t = \prod_{s\leq t} e_s$ is a test supermartingale when each factor is nonnegative, measurable with respect to the past, and has conditional expectation at most one under every null law [7]. A stake chosen from past data is allowed. A stake tuned on the current outcome is not predictable and can destroy the property. Stopping at any data-dependent time is then valid. Repeated fixed-horizon tests are not: they inflate the crossing probability.

S1.6 Representation floor and predictive information

This section states what the two penalty terms of the Interface-JEPA design guarantee about the spread of a representation (Theorem 6.1) and why they act on raw vectors and not on length-normalised ones (Proposition 6.2). It states what a representation must keep to serve a downstream endpoint (Theorem 6.3) and why a small prediction loss does not show that it does (Example 6.4). The base model has no predictive layer.

Let $H \in \mathbb{R}^{B\times d}$ hold B centred raw representation vectors of width d, and let $\Sigma = H^\top H/(B-1)$ be the batch covariance. With a target spread $\tau > 0$ and a small constant $\delta \geq 0$, the two terms are

$$\mathcal{L}_{\mathrm{var}} = \frac{1}{d}\sum_{i=1}^{d}(\tau - \sqrt{\Sigma_{ii}+\delta})_+, \qquad \mathcal{L}_{\mathrm{cov}} = \frac{1}{d}\sum_{i\neq j}\Sigma_{ij}^2.$$

Theorem 6.1 (noncollapse floor). If $\mathcal{L}_{\mathrm{var}} \leq \eta$ and $\mathcal{L}_{\mathrm{cov}} \leq \kappa$, then

$$\lambda_{\min}(\Sigma) \;\geq\; (\tau - d\eta)_+^2 - \delta - \sqrt{(d-1)\kappa}.$$

The bound is attained: for $d \geq 2$, $s > 0$, $0 < a \leq s^2/(d-1)$, $\tau = s$ and $\delta = \eta = 0$, the matrix $\Sigma = s^2 I - a(\mathbf{1}\mathbf{1}^\top - I)$ has $\mathcal{L}_{\mathrm{var}} = 0$, $\mathcal{L}_{\mathrm{cov}} = (d-1)a^2 = \kappa$ and $\lambda_{\min} = s^2 - (d-1)a = s^2 - \sqrt{(d-1)\kappa}$.

Proof. Each hinge term is nonnegative and their mean is at most η, so each is at most $d\eta$. Hence $\sqrt{\Sigma_{ii}+\delta} \geq \tau - d\eta$ and $\Sigma_{ii} \geq (\tau - d\eta)_+^2 - \delta$.

Write $\Sigma = D + O$ with D diagonal and O symmetric with zero diagonal. Weyl's inequality gives $\lambda_{\min}(\Sigma) \geq \min_i D_{ii} - \| O \|_{\mathrm{op}}$. Let μ be an eigenvalue of O with $|\mu| = \| O \|_{\mathrm{op}}$. Since $\mathrm{tr} O = 0$, the other $d-1$ eigenvalues sum to $-\mu$, and Cauchy–Schwarz gives that their squares sum to at least $\mu^2/(d-1)$. So $\| O \|_{\mathrm{F}}^2 \geq \mu^2 d/(d-1)$. Also $\| O \|_{\mathrm{F}}^2 = \sum_{i\neq j}\Sigma_{ij}^2 = d\,\mathcal{L}_{\mathrm{cov}} \leq d\kappa$, which gives $\| O \|_{\mathrm{op}} \leq \sqrt{(d-1)\kappa}$.

For the example, the eigenvalue along $\mathbf{1}$ is $s^2 - (d-1)a$ and every orthogonal direction has eigenvalue $s^2 + a$. For $B \geq d+1$ the example is the covariance of a centred sample $H = \sqrt{B-1}\,T\Sigma^{1/2}$, where $T^\top T = I_d$ and $T^\top \mathbf{1} = 0$. The case $d = 1$ has no off-diagonal term. ■

A positive floor needs the joint condition $(\tau - d\eta)_+^2 > \delta + \sqrt{(d-1)\kappa}$, and loose separate limits on η and κ can fail it. For $d = 8$, $\tau = 1$ and $\delta = 0$ the lower bound is positive exactly when $\eta < 1/8$ and $\kappa < (1-8\eta)^4/7$. The rank of Σ is at most $B-1$, so $B \geq d+1$ is

necessary for a full-rank covariance. A gate that tests the median variance over dimensions does not enforce this bound, because the bound constrains the smallest diagonal entry.

Proposition 6.2 (normalised vectors cannot carry the floor). If every row of H has unit norm before centring, then $\mathrm{tr}\Sigma \le B/(B-1)$ and $\lambda_{\min}(\Sigma) \le B/[d(B-1)]$.

Proof. $\mathrm{tr}\Sigma = (B-1)^{-1}\sum_b \| h_b - \bar{h} \|^2 \le (B-1)^{-1}\sum_b \| h_b \|^2 = B/(B-1)$, and the smallest eigenvalue is at most the mean eigenvalue $\mathrm{tr}\Sigma/d$. ■

For $B = 4096$ and $d = 256$ the limit is 0.003907, so a floor of 0.1 is impossible on normalised vectors. The variance terms therefore act on raw vectors, and cosine comparisons may use normalised copies.

Theorem 6.3 (information retained for an endpoint). Let X, Y and the actions lie in standard Borel spaces, let $Z = h(X)$ be measurable, and let the loss $\ell(a, Y) \in [0, M]$ be measurable. Assume measurable approximate Bayes rules exist with arbitrarily small excess risk. With $R_X^\star$ and $R_Z^\star$ the optimal risks given X and given Z, and I the conditional mutual information in nats,

$$0 \;\le\; R_Z^\star - R_X^\star \;\le\; M\sqrt{2\, I(X; Y \mid Z)}.$$

Proof. Replacing $P(Y \mid x)$ by $P(Y \mid h(x))$ changes the expected loss of any fixed action by at most M times their total variation distance. Comparing two optimal actions uses this twice, so the excess risk is at most $2M\, \mathbb{E}\mathrm{TV}(P_{Y|X}, P_{Y|Z})$. Pinsker's inequality and Jensen's inequality bound this by $2M\sqrt{\mathbb{E}\mathrm{KL}(P_{Y|X} \mid P_{Y|Z})/2} = M\sqrt{2I(X; Y \mid Z)}$. The lower bound holds because every rule that uses Z also uses X. Infima are handled by approximate minimisers. ■

The theorem concerns optimal risks under a bounded loss. It does not bound the error of a trained predictor, and an unbounded loss such as a Huber loss on an unbounded range is outside it.

Example 6.4 (zero prediction loss, unbounded information loss). Let $X = (U, Y)$ with U independent of a finite-valued Y. Let $h(X) = U$, and let the prediction target and the predictor both equal U. The prediction loss is zero and U may have a full-rank covariance. Yet $I(X; Y \mid h(X)) = H(Y)$, which grows without bound as the alphabet of Y grows. A representation can predict its chosen target exactly and discard another endpoint entirely.

A small prediction loss therefore does not bound the conditional information. A population cross-entropy for a categorical endpoint is at least $H(Y \mid h(X))$, which bounds $I(X; Y \mid h(X))$

from above, but an empirical cross-entropy needs a generalisation argument for the selected predictor and data.

S1.7 Decoder pullback metric and common descent

A step in the latent space can change the decoded molecule by unequal amounts in different directions, so the guidance of the full system measures a step by its effect on decoded features. This section proves that the resulting metric is positive definite (Proposition 7.1) and that the preconditioned gradient is the steepest direction (Theorem 7.2). It bounds the error of the linear solve and of the linear model (Propositions 7.3 and 7.4), covers rotation (Proposition 7.5) and states what a common descent direction guarantees (Proposition 7.7, Theorem 7.8). **Supplementary Figure S1** and **Supplementary Figure S3** illustrate the metric and the common direction.

Let $h_\phi \colon \mathbb{R}^D \to \mathbb{R}^P$ collect differentiable dimensionless decoded features, with Jacobian J_h. Let $W_h \succcurlyeq 0$ hold fixed feature weights and $\tau_{\text{ridge}} > 0$ be a ridge. Put $A = W_h^{1/2} J_h$. The metric and its squared length are

$$G_\phi = J_h^\top W_h J_h + \tau_{\text{ridge}} I_D, \qquad \| v \|_{G_\phi}^2 = \| Av \|_2^2 + \tau_{\text{ridge}} \| v \|_2^2.$$

The first term is the linearised displacement of the weighted features. The ridge covers directions that the features do not see.

Proposition 7.1 (spectral bounds). For every v, $\tau_{\text{ridge}} \| v \|^2 \le v^\top G_\phi v \le (\tau_{\text{ridge}} + \| A \|_2^2) \| v \|^2$. Hence $G_\phi \succ 0$ and $\| G_\phi^{-1} \|_2 \le 1/\tau_{\text{ridge}}$.

Proof. The squared length gives the lower bound by dropping its first nonnegative term, and the operator-norm inequality gives the upper bound. The inverse norm is the reciprocal of the smallest eigenvalue. ■

A very small ridge amplifies gradient error in directions the features do not see, and a large ridge removes the preconditioning. The smallest and largest eigenvalue estimates are therefore recorded on validation states.

Theorem 7.2 (steepest direction). For $G \succ 0$ and a differentiable utility U, $\max_{\|d\|_G \le 1} \nabla U^\top d = \sqrt{\nabla U^\top G^{-1} \nabla U}$. When $\nabla U \neq 0$ the maximum is attained at $d^\star = G^{-1} \nabla U / \sqrt{\nabla U^\top G^{-1} \nabla U}$.

Proof. Put $z = G^{1/2}d$, so $\| z \|_2 \leq 1$ and $\nabla U^\top d = (G^{-1/2}\nabla U)^\top z$. Cauchy–Schwarz gives the maximum, attained when z is parallel to $G^{-1/2}\nabla U$. ■

The direction $G^{-1}\nabla U$ is a natural gradient of a statistical family only if G is the Fisher information of that family. The pullback metric measures declared feature displacement, and calling it Fisher would need a probabilistic derivation that is not made here.

Proposition 7.3 (inverse products and residuals). (a) $(A^\top A + \tau_{\text{ridge}} I)^{-1} g = \tau_{\text{ridge}}^{-1} g - \tau_{\text{ridge}}^{-1} A^\top (AA^\top + \tau_{\text{ridge}} I_P)^{-1} Ag$. (b) If $\hat{d}$ is any approximate solution of $G_\phi d = g$ with residual $r = g - G_\phi \hat{d}$, then $\| \hat{d} - G_\phi^{-1} g \|_2 \leq \| r \|_2 / \tau_{\text{ridge}}$.

Proof. (a) is the Woodbury identity. (b) $\hat{d} - G_\phi^{-1} g = -G_\phi^{-1} r$, and Proposition 7.1 bounds the norm. ■

The bound in (b) concerns the linear solve and not the decoded molecule, whose validity is checked after the step.

Proposition 7.4 (linearisation remainder and trust radius). If J_h is L_J-Lipschitz in operator norm on the segment from q to $q + \delta$, then

$$\| h_\phi(q+\delta) - h_\phi(q) - J_h(q)\delta \| \leq 1/2\, L_J \| \delta \|^2, \qquad \| W_h^{1/2}\{h_\phi(q+\delta) - h_\phi(q)\} \| \leq \| A\delta \| + 1/2 \| W_h^{1/2} \| L_J \| \delta \|^2.$$

If $\| \delta \|_{G_\phi} \leq \Delta$, then $\| \delta \|_2 \leq \Delta / \sqrt{\tau_{\text{ridge}}}$ and $\| A\delta \| \leq \Delta$, so the weighted displacement is at most $\Delta + \| W_h^{1/2} \| L_J \Delta^2 / (2\tau_{\text{ridge}})$.

Proof. Write $h(q+\delta) - h(q) = \int_0^1 J_h(q + t\delta)\delta \, \mathrm{d}t$, subtract $J_h(q)\delta$, use $\| J_h(q + t\delta) - J_h(q) \| \leq t L_J \| \delta \|$ and integrate. Multiply by $W_h^{1/2}$ and apply the triangle inequality. The trust statements follow from the squared length. ■

The quadratic term shows why a step that satisfies its linearised metric limit can leave the intended feature ball. The decoded displacement is therefore evaluated after each step, and the step is shortened when it is too large.

Proposition 7.5 (equivariance and padding). Let the state transform by an orthogonal representation ρ_g, and the features by an orthogonal representation σ_g with $h_\phi(\rho_g q, gc) = \sigma_g h_\phi(q, c)$. If $\sigma_g^\top W_h \sigma_g = W_h$, then $G_\phi(\rho_g q, gc) = \rho_g G_\phi(q, c) \rho_g^\top$, and $G_\phi^{-1}\nabla U$ is equivariant for every invariant differentiable U. A feature map must also factor through the active state,

$h_{\mathrm{pad}}(q_{\mathrm{pad}}) = h_e(S_e q_{\mathrm{pad}})$ with S_e the selector of active coordinates. A ridge added in padded storage would give padding an artificial geometry, so the metric is built in the packed state and only the resulting vector is scattered back.

Proof. Differentiating the feature relation gives $J_h(\rho_g q, gc) = \sigma_g J_h(q, c)\rho_g^{\top}$. Substituting in the definition of G_ϕ and using $\sigma_g^{\top} W_h \sigma_g = W_h$ and $\rho_g^{\top}\rho_g = I$ gives the first claim. Lemma 2.7 gives the second. ■

Proposition 7.6 (density conventions). If a model declares the Riemannian volume $\mathrm{dvol}_G = \sqrt{\det G_\phi}\,\mathrm{d}q$, a Lebesgue density p_L and a Riemannian density p_G satisfy $p_L = p_G\sqrt{\det G_\phi}$, so $\log p_G = \log p_L - 1/2\,\log\det G_\phi$. The Riemannian divergence is $\mathrm{div}_G v = (\det G)^{-1/2}\,\partial_i\big((\det G)^{1/2} v^i\big)$.

Proof. The two densities describe the same probability on the same set, so $p_L\,\mathrm{d}q = p_G\,\mathrm{dvol}_G$. ■

Using div_G in a likelihood formula while the base density is Lebesgue mixes two reference measures. The Methods use G_ϕ as a preconditioner and keep the Lebesgue density of the intrinsic chart.

A common direction is a step that improves all utilities to first order. It is the point of smallest length in the convex hull of the preconditioned gradients, and the origin in that hull means that no such step exists at this state.

For n_f scaled losses $\tilde{f}_i = (f_i - b_i)/r_i$ with offsets b_i and scales $r_i > 0$, let $g_i = \nabla\tilde{f}_i$, $a_i = G_\phi^{-1} g_i$ and $H_{ij} = g_i^{\top} G_\phi^{-1} g_j$. Let $\lambda^\star$ minimise $1/2\,\lambda^{\top} H\lambda$ over the simplex Δ_{n_f}, and put $w = \sum_i \lambda_i^\star\, a_i$.

Proposition 7.7 (projection inequality). For every i, $g_i^{\top} w = \langle a_i, w\rangle_{G_\phi} \geq \| w \|_{G_\phi}^2$, with equality when $\lambda_i^\star > 0$.

Proof. The conditions for the simplex problem are $H\lambda - \nu\mathbf{1} - \mu = 0$, $\mathbf{1}^{\top}\lambda = 1$, $\lambda, \mu \geq 0$ and $\lambda_i\mu_i = 0$. Row i gives $(H\lambda)_i = \nu + \mu_i \geq \nu$. Multiplying by $\lambda^{\top}$ and using complementarity gives $\lambda^{\top} H\lambda = \nu$. Now $(H\lambda)_i = \langle a_i, w\rangle_{G_\phi}$ and $\lambda^{\top} H\lambda = \| w \|_{G_\phi}^2$. Equality holds where $\mu_i = 0$. ■

Theorem 7.8 (one step lowers every loss). Let each $\tilde{f}_i$ have an L_i-Lipschitz gradient near q and let $w \neq 0$ satisfy Proposition 7.7. Then $\tilde{f}_i(q - hw) \leq \tilde{f}_i(q) - h \| w \|_{G_\phi}^2 + 1/2\, L_i h^2 \|$

$w \|_2^2$. For any $c \in (0,1)$, the common Armijo condition $\tilde{f}_i(q - hw) \le \tilde{f}_i(q) - ch \| w \|_{G_\phi}^2$ holds for every i whenever $0 < h \le \min_i 2(1-c) \| w \|_{G_\phi}^2 / (L_i \| w \|_2^2)$ and the step stays in the neighbourhood.

Proof. The descent lemma gives $\tilde{f}_i(q - hw) \le \tilde{f}_i(q) - h g_i^\top w + 1/2\, L_i h^2 \| w \|_2^2$. Use $g_i^\top w \ge \| w \|_{G_\phi}^2$ and rearrange for the fraction c. ■

Four limits and one related fact apply. If $w = 0$, a convex combination of the gradients vanishes, which is first-order Pareto stationarity and not global Pareto optimality. An approximate linear solve, an approximate simplex solution or a later projection requires the inequalities of Proposition 7.7 and Theorem 7.8 to be checked on the realised correction. The weights depend on the state, so the combined field need not be the gradient of one potential. With smooth constraints, a local weak Pareto point satisfies the Karush–Kuhn–Tucker condition $\sum_i \lambda_i \nabla f_i + \sum_j \mu_j \nabla c_j + \sum_k \nu_k \nabla h_k = 0$ with $\lambda \in \Delta$ and $\mu_j \ge 0$ under a constraint qualification. The condition is necessary and, without convexity, not sufficient. The related fact concerns fixed weights. A linear scalarisation with all weights positive returns a Pareto-optimal point, because a dominating point would lower the weighted sum. A nonconvex frontier has Pareto-optimal points that no weighting reaches (standard, by the supporting-hyperplane argument), and the min-norm direction avoids that limit.

S1.8 Numerical error budgets

This section bounds the error of each approximation behind a reported number: the integrator step, the stopping rule of a solver, the finite number of probes or particles, and the rounding of floating-point sums. It shows how the bounds add up for a log density and which refinement test measures each part. The base model relies only on the bound on floating-point sums (Proposition 8.6); the other results support the full system, because the base model integrates no flow and runs no particle method.

Coupled state and log density. Let $v(q,t)$ be the field and $d = \nabla \cdot v$ its divergence. The log density ℓ_t along a trajectory solves

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} q_t \\ \ell_t \end{bmatrix} = \begin{bmatrix} v(q_t,t) \\ -d(q_t,t) \end{bmatrix}, \qquad \ell_0 = \log p_0(q_0),$$

and ℓ_T equals the model log density only for the same trajectory, field and divergence. The production solver integrates both components in one system, or records the interpolation and

quadrature error that a separate pass adds. An adaptive solver measures its error in a weighted norm,

$$\| e \|_{\mathrm{tol}}^2 = \frac{1}{D+1}\Big[\sum_{j=1}^{D}\Big(\frac{e_{q,j}}{a_{q,j}+r_q|q_j|}\Big)^2 + \Big(\frac{e_\ell}{a_\ell + r_\ell|\ell|}\Big)^2\Big],$$

where a and r are absolute and relative tolerances. If ℓ shares one absolute tolerance with a state component, its share of the sum is negligible for large D. The solver profile therefore declares separate tolerances for state and log density.

Theorem 8.1 (discrete Grönwall bound). Let a one-step method Ψ_h on a grid $t_k = kh$ satisfy $\| \Psi_h(x,t) - \Psi_h(y,t) \| \le (1+Lh) \| x - y \|$, and let its local defect along the exact solution satisfy $\| q(t_{k+1}) - \Psi_h(q(t_k), t_k) \| \le Ch^{p+1}$. After $K = T/h$ steps,

$$\| q(T) - q_K \| \le \begin{cases} (C/L)(e^{LT}-1)\,h^p, & L > 0, \\ CT\,h^p, & L = 0. \end{cases}$$

Proof. Let $e_k = \| q(t_k) - q_k \|$. Adding and subtracting $\Psi_h(q(t_k), t_k)$ gives $e_{k+1} \le (1+Lh)e_k + Ch^{p+1}$. From $e_0 = 0$, $e_K \le Ch^{p+1} \sum_{j<K} (1+Lh)^j = Ch^p\{(1+Lh)^K - 1\}/L$, and $(1+Lh)^K \le e^{LT}$. For $L = 0$ the sum is K equal defects. ■

The bound is for a fixed-step method with a known defect constant C, and the embedded error estimate of an adaptive solver is not that constant. A convergence study halves the tolerance or the largest step and checks the endpoint, the integrated divergence and every statistic that a gate reads. A plateau can mean rounding or trace noise and does not by itself show convergence.

Proposition 8.2 (projections and events). If the exact solution stays in a closed convex set Q and Π_Q is the Euclidean projection, then $q_{k+1} = \Pi_Q\{\Psi_h(q_k, t_k)\}$ has no larger error than the unprojected step. If integration stops at a root of $g(q_t, t) = 0$ where $\partial_t g + \nabla g^\top v \neq 0$, the crossing is transversal and local interpolation can locate it.

Proof. The projection is nonexpansive, and $q(t_{k+1}) \in Q$ is its own projection, so $\| \Pi(x) - q(t_{k+1}) \| \le \| x - q(t_{k+1}) \|$. Transversality is the condition of the implicit function theorem for the crossing time. ■

Molecular validity sets, with discrete sequence rules and topology changes, are not convex, and their decoder projection can be discontinuous, so formal order does not transfer across such a projection. Projected generation is therefore a separate solver profile whose endpoint distribution is validated directly. A gate on a discontinuous decoded predicate is an admission test after the step and not a solver event.

Stable sums and entropic transport. The scaling equations of Section S1.4 and the particle weights are sums of exponentials, and the next statements control their evaluation.

Lemma 8.3 (log-sum-exp shift). For finite $z_1, \dots, z_m$ and any c, $\log \sum_i e^{z_i} = c + \log \sum_i e^{z_i - c}$. With $c = \max_i z_i$ every shifted term $e^{z_i - c}$ lies in $(0,1]$ and the sum is at least one, so the sum can neither overflow nor underflow to zero. ■

The scaling equations of Theorem 4.3 are evaluated in this form, for example $a_i = -\tau_{\mathrm{OT}} \varepsilon \log \sum_j \mu_{E,j} \exp\{(b_j - C_{ij})/\varepsilon\}$, with each log-sum-exp shifted by its maximum. When ε is small, the subtraction of nearly equal primal and dual values can still lose accuracy. The solver therefore reports the absolute primal-dual gap, a scale-normalised gap, the marginal residuals and the plan change, and no one of them replaces the other three.

Proposition 8.4 (objective gap and plan distance). Let $\Gamma^\star$ minimise a differentiable μ-strongly convex F on a convex set and let $\hat{\Gamma}$ be feasible. Then $F(\hat{\Gamma}) - F(\Gamma^\star) \geq \mu/2 \parallel \hat{\Gamma} - \Gamma^\star \parallel_{\mathrm{F}}^2$. For the contact objective of Section S1.4, on the set $\{0 < \Gamma_{ij} \leq \Gamma_{\max}\}$, one valid constant is $\mu = \varepsilon_{\mathrm{OT}} / \Gamma_{\max}$.

Proof. Optimality of $\Gamma^\star$ gives $\langle \nabla F(\Gamma^\star), \hat{\Gamma} - \Gamma^\star \rangle \geq 0$, and strong convexity adds $\mu/2 \parallel \hat{\Gamma} - \Gamma^\star \parallel^2$. The Hessian of $\varepsilon_{\mathrm{OT}} \mathrm{KL}(\Gamma \mid \mu_R \otimes \mu_E)$ is $\mathrm{diag}(\varepsilon_{\mathrm{OT}} / \Gamma_{ij}) \succcurlyeq (\varepsilon_{\mathrm{OT}} / \Gamma_{\max}) I$, and the other terms are convex. ■

The constant depends on an upper bound for the plan entries, and iterates near zero have large curvature. A generic tolerance on plan change cannot be turned into a guarantee on the objective, or the reverse, without such a bound.

Particle weights. Particle algorithms add log weights over steps. Set $\ell_k^i = \log \widetilde{w}_{k-1}^i + \log \omega_k$, where ω_k is the incremental weight of Section S1.4, $\widetilde{w}_{k-1}^i = 1/N_{\mathrm{p}}$ after resampling and $\widetilde{w}_{k-1}^i = \bar{w}_{k-1}^i$ when resampling is skipped. Then $\log \bar{\ell}_k = \mathrm{LSE}(\ell_k^{1:N_{\mathrm{p}}})$ and $\bar{w}_k^i = \exp\{\ell_k^i - \log \bar{\ell}_k\}$. Subtracting the maximum prevents overflow, but values far below the maximum underflow to zero. The loss is harmless only if the discarded mass is below a declared tolerance.

Proposition 8.5 (mass lost by truncation). Let $B \geq 0$, let $m = \max_i \ell_i$ and set to zero every term with $\ell_i - m < -B$. If r of the N terms are removed, their normalised total mass is at most $r e^{-B} / (1 + r e^{-B}) \leq r e^{-B}$.

Proof. Each removed exponential is below e^{-B} times the largest, and the retained sum contains at least the largest term. ■

As a numerical illustration, for $r = 100$ and $B = 40$ the discarded mass is at most 4.2×10^{-16}. A weight that is not finite is a typed numerical failure, and replacing it by zero would hide a support or density-ratio error. A complete update checks finiteness before normalisation, positive total mass after it, and the ancestor indices after resampling. If a check fails, the estimate of the normalising constant is withheld.

Floating-point sums. The classical forward bound for summation gives the accuracy of long reductions.

Proposition 8.6 (summation bound). Let u be the unit roundoff, $\gamma_k = ku/(1-ku)$ and $ku < 1$. Sequential floating-point summation of $x_1, \dots, x_n$ satisfies $|\mathrm{fl}(\sum_i x_i) - \sum_i x_i| \leq \gamma_{n-1} \sum_i |x_i|$. Pairwise summation has the same form with $\gamma_{\lceil \log_2 n \rceil}$.

This is the classical forward bound and is cited. Numerical illustration: for $n = 10^6$ terms in FP32 ($u = 2^{-24}$) the sequential bound is $\gamma_{n-1} = 0.0634$ and the pairwise bound is $\gamma_{20} = 1.2 \times 10^{-6}$. In BF16, $u = 2^{-8}$, and the condition $ku < 1$ already fails for $n \geq 257$. Accumulation is therefore kept in FP32, and the reduction order is fixed where bitwise agreement is required. The base model sums gradients across ranks in FP32 for this reason: the bound is informative there and is not informative in BF16.

Bitwise reproducibility needs a fixed reduction tree, deterministic kernels and fixed random streams. Statistical reproducibility needs independent seeds with stable distributions, and its tolerance is declared before evaluation.

Assembling a budget. For a reported log density, a conservative decomposition by the triangle inequality is

$$|\hat{\ell} - \ell| \leq E_{\mathrm{state}} + E_{\mathrm{quad}} + E_{\mathrm{trace}} + E_{\mathrm{round}} + E_{\mathrm{chart}},$$

for the trajectory error propagated into the divergence, the time quadrature, the probe noise, the floating-point reductions and the chart constants. Proposition 3.9 bounds the first two together from the solver defect ε_f and the divergence error ε_d. Proposition 3.8 gives the variance behind the third. For a guided endpoint statistic, particle error and selection error are added as separate terms and are not folded into the probe term. Each source has its own refinement test.

The ODE state is tested with a tighter tolerance or a smaller largest step, comparing the endpoint state in the intrinsic and decoded metrics. The divergence quadrature is tested with denser or higher-order integration on a fixed trajectory, comparing the integrated log-density correction. The trace estimator is tested with more independent probes and, for a small state, an exact trace,

comparing the probe mean, the variance and the exact error. The contact solve is tested with a tighter dual gap and more iterations, comparing the objective, the marginals and the thresholded contacts. The particle approximation is tested with more particles and independent complete repeats, comparing the normaliser, the endpoint statistic, the effective sample size and the ancestry. Floating-point error is tested with wider accumulation and a fixed reduction tree, comparing the residual and the final decision and not one intermediate tensor.

A tolerance is an instruction to an algorithm and is not a measurement of error. Solver tolerances, Sinkhorn stopping thresholds and linear residual targets become evidence only when a refinement study compares the scientific quantities that result. A report therefore gives the configured tolerance and the realised change under refinement.

S1.9 Sensitivity and identifiability

This section asks which parameters of a model with a finite parameter vector ϑ and declared observables $\mathcal{O}(\vartheta) = \mathbb{E}_\vartheta T(X)$ can be told apart from data, and how far an output moves when an input moves. The rank of the Jacobian of $\mathcal{O}$ gives a sufficient identifiability test (Theorem 9.1), and the same matrix gives the covariance of a linearised inverse (Proposition 9.2). A selection step is discontinuous and has an exact stability radius (Proposition 9.3). Bounded and Lipschitz observables are insensitive to small changes of a law (Proposition 9.4).

Definition. A parameter ϑ_0 is *locally identifiable* through $\mathcal{O}$ if some neighbourhood U of ϑ_0 exists in which $\mathcal{O}(\vartheta) = \mathcal{O}(\vartheta_0)$ implies $\vartheta = \vartheta_0$. Structural identifiability concerns the map $\mathcal{O}$. Practical identifiability also concerns noise, design and sample size.

Theorem 9.1 (rank condition). If $\mathcal{O}\colon \mathbb{R}^p \to \mathbb{R}^r$ is continuously differentiable near ϑ_0, $r \geq p$, and $J_\mathcal{O}(\vartheta_0)$ has column rank p, then ϑ_0 is locally identifiable through $\mathcal{O}$.

Proof. Full column rank means that some p rows of $J_\mathcal{O}(\vartheta_0)$ form a nonsingular square matrix. Let $\tilde{\mathcal{O}}$ keep those components. The inverse function theorem makes $\tilde{\mathcal{O}}$ one-to-one near ϑ_0. Equal $\mathcal{O}$ implies equal $\tilde{\mathcal{O}}$, hence equal parameters. ■

The rank condition is sufficient and not necessary: $\mathcal{O}(\vartheta) = \vartheta^3$ is injective near zero with a zero derivative there. A small singular value of $J_\mathcal{O}$ says that some parameter combination changes the observables only weakly to first order. It does not show exact nonidentifiability.

Proposition 9.2 (covariance of a linearised inverse). Let $\hat{\mathcal{O}} = \mathcal{O}(\vartheta_0) + J_\mathcal{O}(\vartheta - \vartheta_0) + \epsilon$ with $\mathbb{E}\epsilon = 0$, $\mathrm{Cov}(\epsilon) = \Sigma_\mathcal{O}$ and $J_\mathcal{O}$ of full column rank. The generalised least-squares estimator has

covariance $(J_{\mathcal{O}}^{\top}\Sigma_{\mathcal{O}}^{-1}J_{\mathcal{O}})^{-1}$. If the smallest eigenvalue of $\mathcal{I}_{\mathcal{O}} = J_{\mathcal{O}}^{\top}\Sigma_{\mathcal{O}}^{-1}J_{\mathcal{O}}$ tends to zero, the largest variance diverges.

Proof. The estimator is $(J^{\top}\Sigma^{-1}J)^{-1}J^{\top}\Sigma^{-1}\hat{\mathcal{O}}$ plus a constant, and its covariance follows from $\mathrm{Cov}(\epsilon) = \Sigma_{\mathcal{O}}$. The largest eigenvalue of the inverse is the reciprocal of the smallest eigenvalue of $\mathcal{I}_{\mathcal{O}}$. ■

The matrix $\mathcal{I}_{\mathcal{O}}$ is a design information matrix for the chosen observables. It equals the Fisher information only under a likelihood whose score gives the same expression. A new observable adds information only along a direction that the others constrain weakly, so designs are compared by the eigenvectors of $\mathcal{I}_{\mathcal{O}}$ and not by its trace.

Proposition 9.3 (stability of a top-N set). Let sorted scores satisfy $S_{(1)} \geq \cdots \geq S_{(M)}$ and let the margin be $\Delta_{\mathrm{sel}} = S_{(N)} - S_{(N+1)} > 0$. If every perturbation obeys $\| \delta S \|_{\infty} < \Delta_{\mathrm{sel}}/2$, the selected set is unchanged.

Proof. For a selected i and an unselected j, $S_i - S_j \geq \Delta_{\mathrm{sel}}$. After the perturbation $(S_i + \delta S_i) - (S_j + \delta S_j) \geq \Delta_{\mathrm{sel}} - 2 \| \delta S \|_{\infty} > 0$. ■

An admission indicator $A = \mathbf{1}\{S \geq \tau\}$ breaks the smooth delta method. If S has a density f_S continuous at τ, then $\mathrm{d}\,\mathbb{P}(S \geq \tau)/\mathrm{d}\tau = -f_S(\tau)$, so a high density at the threshold makes the yield sensitive even when the score is accurate. When the margin Δ_{sel} is small, uncertainty is propagated by rerunning the whole selector on bootstrap or posterior score draws.

Proposition 9.4 (bounded and Lipschitz observables). If $\mathrm{TV}(P, Q) \leq \delta$ and $a \leq T \leq b$, then $|\mathbb{E}_P T - \mathbb{E}_Q T| \leq (b - a)\delta$. If T is L_T-Lipschitz under the ground metric of the Wasserstein distance W_1, then $|\mathbb{E}_P T - \mathbb{E}_Q T| \leq L_T W_1(P, Q)$.

Proof. Scale T to $f = (T - a)/(b - a) \in [0,1]$. The variational definition of total variation gives $|\mathbb{E}_P f - \mathbb{E}_Q f| \leq \delta$. The second bound is the Kantorovich–Rubinstein duality for Lipschitz functions, used as stated. ■

S1.10 Standing of the results

Every result in this supplement is proved in full for the stated conditions, except Theorem 4.14, which is given as a sketch, and the classical theorems that are used as stated and marked cited. These are the Markov property of the reversed process (Theorem 3.6), the Wasserstein duality (Section S1.5), the Woodbury identity and the Karush–Kuhn–Tucker condition (Section S1.7),

the summation bound (Proposition 8.6) and the Kantorovich–Rubinstein duality (Proposition 9.4).

Theorem 4.7 establishes unbiased normaliser estimation with carried weights. Theorem 4.10 gives the exact-twist result. Example 4.8 and Propositions 4.12 and 4.13 distinguish unbiasedness from convergence and describe the limitations of systematic resampling. The kernel, potential and horizon are written K_j, Ψ_j and M throughout.

S2 Data tables

The public datasets considered for the full system are listed below with the licence class of each, together with the cut-off dates of published antibody-design systems, the candidate evaluation antigens and the rule for recording sources. The base model trains on protein structures (PDB entries, AlphaFold DB models and synthetic dimers). The antibody-specific sources belong to the full system. The Data section of the main text describes how such data are curated.

Dataset roster. Supplementary Table S3 gives the content and size of each source, the class of its reported licence and its role. Counts were read from release pages and database records on 5 and 6 October 2026, or are the figures that a cited paper gives for its own dataset. Terms attach to the object, so an article under CC BY does not make its dataset CC BY. Pretrain sets are unsupervised corpora. Train sets supply supervised tasks. Calibrate sets fix a threshold. Benchmarks are held out from training. Panels supply off-target or self background. References are consulted lists or annotation tables.

Dataset	Content and count	Licence class (terms reported)	Role
PDB [8]	Dated mmCIF snapshot, first biological assembly per entry	Open (CC0 1.0)	pretrain, reference
AlphaFold DB v6, 2025-09-15 [9]	241,070,489 structures; human 23,586; mouse 21,452; rat 22,152	Open (CC BY 4.0)	pretrain, panel
AlphaFold DB, Foldseek-clustered subset; TED domains [10]	About 3 M representatives; about 550 k after the pLDDT, length and coil filters; about 350 k after the β-sheet filter	Open (CC BY 4.0)	pretrain
Teddymer synthetic dimers [11]	3,556,223 clusters; 510,454 representatives after the interface filters	Non-commercial (CC BY-NC 4.0)	pretrain
PDB multimers, filtered [11]	45,856 structures	Open (CC0 1.0)	pretrain

Dataset	Content and count	Licence class (terms reported)	Role
SAbDab, 2026-09-30 [12, 13]	22,548 structures from 11,667 PDB entries; 6,927 antibodies; 20,111 with antigen. ML set v0.2.0: 15,810 structures, 5,424 antibodies	Open (CC BY 4.0)	train
SNAC-DB [14]	15,523 curated complexes; benchmark of 113 antibody-antigen and 101 VHH-antigen complexes	Reserved (training reserved by the authors)	benchmark
AbDb [15]; AACDB	AbDb: about 10 to 11 k redundancy-free Fv structures; 866 antibody-antigen entries, 825 complexes at 4.0 Å. AACDB v1.0: 7,498 complexes	Restricted (academic or author terms)	pretrain, train
Chimera-Bench [16]	20,509 SAbDab entries; 9,171 after quality filters; 2,981 after deduplication at 95 percent identity and 80 percent coverage; 2,922 validated complexes	No licence found	benchmark
FoldBench [17]	1,522 assemblies in nine tasks, among them 172 antibody-antigen pairs	No licence found (article CC BY-NC-ND 4.0)	benchmark
Smorodina panel [18]	106 cognate VHH-antigen complexes from 91 PDB entries	Preprint CC BY 4.0; terms of the structures not stated	benchmark, panel
Hitawala-Gray benchmark [19]	118 antibody-antigen and 70 nanobody-antigen complexes; boundary 2021-09-30; resolution below 2.8 Å; 95 percent identity filter	Open (CC BY 4.0 release label)	benchmark
OAS, unpaired [20]	2,428,152,691 sequences from 91 studies, 1.115 TB	Open (CC BY 4.0)	pretrain
OAS, paired [20]	3,087,576 pairs from 21 studies, 3.54 GB	Open (CC BY 4.0; per-study terms differ)	pretrain
ABodyBuilder2 OAS models; IgFold OAS models	About 1.5 M paired Fv models, 91.8 GB; 104,994 clusters plus 1,340,180 human models	Open (CC BY 4.0); Restricted (academic licence)	pretrain
CoV-AbDab, 2024-02-08	12,916 entries	Open (CC BY 4.0)	reference
Thera-SAbDab, 2025-03-18 [21]; PLAbDab, 2024-08-28 [22]; PLAbDab-nano	1,133 entries; 176,894 rows with 78,868 unique pairs; 4,913 sequences	No licence found	reference
INDI2; INDI1; PairedAbNGS	About 30 M NGS VHH with 2,768 nanobody-antigen complexes; 11,228,600 NGS VHH; 7,200,634 sequences (4,280,331 unique)	Non-commercial (INDI2, PairedAbNGS); no licence found (INDI1)	benchmark
iReceptor; NanoLAS; cAb-Rep	5.9 B annotated sequences; derivative collections	Restricted (commercial users pay); no licence found	not used
IMGT/GENE-DB v3.1.43	12,912 genes, 18,271 alleles	Open (CC BY 4.0)	reference

Dataset	Content and count	Licence class (terms reported)	Role
IgBert, p-IgGen and AbNatiV2 corpora	2.04 B unpaired plus 1.99 M paired; 1,800,545 paired; 20.2 M VHH	Open (CC BY 4.0, IgBert); not stated (p-IgGen); Non-commercial (CC BY-NC-SA 4.0, AbNatiV2)	pretrain, benchmark
UniProt 2026_03	Swiss-Prot 575,748; TrEMBL 149,430,635; UniRef100, 90, 50: 221 M, 121 M, 38.8 M	Open (CC BY 4.0)	reference
Reference proteomes; Human Protein Atlas v25.1	Human 147,520 proteins in 20,652 genes; cyno 23,292 genes; mouse 21,860; rat 21,471; 20,162 genes	Open (CC BY 4.0)	panel
SURFY; CSPA; OpenCell	2,886; 1,492 human; 1,310 proteins	No-derivatives or share-alike (SURFY CC BY-NC-ND 4.0; OpenCell images CC BY-SA); CSPA not stated	panel
MSA databases	ColabFold uniref30_2302 about 103 GB, envdb_202108 about 118 GB; BFD about 292 GB; MGnify 2026_07 over 5.7 B	Open (CC BY 4.0; CC0)	reference
GlyConnect 2.6.0; GlyGen 2.11.1	GlyGen: 27,052 human glyco-site rows	Open (CC BY 4.0), except UniCarbKB CC BY-NC-ND 3.0 and OMA CC BY-SA 2.5 inside GlyGen	reference
IEDB, 2026-09-28	B-cell assays 1,692,919; T-cell 578,203; MHC ligand 6,258,853; class II rows 1,975,591; 5,888 BCR groups, 3,639 with PDB ids	Open (CC BY 4.0)	reference, train
HGNC; Ensembl Compara	Gene groups; paralogs	Open (CC0; unrestricted)	panel
SKEMPI 2.0, 2018-06-06 [23]	7,085 rows from 345 PDB entries; antibody-antigen 1,211 rows from 55 entries	Open (CC BY 4.0)	train, calibrate
AB-Bind [24]; AbBiBench [25]	1,101 mutants in 32 complexes; 184,500 rows in 14 sets for 9 antigens	No licence found (AB-Bind); Open (CC BY 4.0, AbBiBench)	benchmark
SAAINT-DB v20260827; PPB-Affinity	11,695 PDB entries, 1,331 with affinity, 2,589 numeric KD rows; 12,062 rows, of which antibody-antigen 1,407	Open (MIT, CC BY 4.0); PPB-Affinity CC BY 4.0 with 3,600 rows taken from PDBbind	train
PDBbind v2025	35,924 entries	Restricted (paid; bars redistribution)	not used
Phillips 2021; Engelhart 2022; Makowski 2022; Shanehsazzadeh 2023	CR9114 65,091 variants and CR6261 1,812; 71,384 scFv (log10 nM); 4,000 labelled sequences; SPR KD of trastuzumab designs	Open (CC BY; BSD)	train
Mason 2021, Koenig 2017, Warszawski 2019, Adams 2016, Greaney and Starr escape maps	Enrichment or binary labels	Mixed; some none found	benchmark

Dataset	Content and count	Licence class (terms reported)	Role
Absolut! [26]	159 antigens, 6.9 M CDR-H3, 1.1 B structures, 72.3 GB	Non-commercial (data CC BY-NC 4.0)	benchmark
ANDD, 2026; Graphinity ddG	48,683 sequences, 9,557 affinities (2,271 predicted); 942,723 FoldX plus 20,829 Flex labels (computed)	Open (CC BY 4.0)	train
AbAgym; ASD v2.0; AbRank	323,752; 1,177,819; 381,713 rows	Non-commercial	benchmark
TAP and Jain 2017 [27, 28]	Jain: 137 antibodies, 12 assays. TAP: five metrics over 242 clinical-stage therapeutics	No licence found	calibrate
GDPa1 (Ginkgo, v1.2)	246 IgGs, 10 assays; GDPa3 80; GDPa5 160 VHH-Fc	Restricted (custom, gated)	train
Harvey 2022 PSR [29]; Boughter 2020	65,147 low and 69,155 high first-round VHH; 1,053 Fv (529 polyreactive)	Open (CC BY 4.0; CC BY)	train
NanoMelt; NbThermo; TEMPRO	640 VHH Tm; 564; 567	Non-commercial (NanoMelt, CC BY-NC-SA 4.0); papers CC BY, repositories none	benchmark, train
Tsuboyama 2023	About 776,000 stabilities	Open (CC BY 4.0)	train
NetMHCIIpan 4.3k; HLA Ligand Atlas	Over 650,000 points; 142,625 class II peptides	Restricted (internal research only); Open (CC BY 4.0)	benchmark; train
OASis DB [30]	139,000,294 9-mers, 231 subjects	Open (CC BY 4.0; code MIT)	train
Sapiens; AbLang2; AbNatiV	Released language models	Open (MIT; BSD-3); Non-commercial (AbNatiV)	reference

Supplementary Table S3. Dataset roster.

Counts are numbers of entries, structures, sequences, rows or proteins as stated in the content column; storage sizes are in GB or TB. The licence class is the one reported by the source.

Licence classes. The design admits CC0, CC BY 4.0, MIT, Apache 2.0 and public-domain data for weights that may be released. Non-commercial data are kept out of the lineage of released weights. The scores of a non-commercial scorer used as labels or as steering may fall outside the licence, and a head distilled from them does not. No-derivatives and share-alike sources are excluded. Sources with no licence found are public and keep all rights reserved, so they serve as references or for evaluation until written terms exist. Sources with custom, internal-research, paid or academic terms are excluded. A mixed collection is filtered by component. The licence labels are those the sources report and are not legal advice. The Omniverso code is under AGPL-3.0, which grants nothing on a weight file or on a dataset.

Source records. For each dataset, record its release, retrieval date, licence terms, inclusion criteria, deduplication rule and retained counts by split.

Candidate evaluation antigens. Candidate evaluation antigens are selected using the deposition-date and sequence-similarity criteria below. The resulting set distinguishes unseen antibody interfaces from antigens with homologues in a disclosed training set. A search of the PDB in October 2026 found 25 antigens that meet all of these conditions.

- The entry is an antibody or nanobody complex with a soluble protein antigen, at a resolution of at most 3.5 Å.
- It was deposited from 2025-01-07 to 2026-07-17, which is after the cut-offs of Chai-2, RFantibody, BoltzGen and Proteina-Complexa in **Supplementary Table S4** and after the temporal cut-off T^* (main text, Materials and Methods, Data).
- No earlier antibody complex exists on the same UniProt entry.
- No chain has 40 percent identity to a chain of an immunoglobulin entry deposited earlier.

The grade follows the training list of Proteina-Complexa. Grade A means that no entry of the list has 40 percent identity to the antigen. Grade B means at least one entry does. Grade B-high marks the grade B antigens with two or more entries at 70 percent identity. The pair after a grade B antigen gives its entries at 40 and at 70 percent identity.

- **Grade A (10).** CD30 (13HA, Fab); oncostatin M (9VWO, VHH); Rel of *C. tepidum* (9S6K, VHH); measles H (9Q1Z, Fab); B7-H3 (9LY5, Fab); GITR (9P8X, Fab); Langya G (9LEO, Fab); canine IL-31 (9UK5, Fab); Kasokero L endonuclease (9UZB, Fab); *P. vivax* CSS-PTRAMP (9NSD, VHH).
- **Grade B (9).** IL-33 (9X0J, two Fabs; 1/1); AZGP1 (13BR, Fab; 1/1); CCHFV nucleoprotein (9ZMQ, Fab; 1/0); CbpE (12QJ, Fab; 2/0); mouse IZUMO1 (9MYL, Fab; 3/0); leptin (9PUO, Fab; 1/1); macaque VWF (9QRV, Fab; 2/1); Ara h 9 (9NCD, Fab; 4/0); cathepsin D1 of *S. mansoni* (9RXI, VHH; 8/0).
- **Grade B-high (6).** CD44 HABD (32NZ, scFv; 2/2); SARS-CoV-2 nsp14 (9TH6, VHH; 6/6); CEACAM1 (23VF, Fab; 17/16); HCMV assemblin (9Y7M, Fab; 9/9); mouse MPO (21RN, two Fabs; 25/25); heparanase (9S8W, Fab; 16/16).

Homologues at 40 to 70 percent identity exist for the grade B antigens, so "held out" means an unseen antibody interface and not an unseen antigen.

Cut-off dates of published systems. A published result is comparable with another only if the training and test dates are known. **Supplementary Table S4** gives the targets and dates reported for published systems. Every comparison with these systems uses published numbers only and states that the panels are not matched.

System	Benchmark targets	Cut-off or boundary
Chai-2 [31]	52 targets	PDB date cut-off 2021-09-30 (exclusive); targets with a SAbDab antigen at 70 percent identity or more over 80 percent of the query were excluded
RFantibody [32]	HIV Env 2NY7; RBD 6M0J; RSV-F 7LVW and 7LUC; HA 5VLI; TcdB 6C0B and 7ML7; IL-7Rα 3DI3	SAbDab release to 2023-01-13
Germinal [33]	PD-L1, IL3, IL20, BHRF1	Not reported
BoltzGen [34]	3QKG, 1G13, 1JQD, 2PNY, 7AAH, 3APU, 2A1X, 3CH4, 1NB0, and PD-L1, TNF, IL-7Rα, PDGFRβ, insulin receptor	Training cut-off 2023-06-01
Proteina-Complexa [11]	19 protein targets; no antibody task	Last training deposit 2024-08-02, from the released training list
mBER [35]	145 screened of 436 targets; list unpublished	Not reported
SNAC-DB benchmark [14]	113 antibody-antigen and 101 VHH-antigen complexes	PDB deposits up to 2025-03-31
Hitawala-Gray benchmark [19]	118 antibody-antigen and 70 nanobody-antigen complexes	Boundary 2021-09-30

Supplementary Table S4. Targets and cut-off dates of published systems, as the cited sources report them. Dates are calendar dates. The temporal cut-off of Omniverso is $T^* = 2025\text{-}01\text{-}01$.

The date T^* is later than the cut-offs of Chai-2, RFantibody, BoltzGen and Proteina-Complexa and earlier than the end of the SNAC-DB window. It therefore controls overlap with the disclosed training sets of those four systems, and supports no other claim.

S3 Evaluation protocol

The evaluation protocol of the full system is a set of gates. A gate is a pass-or-fail test with a fixed population, statistic, threshold and sample size. The registry holds 53 gates. All limits below are design values that are fixed before any evaluation. A change made after an outcome is read counts as exploratory analysis and cannot clear a confirmatory gate. The base model is evaluated only by its training diagnostics (reconstruction error, residue cross-entropy and flow loss), which describe the course of training and do not measure model quality. Quality is judged by the evaluation protocol: the gates of this section and the evaluation design of the manuscript (Materials and Methods, Evaluation design).

Supplementary Figure S6 draws the fixed-horizon record (panel a), one decision taken from it (b) and a sequential trace with the rule for production gates (c and d). The text follows the order of the protocol: the record, the registered gates, the sample-size rules and the validation levels.

The fixed-horizon record. A gate is a promise written before the data exist. It says what is counted, what is measured, how uncertainty is handled and which result counts as failure. Writing it first stops a favourable result from choosing its own test. A consumer is one use of evidence. A chirality rate, an antigen-shuffle contrast, a panel-level false-discovery proportion and a co-folding score are four consumers with four records. The record of consumer k has eight fields:

$$\mathcal{E}_k = (\mathcal{P}_k, \mathcal{U}_k, n_k, T_k, I_k, \alpha_k, \tau_k, \mathcal{M}_{\mathrm{mult},k}).$$

- $\mathcal{P}_k$, the population: the objects tested and the rule that samples them.
- $\mathcal{U}_k$, the independent unit: what counts as one observation.
- n_k, the terminal sample size: the number of units, fixed in advance.
- T_k, the statistic: the number computed once, at n_k.
- I_k, the uncertainty procedure: the fixed-horizon interval or test.
- α_k, the error level: the declared error probability, between 0 and 1.
- τ_k, the threshold: the limit, and the direction in which failure lies.
- $\mathcal{M}_{\mathrm{mult},k}$, the multiplicity rule: the correction across the arms of one gate.

The index k matters because population, unit and loss differ between consumers. The independent unit is what is resampled. Residues of one domain, or designs against one antigen cluster, are not independent, so they enter through a domain-level or cluster-level unit. The statistic is computed once, at the committed n_k, with no early look and no optional stopping. The configuration accepts a positive integer for the evaluation horizon and the panel size and rejects the value `auto`.

A worked example shows why the bound decides and the point estimate does not. Suppose a gate commits 200 independent domains and a ceiling of 0.10 on the rate of domains with at least one registered chirality failure. Let K be the number of failing domains. If every domain fails independently with the same probability p, then K is binomial with $n = 200$. The one-sided upper limit at error level α is a quantile of a beta distribution:

$$I_\beta(k, n; \alpha) = \mathrm{Beta}^{-1}(1 - \alpha;\ k + 1,\ n - k)\ \ (k < n), \qquad I_\beta(n, n; \alpha) = 1.$$

This is the exact Clopper–Pearson limit [36]. For every p it lies above p with probability at least $1-\alpha$. With 18 failures the point estimate is $18/200 = 0.09$, and the limit at $\alpha = 0.05$ is $\mathrm{Beta}^{-1}(0.95;\ 19,\ 182) \approx 0.131$. The limit is above the ceiling, so the gate fails although the point estimate is below it. Twelve failures give a limit of 0.0954 and thirteen give 0.1014. At most twelve failures in 200 therefore clear the ceiling.

Some gates weight their units. For predetermined weights $w_i \geq 0$ with $\sum_i w_i > 0$ and losses $\ell_i \in [0,1]$, the weighted risk and the effective sample size are

$$\hat{R}_w = \frac{\sum_i w_i\, \ell_i}{\sum_i w_i}, \qquad n_{\mathrm{eff}} = \frac{(\sum_i w_i)^2}{\sum_i w_i^2}.$$

Weights 2, 1 and 1 on losses 0, 1 and 1 give $\hat{R}_w = 0.5$ and $n_{\mathrm{eff}} = 16/6 \approx 2.67$. The effective size measures how concentrated the weights are. An interval may use it in place of the sample size only when the theorem behind that interval allows the substitution.

Three populations keep their roles apart. Historical data fit the model and serve exploration. Held-out calibration objects set thresholds, calibration maps and interval procedures. Generated outputs, made after the generator, the guide, the selector and the compute policy are frozen, are the only population on which a claim about the generator is read.

Sequential evidence is a separate interface (Supplementary Figure S6c and d). It needs a declared family of null distributions, a filtration $\mathcal{F}_t$ (everything observed up to step t) and a stake chosen from $\mathcal{F}_{t-1}$ before the next observation. Each factor $e_t \geq 0$ must satisfy $\mathbb{E}_P[e_t \mid \mathcal{F}_{t-1}] \leq 1$ for every P in the null family. The running product $E_t = \prod_{j \leq t} e_j$, with $E_0 = 1$, is then a nonnegative test supermartingale. Under the null, the chance that it ever reaches $1/\alpha$ is at most α [7].

Factors 1.2, 0.8 and 1.1 give the capital path 1.2, 0.96 and 1.056. The arithmetic is a trace, and the validity comes from the conditional-expectation condition. A family of valid terminal e-values may enter the e-BH procedure under that procedure's conditions [37]. The registered gates read fixed-horizon statistics only. F-AV and F-J7 test the sequential code on simulated streams and are diagnostics.

Every claim in a report also sits in an evidence row that binds five fields before any outcome is read. The hypothesis names the claim class and its population. The comparator is the unguided, null, shuffled or baseline arm. The metric is the terminal statistic with its unit. The uncertainty

field holds the one-sided bound, the error level and the multiplicity rule. The refuting condition says what would contradict the row.

The registered gates. The 53 gates have identifiers that begin with F-. By family they are: foundations 3, substrate 2, conditioning 4, interface-JEPA 10, steering 6, maturation 6, judge 7, coverage 7, immunoglobulin 4, and liability and glycan 4. **Supplementary Table S5** lists them in registry order. Cohort sizes list the sampled objects; the independent resampling unit is specified separately for each gate. Where the margin of a limit depends on a comparator, it is fixed before evaluation and written as such.

Gate	What is tested	Acceptance rule
F-Gphi	Decoder-metric guidance against a Euclidean control: decoded clashes and bond-geometry violations at matched binding enrichment	Lower bound of the paired relative reduction ≥ 0.20; 500 designs
F-AV	Sequential-evidence code on simulated null and displaced streams (diagnostic)	Upper 95% limit on the exclusion rate ≤ 0.15; 200 streams per arm, horizon 2,000
F-CHI	Chirality and improper-dihedral statistics of decoded all-atom structures against a PDB reference	Rate limits ≤ 0.5%; KS distance ≤ 0.10; 1,000 plus 500 decodes, reference ≥ 200 structures
F-KEY	Partially latent interface against an explicit all-atom baseline: CDR-H3 recovery, decoded-versus-refolded RMSD and contacts	Non-inferiority margin, RMSD ceiling and contact floor fixed before evaluation; ≥ 200 complexes
F-DIV	Flow-density arithmetic (traces, step doubling) on synthetic vector fields	Property tests PT-29 to PT-31 pass at registered tolerances; no dataset
F-CB	Dependence on the true antigen: substitution, decoy, removal and renumbering controls	Margin for each control fixed before evaluation
F-MI	Condition-matched contrastive score per residue (a diagnostic, not a mutual-information bound)	Lower limit ≥ 0.20 nats per residue
F-EPI	Epitope precision, coverage and their harmonic mean on predicted complexes	Three floors fixed before evaluation; 1,000 designs
F-CYC	Cycle-consistency accuracy on same-antigen, different-epitope pairs, by antigen-fold stratum	Every stratum lower bound ≥ 0.70; 2,000 pairs, ≥ 100 per stratum
F-J1a	Size of the cognate-versus-decoy score gap, per head	Per-head margin fixed before evaluation; ≥ 200 designs, 10 decoys each
F-J1b	Cognate-versus-decoy AUC, per head	Lower limit ≥ 0.80; as F-J1a
F-J1c	Permutation test of the same gap, per head	$p \leq 0.001$ with 99,999 permutations; as F-J1a
F-J2	Triage recall of co-folding hits at a fixed judge budget	Recall ≥ 0.90 at triage ratio 0.15; gain ≥ +0.10 over each comparator; 1,500 survivors
F-J3	Latent variance, covariance and task gain against a nominated comparator	Variance floor, covariance ceiling and task margin fixed before evaluation; ≥ 200 complexes
F-J4	Effect of adding a head's score on the external judge's target-versus-decoy margin	Lower limit of the margin change ≥ 0; 2,000 target-decoy pairs
F-J5	Violation rate of the head's lower interval on an exchangeable holdout	Upper limit ≤ $\beta + \varepsilon$, both fixed before evaluation

Gate	What is tested	Acceptance rule
F-J6	Residual dependence between head errors and judge errors	Residual correlation ≤ 0.30; score correlation ≤ 0.60; information ≤ 0.10 nats; recall leak ≤ 0.05; ≥ 200 complexes, ≥ 500 survivors, 5% of rejects
F-J7	Crossing frequency and delay of a diagnostic recurrence on simulated streams (diagnostic)	No threshold; ≥ 1,000 streams
F-J8	Polyreactivity transfer from single-domain to paired-Fv format	Paired-Fv AUROC lower limit ≥ 0.70; AUROC ratio lower limit ≥ 0.85; ≥ 300 constructs per format
F-SPEC	Negative-design steering: decoy-margin gain without loss on the on-target judge	Decoy-margin floor and on-target tolerance fixed before evaluation; 1,000 paired designs
F-UGAP	Agreement of 8-particle and 256-particle steering ensembles on bounded functionals	Upper endpoint ≤ 0.5 reference standard deviations; 100 contexts
F-TTS	Offline comparison of an evidence-allocated schedule with the fixed schedule (diagnostic)	Enrichment@50 gain ≥ 0.10; diversity ≥ 0.80 of the fixed schedule; 12 antigens × 200 designs per arm
F-ENT	Entropic reweighting ($\eta > 0$) against $\eta = 0$	Three margins fixed before evaluation
F-MN	Min-norm multi-objective correction against fixed weights	Hypervolume margin and per-objective tolerances fixed before evaluation
F-Dev	Fidelity and effect of the soft developability relaxation	Fidelity, error, disagreement and safeguard limits fixed before evaluation; 1,000 paired designs
F-MAT	Maturation yield, whole-complex rank correlation, humanness and developability minima	Yield margin and two score floors fixed before evaluation
F-M4	Edit-budget fidelity and monotonicity of the budget map	Deviation ≤ max(1, 0.2k) for budget k; 300 parents per length decile
F-M5	Edit targeting by somatic-hypermutation motifs	Descriptive; no threshold
F-M6	Reversion towards germline: humanness gain and affinity retention	Humanness gain ≥ 0.05; affinity fraction ≥ 0.90; no new liability; ≥ 100 parents
F-M7	New CDR liabilities among accepted variants	Upper limit ≤ 0.02; 500 variants
F-M8	Offline comparison of an evidence-allocated maturation schedule (diagnostic)	Yield gain ≥ 0.08; edit-count deviation ≤ max(1, 0.2k); ≥ 200 parents
F-JUDGE	Multi-seed judge with reranker against native co-folder confidence, random order and a physics ranker	Against native confidence: DockQ gain ≥ +0.05 and enrichment@50 gain ≥ +0.10; smaller gains against random order and a physics ranker; ≥ 200 targets
F-BAI	Bad-admission incidence of the pose-admission screen	Upper limit ≤ 0.10; 1,000 designs
F-IND	Rank association between judge residuals and steering-surrogate residuals	Absolute Spearman correlation ≤ 0.25, failure side 0.35; 500 designs
F-C1	Ranking-trained reranker against a pointwise reranker	DockQ gain ≥ +0.01; enrichment gain ≥ +0.02; 200 targets
F-C2	Epitope-registration test on seeded off-epitope and on-epitope designs	Off-epitope rejection ≥ 0.90; on-epitope rejection ≤ 0.10; ≥ 100 per arm
F-C3	State-discrimination margin for targets with two resolved states	AUC lower limit ≥ 0.70; ≥ 30 target-binder pairs

Gate	What is tested	Acceptance rule
F-C4	Offline replay of adaptive pose-search scheduling	Admission incidence ≤ 0.10; cost cut ≥ 30%; DockQ within 0.02; enrichment within 0.05; 600 cases
F-COV	Leave-one-cluster-out coverage gap of the specificity panel	Gap ≤ 0.20 on both arms; 1,000 designs in ≥ 8 clusters
F-DIL	Sensitivity of the reported risk to a deliberately diluted panel	Upper limit ≤ 0.02 on the risk scale; 300 designs, ≥ 3 clusters
F-ORAC	Agreement between the assay-trained polyspecificity oracle and docking	Divergence upper limit ≤ 0.15; positive rank-correlation lower limit; 200 pairs, ≥ 60 assay-labelled
F-BIN	Displacement and direction reversal among competition-positive pairs over a concentration series	Displacement fraction ≤ 0.15; reversal fraction ≤ 0.10; ≥ 40 pairs
F-SET	Diversified panels against a coverage-only comparator	False-discovery proportion limit ≤ 0.30; diversity gain ≥ 0.05; 400 designs, ≥ 20 panels, ≥ 8 folds
F-SIGN	Whether any grid point meets both tier ceilings	At least one grid point meets both ceilings; 1,000 designs
F-DEP	Conditional dependence between steering and measured oracles	Bands at 0.05 and 0.15 nats; 400 labelled designs
F-IMGT	IMGT numbering round trip against independent curated labels	Agreement lower limit ≥ 0.995; zero undocumented disagreements; zero out-of-frame indices; ≥ 600 domains
F-VHH	Conformance of candidates to a registered scaffold profile	Conformance 1.000 over the whole panel
F-LIAB	Liability channel: adjudicated liability rate, affinity retention and charge shift	Margin and direction fixed before evaluation
F-CAN	Canonical-class assignment of CDR loops	Descriptive; no threshold
F-PAIR	Heavy-light pairing: cognate-versus-shuffled discrimination	Descriptive; no threshold
F-IMM	Predicted MHC-II core-load reduction with safeguards on germline identity, pose and affinity	Relative reduction lower bound ≥ 25%; 300 paired designs
F-GLY	Whether glycan accessibility predicts antibody contact (conditioning and epitope admission)	Odds-ratio lower bound ≥ 2.0; 120 complexes
F-GLY-REF	Glycan shielding as part of the reference measure	Gap-difference upper bound ≤ 0.02 and odds-ratio lower bound ≥ 2.0; 1,000 plus 400 designs

Supplementary Table S5. The 53 registered gates with their acceptance rules.

Rates, correlations and gains are dimensionless; counts identify the stated designs, targets, panels or folds and are not uniformly independent units. Most bounds are one-sided at the 95% level; F-C1 and the random-order arm of F-JUDGE use higher levels. Abbreviations: AUC and AUROC, area under the receiver operating characteristic curve; KS, Kolmogorov–Smirnov; DockQ, docking model quality score; CDR, complementarity-determining region; MHC-II, major histocompatibility complex class II; PDB, Protein Data Bank.

A gate may have several arms, and each arm has its own statistic, limit and bound. A gate passes only if every one of its arms passes. An arm below its required sample size fails the gate, and the gate is marked underpowered. A gate with no evaluated arm has neither a pass nor a fail.

The registry is the only source of limits. Before evaluation, fix each arm's threshold, sample unit, terminal sample size, uncertainty procedure and multiplicity correction.

Each numerical limit is an engineering operating point or a refutation target. For F-IMM, F-GLY, F-GLY-REF and F-IMGT the cited sources fix the method and the inputs and not the acceptance value. A limit is re-examined on a fresh prospective set. F-J1a, F-J1b and F-J1c form one conjunction on one head: they share at least 200 designs with ten nested decoys each, and a head is admitted only if all three pass. F-GLY certifies the glycan shielding score for conditioning and epitope admission. F-GLY-REF certifies its use in the reference measure, and a record that cites F-GLY for that use is invalid.

Sample size and power. A pass says little if the test could not have failed. **Supplementary Table S6** shows, for the most common design in the registry, how small a fall each cohort size can detect. The design is a Bernoulli endpoint with a baseline success rate of 0.80, a one-sided exact binomial test at level 0.05 and 80% power.

Independent units	Smallest detectable rate	Smallest detectable fall
50	0.632	0.168
100	0.686	0.114
200	0.726	0.074
300	0.737	0.063
500	0.753	0.047
1,000	0.767	0.033
2,400	0.779	0.021

Supplementary Table S6. Smallest fall in a success rate, from a baseline of 0.80, that a one-sided exact binomial test at level 0.05 detects with 80% power on independent Bernoulli units.
Rates are proportions.

With 200 independent Bernoulli units, the test has 80% power at a success rate of approximately 0.726. Smaller departures from 0.80 have lower power. Failure to reject does not establish equivalence or adequate performance. With 1,000 units, a decrease of approximately 3.3 percentage points gives 80% power. The smallest detectable effect is recorded beside each estimated gate. The table covers rates only, and a correlation, an AUC or a quantile needs its own calculation. The sizes count independent units and not designs. F-TTS lists 2,400 designs, but its test resamples 12 antigens, so its independent units are the 12 antigens.

Validation levels. Six validation levels order the evidence from arithmetic to biology, and a claim needs evidence at the level that matches it. A derivation shows that units agree and cannot show that a design binds.

1. **Algebra** asks whether shapes, units, masks, signs and limiting cases agree. The evidence is derivations and numerical fixtures.
2. **Component** asks whether one module meets its declared contract. The evidence is cluster-disjoint examples and named gates.
3. **Interface** asks whether producer and consumer agree on type, scale, mask and provenance. The evidence is cross-module fixtures.
4. **System** asks whether a fixed campaign replays from its run record and artefacts.
5. **External** asks whether the frozen system transfers to registered benchmark populations. The evidence is independent complexes, panels or assay units.
6. **Experimental** asks whether selected candidates meet measured biological endpoints. The evidence is predeclared laboratory protocols, which this computational work does not contain.

An arm whose required model or data are missing is recorded as not evaluated; a missing value is never estimated.

S4 Code

The package supplies code for training an autoencoder and a flow-matching denoiser from random initialisation on an unconditional task. The listings below are the files as supplied. The text gives the purpose, the interface and the design reason of each module. The mathematics is in the Materials and Methods and in Section S1, and the parameter and cost arithmetic is in Section S5.

Obtaining and running the code. The code is supplied as the archive `Omniverso code package.zip` with the Supplementary Information. The base model needs Python 3.11 and PyTorch 2.11; the data package needs Biopython, and the numerical checks need NumPy, SciPy and pandas. `code/training` holds the base model with its configurations and tests. `code/data` retrieves and cleans PDB entries and performs temporal, duplicate-identity and structural-fingerprint checks. The supplied cleaning code calls ANARCI with package defaults to extract CDR ranges; it does not implement the ANARCII numbering-of-record workflow or the study-specific bit-score threshold of 100. `code/numerics` checks the statements of Section S1 on small synthetic problems. `code/tools` holds the test runner and the script that reproduces the cost arithmetic of Section S5.

The commands below unpack the archive, run the unit tests and train a reduced autoencoder and denoiser on synthetic complexes. `train.py` writes below one storage root, set by `OMNIVERSO_ALLOWED_ROOT` or `--allowed-root`; its write guard refuses every root except `scratch/smoke-local` in the directory that holds `code`, a non-home cloud volume named in `OMNIVERSO_CLOUD_ROOT`, or a directory below `/mnt/fox2`.

```
unzip "Omniverso code package.zip"
python -m pip install torch==2.11.0 numpy
python code/tools/run_tests.py                      # unit tests of the base model
cd code/training
export OMNIVERSO_ALLOWED_ROOT="$(cd ../.. && pwd)/scratch/smoke-local"
python train.py --config configs/smoke_local.json --stage ae \
    --device cpu --backend gloo --output-name demo
python train.py --config configs/smoke_local.json --stage denoiser \
    --device cpu --backend gloo --output-name demo \
    --ae-checkpoint "$OMNIVERSO_ALLOWED_ROOT/runs/demo/ae/checkpoint_00000020.pt"
```

A run on real shards uses a copy of `configs/reference_2gpu.json` with `manifest` set, a positive `--updates` value and one process per card launched with `torchrun`; the denoiser stage also requires `--ae-checkpoint` pointing to an autoencoder checkpoint containing at least one completed update.

Modules of the base model. The module `precision.py` fixes which arithmetic runs in BF16 and which stays in FP32. It is the only place that switches BF16 autocast on. `fp32(device)` returns a context with autocast off, and `trunk_autocast(device, enabled)` returns the BF16 context or an empty one. `set_tf32(enabled)` sets the TF32 flags of matrix products and cuDNN and returns their values for the training log. `FloatLayerNorm` casts its input to FP32 and runs with autocast off. `attention(q, k, v, bias)` calls PyTorch's scaled dot-product attention in the query dtype, with the additive bias cast to that dtype. `linear_float(layer, x)` applies a linear layer in FP32.

Trunk matrix products use BF16 autocast, and the forward and backward passes use mixed precision. Layer normalisation, output heads, losses, coordinate arithmetic, stored parameters and accumulated gradients use FP32, and the softmax inside attention accumulates in FP32. TF32 is off unless a run switches it on.

`code/training/precision.py` (46 lines)

```
"""Explicit compute policy. Storage and accumulated gradients stay float32."""

from contextlib import nullcontext

import torch
from torch import nn
from torch.nn import functional as F


def fp32(device):
    return torch.autocast(device_type=torch.device(device).type, enabled=False)


def trunk_autocast(device, enabled=True):
    if not enabled:
        return nullcontext()
    return torch.autocast(device_type=torch.device(device).type, dtype=torch.bfloat16)


def set_tf32(enabled):
    torch.backends.cuda.matmul.allow_tf32 = bool(enabled)
    torch.backends.cudnn.allow_tf32 = bool(enabled)
    return {
        "matmul": bool(torch.backends.cuda.matmul.allow_tf32),
        "cudnn": bool(torch.backends.cudnn.allow_tf32),
    }


class FloatLayerNorm(nn.LayerNorm):
```

```
    def forward(self, x):
        with fp32(x.device):
            return F.layer_norm(
                x.float(), self.normalized_shape, self.weight, self.bias, self.eps
            )


def attention(q, k, v, bias):
    # The kernel runs in the query dtype (BF16 under autocast) and accumulates the softmax in FP32.
    return F.scaled_dot_product_attention(
        q, k, v, attn_mask=bias.to(q.dtype), dropout_p=0.0
    )


def linear_float(layer, x):
    with fp32(x.device):
        return layer(x.float())
```

models.py defines the autoencoder, the flow denoiser, the centring and unit conversion, the flow path with its noise, the distance bins and the losses. `centre_atoms(batch)` centres the atoms of each complex on the centroid of its present atoms, computed in FP32 in ångströms, and multiplies by `NM_PER_ANGSTROM` $= 0.1$. It is the only place where the unit changes; shards stay in ångströms, and missing and padded atoms are exact zeros.

`SequenceBlock` maps a condition vector to six modulation vectors (shift, scale and gate for attention and for the feed-forward layer). It runs attention with an optional bias from the pair stream and then a gated feed-forward layer of four times the width. A block may first update the pair stream with a small residual network.

`SequenceStack` applies per-layer activation checkpointing [38] in training when the padded length reaches `checkpoint_min_length`. `Encoder` maps the centred coordinates, the atom mask and the residue category (169 numbers per residue) to a deterministic latent of 8 numbers per residue. `Decoder` maps the centred C-alpha position and the latent to all-atom coordinates in nanometres (37×3) and to 21 residue logits. `FlowDenoiser` takes the noisy state (x, z), the two times t_x and t_z and the batch, and returns one velocity for the coordinates and one for the latent. Its pair representation of 256 numbers per residue pair is built from the noisy C-alpha distance in 30 bins, the clipped residue separation (258 categories, one for pairs in different chains) and a same-chain flag. The time condition is two sinusoidal encodings of $1000\,t_x$ and $1000\,t_z$ passed through a small network.

Coordinates inside the model are in nanometres. The flow path runs from noise at time 0 to data at time 1, and the target velocity is data minus noise. Coordinate noise has standard deviation 1 nm per coordinate, one translation per complex with standard deviation 0.2 nm (2 Å) per axis is added to every residue, and latent noise is unit Gaussian. The 30 distance-bin centres are defined in ångströms at 1 to 30. The code compares the distance d in nanometres with the 29 midpoints $m_j = 0.1\,(j + 1/2)$ nm, converted once by multiplication, so that the bin index is

$$k(d) = \#\{\, j \in \{1, \ldots, 29\} : m_j < d \,\}.$$

A distance equal to a midpoint takes the lower bin; dividing the distance by 0.1 instead moves exact midpoints across the boundary through FP32 rounding. The reconstruction error is the sum of squared coordinate errors over present atoms and three coordinates, divided by three times the number of present atoms, in nm^2; it is reported as the per-coordinate RMSE in ångströms, $10\sqrt{\mathrm{MSE}}$, which is $1/\sqrt{3}$ of an atom RMSD.

The autoencoder loss is the reconstruction plus the residue cross-entropy plus 0.01 times the mean squared latent. The flow loss is the mean squared error of the coordinate velocity plus that of the latent velocity, each averaged over its own block with weight 1. Padded keys receive minus infinity in the attention bias, and padded residues are zeroed after both residual updates and in the sampler and decoder outputs. Padding is excluded from the masked objective. The padding test compares fixed physical inputs padded to lengths 5 and 9 in CPU FP32 and requires relative loss and gradient differences no greater than 10^{-5}.

Construction fails unless the width divides by the number of heads and the pair option is `every_layer`, `every_2` or `none` (a pair update in every layer, in layers 2, 4, ..., 14, or nowhere). With the reference configuration the encoder has 127,949,832 parameters, the decoder 127,923,844 and the autoencoder 255,873,676; the denoiser has 153,102,091, 151,258,123 or 149,414,155 parameters for the three pair options, against about 159 M for the published reference denoiser [11]. The coordinate loss uses nanometre coordinates; the latent is not standardised, and both block losses receive weight 1.

`code/training/models.py` (406 lines)

```
"""Deterministic autoencoder and unconditional block-time flow network."""

import math

import torch
from torch import nn
from torch.nn import functional as F
from torch.utils.checkpoint import checkpoint

from precision import FloatLayerNorm, attention, fp32, linear_float


# Shards use Angstrom; centred model coordinates and coordinate noise use nm.
NM_PER_ANGSTROM = 0.1
TRANSLATION_NOISE_NM = 0.2


def parameter_report(module):
    tensors = {
        name: {
            "shape": list(p.shape),
            "count": p.numel(),
            "dtype": str(p.dtype),
            "trainable": p.requires_grad,
        }
```

```
        for name, p in module.named_parameters()
    }
    return {"total": sum(p.numel() for p in module.parameters()), "tensors": tensors}


class CountedModule(nn.Module):
    def parameter_report(self):
        return parameter_report(self)


def sinusoidal(values, width):
    with fp32(values.device):
        frequencies = torch.exp(
            torch.arange(width // 2, device=values.device, dtype=torch.float32)
            * (-math.log(10000.0) / max(width // 2 - 1, 1))
        )
        phase = values.float().unsqueeze(-1) * frequencies
        return torch.cat((phase.sin(), phase.cos()), -1)


def centre_atoms(batch):
    with fp32(batch["atom37"].device):
        mask = batch["atom_mask"] & batch["mask"].unsqueeze(-1)
        atoms = batch["atom37"].float()
        centroid = (atoms * mask.unsqueeze(-1)).sum((1, 2)) / mask.sum(
            (1, 2)
        ).clamp_min(1).unsqueeze(-1)
        centred = (atoms - centroid[:, None, None, :]) * mask.unsqueeze(-1)
        return centred * NM_PER_ANGSTROM, mask


class SequenceBlock(CountedModule):
    def __init__(self, width, heads, cond_width, pair_width=0, update_pair=False):
        super().__init__()
        self.heads = heads
        self.width = width
        self.norm1 = FloatLayerNorm(width)
        self.norm2 = FloatLayerNorm(width)
        self.modulation = nn.Linear(cond_width, 6 * width)
        self.qkv = nn.Linear(width, 3 * width)
        self.projection = nn.Linear(width, width)
        # Attention projections (4 * width^2) plus gated feed-forward (12 * width^2): 16 * width^2 weights per
block.
        self.ff_in = nn.Linear(width, 8 * width)
        self.ff_out = nn.Linear(4 * width, width)
        self.pair_bias = (
            nn.Linear(pair_width, heads, bias=False) if pair_width else None
        )
        self.pair_mlp = (
            nn.Sequential(
                FloatLayerNorm(pair_width),
                nn.Linear(pair_width, 2 * pair_width),
                nn.GELU(),
                nn.Linear(2 * pair_width, pair_width),
            )
            if update_pair
            else None
        )

    def forward(self, h, pair, cond, mask):
        if self.pair_mlp is not None:
            pair = (pair + self.pair_mlp(pair)) * (
                mask[:, :, None] & mask[:, None, :]
            ).unsqueeze(-1)
        shift1, scale1, gate1, shift2, scale2, gate2 = (
            self.modulation(cond).unsqueeze(1).chunk(6, -1)
        )
        qkv = self.qkv(self.norm1(h) * (1 + scale1) + shift1)
        b, length, _ = qkv.shape
        q, k, v = qkv.reshape(
            b, length, 3, self.heads, self.width // self.heads
        ).permute(2, 0, 3, 1, 4)
        bias = (
            self.pair_bias(pair).permute(0, 3, 1, 2).float()
            if self.pair_bias is not None
            else h.new_zeros(b, self.heads, length, length, dtype=torch.float32)
        )
```

```
        # Mask keys. Padded queries are zeroed after both residuals.
        bias = bias.masked_fill(~mask[:, None, None, :], float("-inf"))
        attended = (
            attention(q, k, v, bias).transpose(1, 2).reshape(b, length, self.width)
        )
        h = (h + gate1.sigmoid() * self.projection(attended)) * mask.unsqueeze(-1)
        value, gate = self.ff_in(self.norm2(h) * (1 + scale2) + shift2).chunk(2, -1)
        h = (h + gate2.sigmoid() * self.ff_out(F.silu(gate) * value)) * mask.unsqueeze(
            -1
        )
        return h, pair


class SequenceStack(CountedModule):
    def __init__(
        self,
        layers,
        width,
        heads,
        cond_width,
        pair_width=0,
        pair_option="none",
        checkpoint_min_length=300,
        checkpointing=True,
    ):
        super().__init__()
        if width % heads or pair_option not in ("every_layer", "every_2", "none"):
            raise ValueError("Invalid width/heads or pair_option")
        self.checkpointing = checkpointing
        self.checkpoint_min_length = checkpoint_min_length
        self.layers = nn.ModuleList(
            [
                SequenceBlock(
                    width,
                    heads,
                    cond_width,
                    pair_width,
                    pair_width > 0
                    and (
                        pair_option == "every_layer"
                        or (pair_option == "every_2" and i % 2 == 1)
                    ),
                )
                for i in range(layers)
            ]
        )
        self.norm = FloatLayerNorm(width)

    def forward(self, h, pair, cond, mask):
        for layer in self.layers:
            if (
                self.checkpointing
                and self.training
                and h.shape[1] >= self.checkpoint_min_length
            ):
                h, pair = checkpoint(layer, h, pair, cond, mask, use_reentrant=False)
            else:
                h, pair = layer(h, pair, cond, mask)
        return self.norm(h) * mask.unsqueeze(-1)


class Encoder(CountedModule):
    def __init__(self, cfg):
        super().__init__()
        width, cond = cfg["width"], cfg["cond_width"]
        self.input = nn.Linear(111 + 37 + 21, width)
        self.index = nn.Linear(cfg["index_width"], width)
        self.index_width = cfg["index_width"]
        self.condition = nn.Parameter(torch.zeros(cond))
        self.stack = SequenceStack(
            cfg["ae_layers"],
            width,
            cfg["heads"],
            cond,
            checkpointing=cfg["checkpointing"],
            checkpoint_min_length=cfg["checkpoint_min_length"],
        )
        self.latent_head = nn.Linear(width, cfg["latent_width"])
```

```
    def forward(self, batch):
        # Encoder coordinate features are centred nm, converted from the Angstrom batch once.
        atoms, atom_mask = centre_atoms(batch)
        with fp32(atoms.device):
            features = torch.cat(
                (
                    atoms.flatten(-2),
                    atom_mask.float(),
                    F.one_hot(batch["aatype"].long(), 21).float(),
                ),
                -1,
            )
            h = self.input(features) + self.index(
                sinusoidal(batch["residx"], self.index_width)
            )
        cond = self.condition.expand(h.shape[0], -1)
        h = self.stack(h, None, cond, batch["mask"])
        return linear_float(self.latent_head, h) * batch["mask"].unsqueeze(-1)


class Decoder(CountedModule):
    def __init__(self, cfg):
        super().__init__()
        width, cond = cfg["width"], cfg["cond_width"]
        self.input = nn.Linear(3 + cfg["latent_width"], width)
        self.index = nn.Linear(cfg["index_width"], width)
        self.index_width = cfg["index_width"]
        self.condition = nn.Parameter(torch.zeros(cond))
        self.stack = SequenceStack(
            cfg["ae_layers"],
            width,
            cfg["heads"],
            cond,
            checkpointing=cfg["checkpointing"],
            checkpoint_min_length=cfg["checkpoint_min_length"],
        )
        self.coordinate_head = nn.Linear(width, 111)
        self.residue_head = nn.Linear(width, 21)

    def forward(self, ca, z, batch):
        # C-alpha input and all-atom coordinate output are in nm; latents are unchanged.
        with fp32(ca.device):
            h = self.input(torch.cat((ca.float(), z.float()), -1)) + self.index(
                sinusoidal(batch["residx"], self.index_width)
            )
        h = self.stack(h, None, self.condition.expand(h.shape[0], -1), batch["mask"])
        keep = batch["mask"].unsqueeze(-1)
        coords = (linear_float(self.coordinate_head, h) * keep).reshape(
            *h.shape[:2], 37, 3
        )
        logits = linear_float(self.residue_head, h) * keep
        return coords, logits


class Autoencoder(CountedModule):
    def __init__(self, cfg):
        super().__init__()
        self.encoder, self.decoder = Encoder(cfg), Decoder(cfg)
        self.latent_penalty = cfg.get("latent_penalty", 0.01)

    def forward(self, batch):
        # Reconstruction targets and predictions are in nm, so reconstruction MSE is in nm^2.
        atoms, mask = centre_atoms(batch)
        z = self.encoder(batch)
        coords, logits = self.decoder(atoms[:, :, 1], z, batch)
        with fp32(atoms.device):
            reconstruction = (
                (coords.float() - atoms).square() * mask.unsqueeze(-1)
            ).sum((1, 2, 3)) / (3 * mask.sum((1, 2)).clamp_min(1))
            sequence = F.cross_entropy(
                logits.float().transpose(1, 2), batch["aatype"].long(), reduction="none"
            )
            sequence = (sequence * batch["mask"]).sum(1) / batch["mask"].sum(
                1
            ).clamp_min(1)
            penalty = (z.float().square() * batch["mask"].unsqueeze(-1)).sum((1, 2)) / (
                z.shape[-1] * batch["mask"].sum(1).clamp_min(1)
```

```
            )
            loss = reconstruction + sequence + self.latent_penalty * penalty
            # Per-coordinate RMSE in Angstrom; atom RMSD would be 10 * sqrt(3 * reconstruction).
            coordinate_rmse_angstrom = 10 * reconstruction.sqrt()
        return {
            "loss": loss,
            "reconstruction": reconstruction,
            "sequence": sequence,
            "latent_penalty": penalty,
            "coordinate_rmse_angstrom": coordinate_rmse_angstrom,
            "z": z,
            "coordinates": coords,
            "logits": logits,
        }


class FlowDenoiser(CountedModule):
    def __init__(self, cfg):
        super().__init__()
        width, pair_width, cond = cfg["width"], cfg["pair_width"], cfg["cond_width"]
        self.input = nn.Linear(3 + cfg["latent_width"], width)
        self.index_width, self.time_width = cfg["index_width"], cfg["time_width"]
        self.index = nn.Linear(self.index_width, width)
        self.time = nn.Sequential(
            nn.Linear(2 * self.time_width, cond), nn.SiLU(), nn.Linear(cond, cond)
        )
        self.separation = nn.Embedding(258, cfg["separation_width"])
        self.separation_to_pair = nn.Linear(cfg["separation_width"], pair_width)
        self.chain_pair = nn.Embedding(2, pair_width)
        self.distance_to_pair = nn.Linear(30, pair_width)
        self.register_buffer("distance_centres", torch.linspace(1, 30, 30))
        self.stack = SequenceStack(
            cfg["denoiser_layers"],
            width,
            cfg["heads"],
            cond,
            pair_width,
            cfg["pair_option"],
            cfg["checkpoint_min_length"],
            cfg["checkpointing"],
        )
        self.coordinate_head = nn.Linear(width, 3)
        self.latent_head = nn.Linear(width, cfg["latent_width"])

    def forward(self, x, z, tx, tz, batch):
        with fp32(x.device):
            x, z = x.float(), z.float()
            h = self.input(torch.cat((x, z), -1)) + self.index(
                sinusoidal(batch["residx"], self.index_width)
            )
            # Explicit subtraction avoids length-dependent cdist algorithms.
            distances = (x[:, :, None] - x[:, None, :]).square().sum(-1).sqrt()
            # Lock v2 nearest-centre bins are defined in Angstrom (centres 1 to 30). The midpoints are
converted
            # once to nm with the multiplication that converts the coordinates, so a distance that equals a
            # midpoint falls in the lower bin and no division adds rounding noise to the comparison.
            boundaries = (
                (self.distance_centres[1:] + self.distance_centres[:-1]) / 2
            ) * NM_PER_ANGSTROM
            bins = F.one_hot(
                torch.bucketize(distances.contiguous(), boundaries), 30
            ).float()
        same_chain = batch["chain"][:, :, None] == batch["chain"][:, None, :]
        separation = (batch["residx"][:, :, None] - batch["residx"][:, None, :]).clamp(
            -128, 128
        ) + 128
        separation = torch.where(same_chain, separation, 257).long()
        pair = (
            self.distance_to_pair(bins)
            + self.separation_to_pair(self.separation(separation))
            + self.chain_pair(same_chain.long())
        )
        pair = pair * (batch["mask"][:, :, None] & batch["mask"][:, None, :]).unsqueeze(
            -1
        )
        cond = self.time(
            torch.cat(
                (
                    sinusoidal(tx * 1000, self.time_width),
```

```
                    sinusoidal(tz * 1000, self.time_width),
                ),
                -1,
            )
        )
        h = self.stack(h, pair, cond, batch["mask"])
        vx = linear_float(self.coordinate_head, h) * batch["mask"].unsqueeze(-1)
        vz = linear_float(self.latent_head, h) * batch["mask"].unsqueeze(-1)
        return vx, vz


def draw_coordinate_noise(x_data):
    with fp32(x_data.device):
        # Independent 1 nm coordinate noise plus one shared 0.2 nm translation per complex.
        independent = torch.randn_like(x_data, dtype=torch.float32)
        translation = TRANSLATION_NOISE_NM * torch.randn(
            x_data.shape[0], 1, 3, device=x_data.device, dtype=torch.float32
        )
        return independent + translation, translation


def flow_interpolate(data, noise, time):
    with fp32(data.device):
        weight = time.float()[:, None, None]
        data, noise = data.float(), noise.float()
        return (1 - weight) * noise + weight * data, data - noise


def random_so3_matrices(batch_size, device):
    with fp32(device):
        q, r = torch.linalg.qr(
            torch.randn(batch_size, 3, 3, device=device, dtype=torch.float32)
        )
        signs = torch.diagonal(r, dim1=-2, dim2=-1).sign()
        signs = torch.where(signs == 0, torch.ones_like(signs), signs)
        q = q * signs.unsqueeze(-2)
        det = torch.linalg.det(q)
        flip = torch.ones(batch_size, 3, device=device, dtype=torch.float32)
        flip[:, 0] = torch.where(det < 0, -torch.ones_like(det), torch.ones_like(det))
        return q * flip.unsqueeze(-2)


def flow_sample(batch, latent, rotate=False):
    atoms, _ = centre_atoms(batch)
    with fp32(atoms.device):
        x_data, z_data = atoms[:, :, 1], latent.float()
        if rotate:
            rot = random_so3_matrices(x_data.shape[0], atoms.device)
            x_data = torch.einsum("bli,bij->blj", x_data, rot)
        x_noise, _ = draw_coordinate_noise(x_data)
        z_noise = torch.randn_like(z_data)
        tx = torch.rand(x_data.shape[0], device=atoms.device)
        tz = torch.rand(x_data.shape[0], device=atoms.device)
        x, target_x = flow_interpolate(x_data, x_noise, tx)
        z, target_z = flow_interpolate(z_data, z_noise, tz)
        # Padded residues carry exact zeros, as the padding rule of Lock v2 requires.
        keep = batch["mask"].unsqueeze(-1)
        return x * keep, z * keep, tx, tz, target_x * keep, target_z * keep


def flow_loss(vx, vz, target_x, target_z, mask):
    # Coordinate state and target velocity are in nm; x_loss is in nm^2. Latent loss is unchanged.
    with fp32(vx.device):
        count = mask.sum(1).clamp_min(1)
        lx = ((vx.float() - target_x.float()).square() * mask.unsqueeze(-1)).sum(
            (1, 2)
        ) / (3 * count)
        lz = ((vz.float() - target_z.float()).square() * mask.unsqueeze(-1)).sum(
            (1, 2)
        ) / (vz.shape[-1] * count)
        return {"loss": lx + lz, "x_loss": lx, "z_loss": lz}
```

data.py reads shards, builds synthetic complexes for the unit tests and draws the complexes of each update. `seed_for(*parts)` returns the first 32 bits of the SHA-256 of the JSON text of its

arguments, and the training entry point derives the model, data, sampler and noise seeds through it.

`synthetic_complex(seed, length, complex_id)` builds a random-walk backbone with noisy side atoms, a mask that removes about 8 percent of the atoms (never the C-alpha atom), random residue categories and two chains, and `pack_shard` returns the bytes of an NPZ shard without pickle. `ShardReader(manifest)` reads a manifest of schema 1, checks each shard against its SHA-256, keeps all shards in memory and exposes `ids`, `lengths` and item access. It refuses a missing file, a shard outside the manifest directory, a hash mismatch, a wrong dtype or shape and a non-finite coordinate. It also refuses a category outside 0 to 20, a repeated `complex_id`, offsets that do not start at zero and increase, a residue without a C-alpha atom at slot 1, a repeated (chain, residue index) pair and a manifest whose units are not ångströms. `collate(complexes, length, device)` pads to a bucket length and returns the batch with masks, identifiers and residue keys.

`BucketSampler(run_id, buckets, reader, global_batch)` assigns each complex to the bucket of the smallest limit that holds it, and `draw(update_index, rank, world_size, forced_bucket)` returns the bucket and the index lists of one rank. An empty bucket is an error. The draw depends only on the run identifier and the update index. All ranks therefore pick the same bucket and the same 48 indices (with replacement), and each takes its own slice. A resumed run sees the batches it would have seen without the interruption. The global batch must divide by the world size times the microbatch.

`code/training/data.py` (224 lines)

```
"""One-shard reader, deterministic synthetic inputs, and update-index sampling."""

import hashlib
import io
import json
from pathlib import Path
import random

import numpy as np
import torch


FIELDS = {
    "atom37": np.dtype("float32"),
    "atom_mask": np.dtype("bool"),
    "aatype": np.dtype("int8"),
    "chain": np.dtype("int16"),
    "residx": np.dtype("int32"),
}


def seed_for(*parts):
    # Parameter Lock v2 (carrying forward Lock v1 section 4.2): first 32 bits of SHA-256 over the identifiers.
    return int.from_bytes(
        hashlib.sha256(json.dumps(parts).encode()).digest()[:4], "big"
    )


def sha256(path):
    digest = hashlib.sha256()
```

```
    with open(path, "rb") as handle:
        for chunk in iter(lambda: handle.read(1024 * 1024), b""):
            digest.update(chunk)
    return digest.hexdigest()


def synthetic_complex(seed, length, complex_id):
    generator = torch.Generator(device="cpu").manual_seed(seed)
    ca = torch.cumsum(
        0.7 * torch.randn(length, 3, generator=generator, dtype=torch.float32), 0
    )
    atoms = ca[:, None] + 0.15 * torch.randn(
        length, 37, 3, generator=generator, dtype=torch.float32
    )
    atom_mask = torch.rand(length, 37, generator=generator) > 0.08
    atom_mask[:, 1] = True
    atoms *= atom_mask[..., None]
    return {
        "atom37": atoms.numpy(),
        "atom_mask": atom_mask.numpy(),
        "aatype": torch.randint(
            0, 21, (length,), generator=generator, dtype=torch.int8
        ).numpy(),
        "chain": (torch.arange(length) >= length // 2).to(torch.int16).numpy(),
        "residx": torch.arange(length, dtype=torch.int32).numpy(),
        "complex_id": str(complex_id),
    }


def pack_shard(complexes):
    # No object arrays or pickle: offsets index concatenated residue arrays.
    result = {name: np.concatenate([c[name] for c in complexes]) for name in FIELDS}
    result["offsets"] = np.array(
        [0] + list(np.cumsum([len(c["aatype"]) for c in complexes])), dtype=np.int64
    )
    result["complex_id"] = np.array([c["complex_id"] for c in complexes], dtype=np.str_)
    buffer = io.BytesIO()
    np.savez(buffer, **result)
    return buffer.getvalue()


class ShardReader:
    def __init__(self, manifest):
        self.manifest = Path(manifest).resolve()
        with open(self.manifest, encoding="utf-8") as handle:
            doc = json.load(handle)
        if doc.get("schema") != 1 or not doc.get("shards"):
            raise ValueError("No shards present, or unsupported manifest schema")
        if doc.get("units", "angstrom") != "angstrom":
            raise ValueError("Shard coordinates must use Angstrom units")
        self.shards = doc["shards"]
        # Every shard stays resident, so random access never rereads a shard from disk.
        self.loaded = [self._read(i) for i in range(len(self.shards))]
        self.ids, self.lengths, self.locations = [], [], []
        for i, arrays in enumerate(self.loaded):
            offsets = arrays["offsets"]
            for j, identifier in enumerate(arrays["complex_id"]):
                self.ids.append(str(identifier))
                self.lengths.append(int(offsets[j + 1] - offsets[j]))
                self.locations.append((i, j))
        if len(set(self.ids)) != len(self.ids):
            raise ValueError("complex_id must be globally unique")

    def _read(self, index):
        record = self.shards[index]
        path = (self.manifest.parent / record["file"]).resolve()
        if not path.is_relative_to(self.manifest.parent) or not path.is_file():
            raise ValueError(f"No shard present under manifest directory: {path}")
        if sha256(path).lower() != record["sha256"].lower():
            raise ValueError(f"SHA-256 mismatch: {path}")
        # npz members are read into memory once; ShardReader keeps all shards resident.
        with np.load(path, allow_pickle=False, mmap_mode="r") as archive:
            arrays = {
                name: archive[name] for name in (*FIELDS, "offsets", "complex_id")
            }
        offsets, ids = arrays["offsets"], arrays["complex_id"]
        if (
            offsets.dtype != np.int64
```

```
            or offsets.ndim != 1
            or len(offsets) != len(ids) + 1
            or offsets[0] != 0
            or np.any(np.diff(offsets) <= 0)
        ):
            raise ValueError("Invalid shard offsets")
        n = int(offsets[-1])
        shapes = {
            "atom37": (n, 37, 3),
            "atom_mask": (n, 37),
            "aatype": (n,),
            "chain": (n,),
            "residx": (n,),
        }
        for name, dtype in FIELDS.items():
            if arrays[name].dtype != dtype or arrays[name].shape != shapes[name]:
                raise ValueError(f"Invalid dtype or shape for {name}")
        if (
            ids.ndim != 1
            or ids.dtype.kind != "U"
            or not np.isfinite(arrays["atom37"]).all()
            or np.any((arrays["aatype"] < 0) | (arrays["aatype"] > 20))
        ):
            raise ValueError("Invalid coordinates, categories, or complex_id")
        for start, end in zip(offsets[:-1], offsets[1:]):
            if not arrays["atom_mask"][start:end, 1].all():
                raise ValueError("Every residue needs a C-alpha atom at index 1")
            if (
                len(
                    set(
                        zip(
                            arrays["chain"][start:end].tolist(),
                            arrays["residx"][start:end].tolist(),
                        )
                    )
                )
                != end - start
            ):
                raise ValueError("(chain, residx) must identify a unique residue")
        return arrays

    def __len__(self):
        return len(self.ids)

    def __getitem__(self, index):
        shard, member = self.locations[index]
        arrays = self.loaded[shard]
        start, end = arrays["offsets"][member : member + 2]
        return {
            **{name: arrays[name][start:end] for name in FIELDS},
            "complex_id": self.ids[index],
        }


def collate(complexes, length, device="cpu"):
    count = len(complexes)
    result = {
        "atom37": torch.zeros(count, length, 37, 3, dtype=torch.float32),
        "atom_mask": torch.zeros(count, length, 37, dtype=torch.bool),
        "aatype": torch.zeros(count, length, dtype=torch.long),
        "chain": torch.zeros(count, length, dtype=torch.long),
        "residx": torch.zeros(count, length, dtype=torch.long),
        "mask": torch.zeros(count, length, dtype=torch.bool),
    }
    for i, item in enumerate(complexes):
        n = len(item["aatype"])
        if not 0 < n <= length:
            raise ValueError("Complex length outside selected bucket")
        for name in FIELDS:
            result[name][i, :n] = torch.as_tensor(item[name].copy())
        result["mask"][i, :n] = True
    return {
        **{name: value.to(device) for name, value in result.items()},
        "complex_id": [c["complex_id"] for c in complexes],
        "residue_keys": [
            list(zip(c["chain"].tolist(), c["residx"].tolist())) for c in complexes
        ],
    }
```

```
class BucketSampler:
    def __init__(self, run_id, buckets, reader, global_batch=48):
        self.run_id, self.buckets, self.global_batch = run_id, buckets, global_batch
        lower = 0
        self.members = []
        for bucket in buckets:
            upper = bucket["length"]
            indices = [
                i for i, length in enumerate(reader.lengths) if lower < length <= upper
            ]
            if not indices:
                raise ValueError(f"Empty length bucket ({lower}, {upper}]")
            self.members.append(indices)
            lower = upper

    def draw(self, update_index, rank, world_size, forced_bucket=None):
        rng = random.Random(seed_for(self.run_id, update_index))
        bucket_index = (
            rng.randrange(len(self.buckets))
            if forced_bucket is None
            else int(forced_bucket)
        )
        bucket = self.buckets[bucket_index]
        microbatch = bucket["microbatch"]
        if self.global_batch % (world_size * microbatch):
            raise ValueError("Global batch must divide world size times microbatch")
        accumulation = self.global_batch // (world_size * microbatch)
        indices = rng.choices(self.members[bucket_index], k=self.global_batch)
        local = [
            indices[
                (step * world_size + rank) * microbatch : (step * world_size + rank + 1)
                * microbatch
            ]
            for step in range(accumulation)
        ]
        return bucket_index, bucket, local
```

train.py is the entry point. It checks the storage root, builds the data and the model of one stage (`ae` or `denoiser`), runs the update loop and writes checkpoints, logs and a run record. `WriteGuard` resolves every path and writes atomically (temporary file, flush, `fsync`, rename). `TrainingState` holds the AdamW optimiser, the warm-up schedule, the EMA, the step and the cursor, and offers `finish_update`, `payload` and `restore`. The options select the configuration and stage, the device and backend, the storage root and the run name. They also set a total update count (`--updates`), a wall-time limit for the loop (`--max-wall-seconds`), a checkpoint to resume (`--resume`) and the autoencoder checkpoint of the denoiser stage (`--ae-checkpoint`). Further options override the pair option, checkpointing, latent cache and EMA start, and `--compare-checkpoint` compares the final state with a reference checkpoint.

One update uses 48 complexes. Each rank draws its slice and runs its microsteps. A microstep collates a batch, runs the forward pass under BF16 autocast and backpropagates the sum of the per-complex losses divided by 48. Because of this scale the gradients of the ranks must be summed. A communication hook performs an FP32 sum all-reduce and refuses a non-FP32 buffer, and `no_sync` suppresses the reduction on all but the last microstep [39]. In the denoiser stage the latent comes from the frozen encoder under `no_grad` or from a latent cache.

After the last microstep `finish_update` clips the global gradient 2-norm to 1.0 and forms a finite flag, takes its minimum over the ranks, and applies or skips the update. A skip clears the gradients, leaves weights, optimiser, EMA, schedule and cursor unchanged, and restores the random state so that the retry draws the same noise. Three consecutive skips stop the run. An applied update advances the step, the cursor and the warm-up and then updates the EMA, which copies the parameters at the first successful update after `ema_start` and uses decay 0.999 afterwards.

A checkpoint is written after a complete optimiser, EMA and schedule step. Every rank writes its random state, a barrier follows, and rank 0 writes the model file last, so the model file marks a complete checkpoint. The last three checkpoints and every checkpoint named by the run record are kept.

A resume needs the same stage, configuration digest, device type, world size and coordinate contract, and a checkpoint without the origin tag of this code is refused before any weight loads. Checkpoints are PyTorch pickles and must come from a trusted source. The denoiser stage needs an autoencoder checkpoint of at least one completed update. The latent cache is valid only in the denoiser stage with canonical encoder input (no rotation, no crop). It is keyed by the encoder checksum, the data manifest, the precision flags, the coordinate contract and the hashes of `models.py`, `precision.py` and `data.py`. Before training, the free storage must exceed the estimated need by 10 percent and the host-memory estimate must stay below the configured ceiling. One script with a few explicit rules (summed gradients, one finite flag, a model file written last, a refused mismatch) makes each failure stop the run or leave the state as it was.

`code/training/train.py` (1256 lines)

```
"""Offline training entry point. All persistent output uses WriteGuard."""

import argparse
from contextlib import nullcontext
import hashlib
import json
import os
from pathlib import Path
import random
import shutil
import statistics
import subprocess
import time
import zipfile

import numpy as np
import torch
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

from data import (
    BucketSampler,
    ShardReader,
    collate,
```

```
    pack_shard,
    seed_for,
    sha256,
    synthetic_complex,
)
from models import (
    Autoencoder,
    FlowDenoiser,
    NM_PER_ANGSTROM,
    TRANSLATION_NOISE_NM,
    flow_loss,
    flow_sample,
    parameter_report,
)
from precision import set_tf32, trunk_autocast


CACHE_VARIABLES = (
    "XDG_CACHE_HOME",
    "TORCH_HOME",
    "HF_HOME",
    "TRITON_CACHE_DIR",
    "TORCHINDUCTOR_CACHE_DIR",
    "CUDA_CACHE_PATH",
    "PIP_CACHE_DIR",
    "TMPDIR",
    "APPTAINER_CACHEDIR",
    "APPTAINER_TMPDIR",
    "SINGULARITY_CACHEDIR",
    "PYTHONPYCACHEPREFIX",
)
LOCAL_ROOT = Path(__file__).resolve().parents[2] / "scratch" / "smoke-local"
ORIGIN = "omniverso-cycle1-own-training"
COORDINATE_CONTRACT = {
    "coordinate_unit": "nm",
    "nm_per_angstrom": NM_PER_ANGSTROM,
    "translation_noise_nm": TRANSLATION_NOISE_NM,
    "distance_bins": "30 centres, 1 to 30 Angstrom after conversion",
    "reconstruction_metric": "per-coordinate RMSE in Angstrom = 10*sqrt(MSE in nm^2)",
}


def require_coordinate_contract(payload):
    if payload.get("coordinate_contract") != COORDINATE_CONTRACT:
        raise ValueError(
            "Checkpoint coordinate contract is missing or incompatible; retrain both stages in nm"
        )


class WriteGuard:
    def __init__(self, root):
        self.root = Path(root).expanduser().resolve()
        self.home = Path.home().resolve()
        local = self.root == LOCAL_ROOT.resolve()
        cloud_env = os.environ.get("OMNIVERSO_CLOUD_ROOT")
        cloud = False
        if cloud_env:
            cloud_root = Path(cloud_env).expanduser().resolve()
            if cloud_root != Path(cloud_root.anchor) and not cloud_root.is_relative_to(
                self.home
            ):
                cloud = self.root == cloud_root or self.root.is_relative_to(cloud_root)
        # Only the local smoke root, an explicit non-home cloud mount, or /mnt/fox2 is allowed.
        if not (local or cloud or self.root.as_posix().startswith("/mnt/fox2/")):
            raise ValueError(
                f"Refused storage root outside allowed local/cloud mounts: {self.root}"
            )
        if self.root.is_relative_to(self.home) and not local:
            raise ValueError(f"Refused output root under user home: {self.root}")
        if self.root == Path(self.root.anchor):
            raise ValueError("Refused filesystem root as allowed root")

    def path(self, path):
        path = Path(path).expanduser()
        target = (path if path.is_absolute() else self.root / path).resolve()
        if not target.is_relative_to(self.root):
            raise ValueError(f"Refused output outside allowed root: {target}")
        if target.is_relative_to(self.home) and not target.is_relative_to(
```

```
            LOCAL_ROOT.resolve()
        ):
            raise ValueError(f"Refused output under user home: {target}")
        return target

    def mkdir(self, path):
        target = self.path(path)
        target.mkdir(parents=True, exist_ok=True)
        return target

    def atomic(self, path, writer):
        target = self.path(path)
        self.mkdir(target.parent)
        temporary = self.path(str(target) + f".tmp-{os.getpid()}")
        try:
            with open(temporary, "wb") as handle:
                writer(handle)
                handle.flush()
                os.fsync(handle.fileno())
            os.replace(temporary, self.path(target))
            # Persist the rename on systems supporting directory fsync.
            if os.name != "nt":
                fd = os.open(target.parent, os.O_RDONLY | getattr(os, "O_DIRECTORY", 0))
                try:
                    os.fsync(fd)
                finally:
                    os.close(fd)
        finally:
            if temporary.exists():
                self.path(temporary).unlink()
        return target

    def json(self, path, value):
        return self.atomic(
            path,
            lambda handle: handle.write(
                (json.dumps(value, indent=2, allow_nan=False) + "\n").encode()
            ),
        )

    def save(self, path, value):
        return self.atomic(path, lambda handle: torch.save(value, handle))


def configure_environment(guard):
    values = {
        "OMNIVERSO_ALLOWED_ROOT": str(guard.root),
        "OMNIVERSO_ROOT": str(guard.root),
    }
    for name in CACHE_VARIABLES:
        default = guard.root / ("tmp" if "TMP" in name else "cache") / name.lower()
        values[name] = str(guard.mkdir(os.environ.get(name, str(default))))
    for name in ("TEMP", "TMP"):
        # Windows normally supplies home-based defaults for these variables.
        values[name] = str(guard.mkdir(guard.root / "tmp"))
    os.environ.update(values)
    return values


def rng_state(device):
    state = {
        "python": random.getstate(),
        "numpy": np.random.get_state(),
        "torch": torch.get_rng_state(),
    }
    if torch.device(device).type == "cuda":
        state["cuda"] = torch.cuda.get_rng_state(device)
    return state


def restore_rng(state, device):
    random.setstate(state["python"])
    np.random.set_state(state["numpy"])
    torch.set_rng_state(state["torch"].cpu())
    if "cuda" in state:
        torch.cuda.set_rng_state(state["cuda"].cpu(), device)
```

```
class Warmup:
    def __init__(self, optimizer, lr, warmup):
        self.optimizer, self.lr, self.warmup, self.step = optimizer, lr, warmup, 0
        self._apply()

    def _apply(self):
        value = self.lr * min(1.0, (self.step + 1) / max(1, self.warmup))
        for group in self.optimizer.param_groups:
            group["lr"] = value

    def advance(self):
        self.step += 1
        self._apply()

    def state_dict(self):
        return {"lr": self.lr, "warmup": self.warmup, "step": self.step}

    def load_state_dict(self, value):
        self.lr, self.warmup, self.step = value["lr"], value["warmup"], value["step"]
        self._apply()


class TrainingState:
    def __init__(self, model, cfg, device):
        self.model, self.cfg, self.device = model, cfg, torch.device(device)
        self.optimizer = torch.optim.AdamW(
            model.parameters(),
            lr=cfg["learning_rate"],
            betas=(0.9, 0.999),
            eps=1e-8,
            weight_decay=0.0,
            fused=self.device.type == "cuda",
        )
        self.scheduler = Warmup(
            self.optimizer, cfg["learning_rate"], cfg["warmup_steps"]
        )
        self.ema, self.step, self.cursor = {}, 0, 0

    def finish_update(self):
        parameters = [p for p in self.model.parameters() if p.grad is not None]
        norm = torch.nn.utils.clip_grad_norm_(
            parameters, self.cfg["clip_norm"], error_if_nonfinite=False
        )
        finite = (
            torch.isfinite(norm)
            & torch.stack([torch.isfinite(p.grad).all() for p in parameters]).all()
        ).int()
        if dist.is_initialized():
            dist.all_reduce(finite, op=dist.ReduceOp.MIN)
        # This scalar is the coherent skip decision, not a norm/logging transfer.
        # Its host synchronisation is accounted for in complete-update wall time.
        if not bool(finite.item()):
            self.optimizer.zero_grad(set_to_none=True)
            return False, norm
        self.optimizer.step()
        self.step += 1
        self.cursor += 1
        self.scheduler.advance()
        with torch.no_grad():
            if self.step > self.cfg["ema_start"]:
                if not self.ema:
                    self.ema = {
                        name: p.detach().clone().float()
                        for name, p in self.model.named_parameters()
                    }
                else:
                    for name, p in self.model.named_parameters():
                        self.ema[name].lerp_(
                            p.detach().float(), 1 - self.cfg["ema_decay"]
                        )
        self.optimizer.zero_grad(set_to_none=True)
        return True, norm

    def payload(self, stage, config_digest, world_size):
        return {
            "origin": ORIGIN,
```

```
            "stage": stage,
            "config_digest": config_digest,
            "coordinate_contract": COORDINATE_CONTRACT.copy(),
            "world_size": world_size,
            "device_type": self.device.type,
            "model": self.model.state_dict(),
            "optimizer": self.optimizer.state_dict(),
            "ema": self.ema,
            "scheduler": self.scheduler.state_dict(),
            "step": self.step,
            "cursor": self.cursor,
            "microstep": 0,
        }

    def restore(self, payload):
        require_coordinate_contract(payload)
        if payload.get("origin") != ORIGIN or payload.get("microstep") != 0:
            raise ValueError(
                "Input is not an own-training optimizer-boundary checkpoint"
            )
        self.model.load_state_dict(payload["model"])
        self.optimizer.load_state_dict(payload["optimizer"])
        self.ema = {
            name: value.to(self.device).float()
            for name, value in payload["ema"].items()
        }
        self.scheduler.load_state_dict(payload["scheduler"])
        self.step, self.cursor = payload["step"], payload["cursor"]


def save_checkpoint(state, guard, directory, stage, digest, rank=0, world=1, pinned=()):
    stem = f"checkpoint_{state.step:08d}"
    rng_path = guard.save(
        directory / f"{stem}.rng-rank{rank}.pt", rng_state(state.device)
    )
    if dist.is_initialized():
        dist.barrier()
    path = directory / f"{stem}.pt"
    if rank == 0:
        guard.save(path, state.payload(stage, digest, world))
        checkpoints = sorted(directory.glob("checkpoint_????????.pt"))
        keep = set(checkpoints[-3:]) | {guard.path(p) for p in pinned}
        for card_path in (guard.root / "runs").rglob("run_card.json"):
            with open(card_path, encoding="utf-8") as handle:
                card = json.load(handle)
            referenced = [
                card.get("checkpoint"),
                card.get("resolved_paths", {}).get("ae_checkpoint"),
                *card.get("pinned_checkpoints", []),
            ]
            keep.update(guard.path(p) for p in referenced if p)
        for old in checkpoints:
            if old not in keep:
                guard.path(old).unlink()
                for old_rng in directory.glob(old.stem + ".rng-rank*.pt"):
                    guard.path(old_rng).unlink()
    if dist.is_initialized():
        dist.barrier()
    return path, rng_path


def load_checkpoint(state, guard, path, stage, digest, rank=0, world=1):
    path = guard.path(path)
    payload = torch.load(path, map_location="cpu", weights_only=False)
    require_coordinate_contract(payload)
    if (
        payload["stage"] != stage
        or payload["config_digest"] != digest
        or payload["world_size"] != world
        or payload["device_type"] != state.device.type
    ):
        raise ValueError(
            "Resume requires the same stage, training configuration, device type, and world size"
        )
    state.restore(payload)
    restore_rng(
        torch.load(
            guard.path(path.with_name(path.stem + f".rng-rank{rank}.pt")),
```

```
            map_location="cpu",
            weights_only=False,
        ),
        state.device,
    )


def load_frozen_autoencoder(ae, guard, path):
    payload = torch.load(guard.path(path), map_location="cpu", weights_only=False)
    require_coordinate_contract(payload)
    if (
        payload.get("origin") != ORIGIN
        or payload.get("stage") != "ae"
        or payload.get("step", 0) < 1
    ):
        raise ValueError(
            "Autoencoder input must come from a completed own-training update"
        )
    ae.load_state_dict(payload["model"])
    return ae.encoder.eval().requires_grad_(False)


class SumCommunication:
    def __init__(self, device):
        self.device, self.events = device, []


def sum_gradients(state, bucket):
    buffer = bucket.buffer()
    if buffer.dtype != torch.float32:
        raise ValueError("DDP reduction must be float32")
    pair = None
    if state.device.type == "cuda":
        pair = (
            torch.cuda.Event(enable_timing=True),
            torch.cuda.Event(enable_timing=True),
        )
        pair[0].record(torch.cuda.current_stream(state.device))
    future = dist.all_reduce(buffer, op=dist.ReduceOp.SUM, async_op=True).get_future()

    def complete(result):
        if pair is not None:
            pair[1].record(torch.cuda.current_stream(state.device))
            state.events.append(pair)
        return result.value()[0]

    return future.then(complete)


def synthetic_manifest(cfg, guard, rank):
    directory = guard.root / "data" / cfg["run_id"]
    manifest = guard.path(directory / "manifest.json")
    if rank == 0 and not manifest.exists():
        records, lower = [], 0
        for bucket_index, bucket in enumerate(cfg["buckets"]):
            complexes = []
            for i in range(cfg["synthetic_per_bucket"]):
                identifier = f"b{bucket_index}-{i}"
                seed = seed_for(cfg["run_id"], identifier)
                length = random.Random(seed).randint(
                    max(lower + 1, bucket["length"] // 2), bucket["length"]
                )
                complexes.append(synthetic_complex(seed, length, identifier))
            path = guard.atomic(
                directory / f"shard_{bucket_index}.npz",
                lambda handle: handle.write(pack_shard(complexes)),
            )
            records.append({"file": path.name, "sha256": sha256(path)})
            lower = bucket["length"]
        guard.json(manifest, {"schema": 1, "units": "angstrom", "shards": records})
    if dist.is_initialized():
        dist.barrier()
    return manifest


class LatentCache:
    def __init__(self, encoder, reader, cfg, guard, device, rank, encoder_digest):
```

```
    if (
        not cfg.get("canonical_encoder_input", False)
        or cfg.get("rotation", False)
        or cfg.get("crop", False)
    ):
        raise ValueError(
            "Latent caching requires canonical input without rotation or crop before encoding"
        )
    contract = {
        "encoder_sha256": encoder_digest,
        "manifest_sha256": sha256(reader.manifest),
        "coordinate_contract": COORDINATE_CONTRACT.copy(),
        "bf16": cfg["bf16"],
        "tf32": cfg["tf32"],
        "code_sha256": {
            name: sha256(Path(__file__).parent / name)
            for name in ("models.py", "precision.py", "data.py")
        },
    }
    identity = hashlib.sha256(
        json.dumps(contract, sort_keys=True).encode()
    ).hexdigest()
    self.directory = guard.root / "latents" / identity
    self.guard = guard
    manifest = self.directory / "manifest.json"
    if rank == 0 and not manifest.exists():
        entries = {}
        for index, identifier in enumerate(reader.ids):
            item = reader[index]
            batch = collate([item], reader.lengths[index], device)
            with torch.no_grad(), trunk_autocast(device, cfg["bf16"]):
                z = encoder(batch)[0].cpu().numpy().astype(np.float16)
            path = self.directory / (
                hashlib.sha256(identifier.encode()).hexdigest() + ".npz"
            )

            def write(handle):
                np.savez(
                    handle, latent=z, chain=item["chain"], residx=item["residx"]
                )

            guard.atomic(path, write)
            entries[identifier] = {"file": path.name, "sha256": sha256(path)}
        guard.json(
            manifest, {"contract": contract, "schema": 1, "entries": entries}
        )
    if dist.is_initialized():
        dist.barrier()
    with open(manifest, encoding="utf-8") as handle:
        record = json.load(handle)
    if record.get("schema") != 1 or record.get("contract") != contract:
        raise ValueError(
            "Latent cache contract differs from encoder, data, code, precision, or coordinates"
        )
    self.entries = record["entries"]

def get(self, batch, latent_width):
    result = torch.zeros(
        *batch["mask"].shape,
        latent_width,
        dtype=torch.float32,
        device=batch["mask"].device,
    )
    for i, identifier in enumerate(batch["complex_id"]):
        entry = self.entries[identifier]
        path = self.guard.path(self.directory / entry["file"])
        if sha256(path) != entry["sha256"]:
            raise ValueError(f"Latent cache SHA-256 mismatch: {path}")
        with np.load(path, allow_pickle=False) as archive:
            keys = {
                (int(c), int(r)): j
                for j, (c, r) in enumerate(zip(archive["chain"], archive["residx"]))
            }
            rows = [keys[(c, r)] for c, r in batch["residue_keys"][i]]
            count = len(rows)
            result[i, :count] = torch.from_numpy(
                archive["latent"][rows].astype(np.float32)
            ).to(result.device)
    return result
```

```
def preflight(cfg, guard, trainable_count, frozen_count, world):
    shard_bytes = (
        cfg["synthetic_per_bucket"]
        * sum(b["length"] for b in cfg["buckets"])
        * (111 * 4 + 37 + 1 + 2 + 4)
        if cfg["synthetic"]
        else 0
    )
    cache_bytes = int(cfg["cache_reserve_bytes"])
    largest_shard_bytes = shard_bytes
    real_residues = 0
    if not cfg["synthetic"]:
        manifest = Path(cfg["manifest"]).resolve()
        if not manifest.is_file():
            raise ValueError(f"No shard manifest present: {manifest}")
        with open(manifest, encoding="utf-8") as handle:
            records = json.load(handle).get("shards", [])
        if not records:
            raise ValueError("No shards present in manifest")
        sizes = []
        for record in records:
            shard = (manifest.parent / record["file"]).resolve()
            if not shard.is_relative_to(manifest.parent) or not shard.is_file():
                raise ValueError(f"No shard present under manifest directory: {shard}")
            with zipfile.ZipFile(shard) as archive:
                sizes.append(sum(entry.file_size for entry in archive.infolist()))
        # ShardReader keeps every shard resident in each rank, so the total is what counts.
        largest_shard_bytes = sum(sizes)
        # Upper bound from the total uncompressed bytes, without loading a shard.
        real_residues = sum(sizes) // (111 * 4 + 37 + 1 + 2 + 4) + 1
    latent_bytes = 0
    if cfg["latent_cache"]:
        if cfg["synthetic"]:
            residues = cfg["synthetic_per_bucket"] * sum(
                b["length"] for b in cfg["buckets"]
            )
        else:
            residues = real_residues
        latent_bytes = residues * (cfg["model"]["latent_width"] * 2 + 2 + 4)
    one_checkpoint = trainable_count * 16 + world * 1024 * 1024
    checkpoint_bytes = 3 * one_checkpoint
    projected = (
        shard_bytes + cache_bytes + latent_bytes + checkpoint_bytes + one_checkpoint
    )
    required = int(projected * 1.1 + 1)
    guard.mkdir(guard.root)
    free = shutil.disk_usage(guard.root).free
    if free < required:
        raise ValueError(
            f"Insufficient free space: free={free}, required={required} bytes including 10% headroom"
        )
    # Conservative resident estimate: one shard, parameters, gradients, Adam, EMA.
    host_estimate = world * (
        largest_shard_bytes + trainable_count * 20 + frozen_count * 4
    )
    if host_estimate > cfg["host_limit_bytes"]:
        raise ValueError(
            "Projected host tensor storage exceeds configured host-memory ceiling"
        )
    return {
        "projected_bytes": projected,
        "required_bytes": required,
        "free_bytes": free,
        "shards_bytes": shard_bytes,
        "cache_reserve_bytes": cache_bytes,
        "latent_bytes": latent_bytes,
        "three_checkpoints_bytes": checkpoint_bytes,
        "atomic_checkpoint_temporary_bytes": one_checkpoint,
        "resident_shard_bytes": largest_shard_bytes,
        "host_tensor_estimate_bytes": host_estimate,
    }


def host_peak_bytes():
    if os.name != "nt":
        import resource
```

```
        return int(resource.getrusage(resource.RUSAGE_SELF).ru_maxrss) * 1024
    import ctypes
    from ctypes import wintypes

    class MemoryCounters(ctypes.Structure):
        _fields_ = [("cb", wintypes.DWORD), ("faults", wintypes.DWORD)] + [
            (name, ctypes.c_size_t)
            for name in (
                "peak_working",
                "working",
                "peak_paged",
                "paged",
                "peak_nonpaged",
                "nonpaged",
                "pagefile",
                "peak_pagefile",
            )
        ]

    counters = MemoryCounters()
    counters.cb = ctypes.sizeof(counters)
    kernel = ctypes.windll.kernel32
    kernel.GetCurrentProcess.restype = ctypes.c_void_p
    api = ctypes.windll.psapi.GetProcessMemoryInfo
    api.argtypes = (ctypes.c_void_p, ctypes.POINTER(MemoryCounters), wintypes.DWORD)
    if not api(kernel.GetCurrentProcess(), ctypes.byref(counters), counters.cb):
        raise OSError("Cannot read process peak working set")
    return int(counters.peak_working)


def git_identity():
    root = Path(__file__).resolve().parents[2]
    result = {}
    for name, args in (
        ("head", ["rev-parse", "HEAD"]),
        ("working_tree", ["status", "--porcelain"]),
    ):
        try:
            process = subprocess.run(
                ["git", "-C", str(root), *args],
                capture_output=True,
                text=True,
                check=False,
            )
            result[name] = {
                "value": process.stdout.strip(),
                "returncode": process.returncode,
                "stderr": process.stderr.strip(),
            }
        except OSError as error:
            result[name] = {"error": str(error)}
    return result


def tree_hashes():
    root = Path(__file__).resolve().parent
    return {
        str(path.relative_to(root)).replace("\\", "/"): sha256(path)
        for path in sorted(root.rglob("*"))
        if path.is_file()
    }


def comparisons(left, right):
    if isinstance(left, torch.Tensor):
        if left.shape != right.shape or left.dtype != right.dtype:
            return float("inf")
        if left.numel() == 0:
            return 0.0
        return float((left.cpu().double() - right.cpu().double()).abs().max())
    if isinstance(left, dict):
        if left.keys() != right.keys():
            return float("inf")
        return max((comparisons(left[k], right[k]) for k in left), default=0.0)
    if isinstance(left, (tuple, list)):
        return (
```

```
            max((comparisons(a, b) for a, b in zip(left, right)), default=0.0)
            if len(left) == len(right)
            else float("inf")
        )
    return 0.0 if left == right else float("inf")


def percentile90(values):
    ordered = sorted(values)
    position = 0.9 * (len(ordered) - 1)
    lower = int(position)
    upper = min(lower + 1, len(ordered) - 1)
    return ordered[lower] + (ordered[upper] - ordered[lower]) * (position - lower)


def run(cfg, args):
    guard = WriteGuard(args.allowed_root)
    environment = configure_environment(guard)
    rank, world, local_rank = (
        int(os.environ.get(name, default))
        for name, default in (("RANK", "0"), ("WORLD_SIZE", "1"), ("LOCAL_RANK", "0"))
    )
    device = torch.device(f"cuda:{local_rank}" if args.device == "cuda" else "cpu")
    if device.type == "cuda":
        if not torch.cuda.is_available() or not torch.cuda.is_bf16_supported():
            raise ValueError("CUDA and BF16 support required for GPU run")
        torch.cuda.set_device(device)
    if world > 1:
        dist.init_process_group(backend=args.backend)
    if device.type == "cpu":
        torch.set_num_threads(cfg.get("cpu_threads", 1))
    initial_seed = seed_for(cfg["run_id"], "model")
    random.seed(initial_seed)
    np.random.seed(initial_seed % 2**32)
    torch.manual_seed(initial_seed)
    tf32 = set_tf32(cfg["tf32"])
    model = (
        Autoencoder(cfg["model"]) if args.stage == "ae" else FlowDenoiser(cfg["model"])
    ).float()
    counts = {args.stage: parameter_report(model)}
    encoder = None
    if args.stage == "denoiser":
        if not cfg.get("ae_checkpoint"):
            raise ValueError(
                "Denoiser requires the checkpoint produced by this autoencoder training stage"
            )
        ae = Autoencoder(cfg["model"]).float()
        encoder = load_frozen_autoencoder(ae, guard, cfg["ae_checkpoint"])
        counts["frozen_encoder"] = parameter_report(encoder)
        del ae
    preflight_record = preflight(
        cfg,
        guard,
        counts[args.stage]["total"],
        counts.get("frozen_encoder", {}).get("total", 0),
        world,
    )
    if rank == 0:
        print("PREFLIGHT " + json.dumps(preflight_record), flush=True)
    model = model.to(device).train()
    if encoder is not None:
        encoder.to(device)
    state = TrainingState(model, cfg, device)
    effective = {
        k: v
        for k, v in cfg.items()
        if k not in ("updates", "checkpoint_every", "max_wall_seconds")
    }
    digest = hashlib.sha256(json.dumps(effective, sort_keys=True).encode()).hexdigest()
    directory = guard.mkdir(
        guard.root / "runs" / (args.output_name or cfg["run_id"]) / args.stage
    )
    pinned = []
    previous_card = None
    if (directory / "run_card.json").exists():
        if not args.resume:
            raise ValueError(
                "Run directory already has evidence; use --resume or a new --output-name"
```

```
        )
    with open(directory / "run_card.json", encoding="utf-8") as handle:
        old_card = json.load(handle)
    pinned = [
        p
        for p in [
            old_card.get("checkpoint"),
            *old_card.get("pinned_checkpoints", []),
        ]
        if p
    ]
if args.resume:
    load_checkpoint(state, guard, args.resume, args.stage, digest, rank, world)
    if rank == 0 and pinned:
        previous_card = guard.json(
            directory / f"run_card_previous_{time.time_ns()}.json", old_card
        )
manifest = (
    synthetic_manifest(cfg, guard, rank)
    if cfg["synthetic"]
    else Path(cfg["manifest"])
)
reader = ShardReader(manifest)
sampler = BucketSampler(cfg["run_id"], cfg["buckets"], reader, cfg["global_batch"])
cache_started = time.perf_counter()
cache = (
    LatentCache(
        encoder,
        reader,
        cfg,
        guard,
        device,
        rank,
        sha256(guard.path(cfg["ae_checkpoint"])),
    )
    if encoder is not None and cfg["latent_cache"]
    else None
)
cache_build_seconds = (
    time.perf_counter() - cache_started if cache is not None else 0.0
)
wrapped = (
    DDP(
        model,
        device_ids=[local_rank] if device.type == "cuda" else None,
        broadcast_buffers=False,
        gradient_as_bucket_view=True,
    )
    if world > 1
    else model
)
communication = SumCommunication(device)
if world > 1:
    wrapped.register_comm_hook(communication, sum_gradients)
# Per-rank noise streams start after identical model construction or resume.
if not args.resume:
    torch.manual_seed(seed_for(cfg["run_id"], "noise", rank))
overrides = {
    "ema_start": cfg["ema_start"],
    "warmup_steps": cfg["warmup_steps"],
    "model": cfg["model"],
    "global_batch": cfg["global_batch"],
    "accumulation_by_bucket": [
        cfg["global_batch"] // (world * b["microbatch"]) for b in cfg["buckets"]
    ],
}
module_counts = {
    name or args.stage: sum(p.numel() for p in module.parameters())
    for name, module in model.named_modules()
}
if encoder is not None:
    module_counts.update(
        {
            "frozen_encoder" + ("." + name if name else ""): sum(
                p.numel() for p in module.parameters()
            )
            for name, module in encoder.named_modules()
        }
    )
```

```
    card = {
        "lock_version": 2,
        "origin": ORIGIN,
        "stage": args.stage,
        "coordinate_contract": COORDINATE_CONTRACT.copy(),
        "metric_windows": [],
        "configuration": cfg,
        "config_digest": digest,
        "environment": environment,
        "git": git_identity(),
        "code_sha256": tree_hashes(),
        "torch": torch.__version__,
        "cuda": torch.version.cuda,
        "devices": [
            {
                "name": torch.cuda.get_device_name(i),
                "memory_bytes": torch.cuda.get_device_properties(i).total_memory,
            }
            for i in range(torch.cuda.device_count())
        ]
        if device.type == "cuda"
        else [{"name": "cpu"}],
        "world_size": world,
        "tf32": tf32,
        "time_sampling": "independent uniform[0,1] per complex for x and z",
        "overrides": overrides,
        "parameter_counts": counts,
        "optimizer_policy": {
            "name": "AdamW",
            "betas": [0.9, 0.999],
            "epsilon": 1e-8,
            "weight_decay": 0.0,
            "fused": device.type == "cuda",
            "clip_norm_type": 2.0,
        },
        "module_counts": module_counts,
        "resolved_paths": {
            "allowed_root": str(guard.root),
            "run_directory": str(directory),
            "manifest": str(reader.manifest),
            "resume": str(guard.path(args.resume)) if args.resume else None,
            "ae_checkpoint": str(guard.path(cfg["ae_checkpoint"]))
            if encoder is not None
            else None,
        },
        "preflight": preflight_record,
        "data": {"manifest_sha256": sha256(reader.manifest), "complexes": len(reader)},
        "latent_cache_build_seconds": cache_build_seconds,
        "timing_note": "CUDA events measure stream work; complete wall time includes finite-decision transfers
and checkpoints. Collective duration bounds exposed communication.",
        "pinned_checkpoints": pinned,
        "previous_run_card": str(previous_card) if previous_card else None,
        "status": "running",
    }
    card_path = directory / "run_card.json"
    if rank == 0:
        guard.json(card_path, card)
        print(
            "COUNTS "
            + json.dumps({name: entry["total"] for name, entry in counts.items()}),
            flush=True,
        )
    if device.type == "cuda":
        torch.cuda.reset_peak_memory_stats(device)
    records, pending, skips = [], [], 0
    run_peak_allocated, run_peak_reserved = 0, 0
    metrics = torch.zeros(4, device=device, dtype=torch.float32)
    started = time.perf_counter()
    last_wall = started
    latest = None

    def flush():
        nonlocal last_wall
        if device.type == "cuda":
            torch.cuda.synchronize(device)
        elapsed = time.perf_counter() - last_wall
        successful = sum(not item["skip"] for item in pending)
        skipped = sum(item["skip"] for item in pending)
        for item in pending:
```

```
            events = item.pop("events")
            item["wall_amortised_ms"] = elapsed * 1000 / max(1, successful)
            if events:
                item["microstep_ms"] = [
                    a.elapsed_time(b) for a, b in events["microsteps"]
                ]
                item["optimizer_ema_ms"] = events["optimizer"][0].elapsed_time(
                    events["optimizer"][1]
                )
                item["update_device_ms"] = events["total"][0].elapsed_time(
                    events["total"][1]
                )
                item["collective_ms"] = sum(
                    a.elapsed_time(b) for a, b in events["communication"]
                )
                item["exposed_communication_upper_bound_ms"] = item["collective_ms"]
            records.append(item)
        pending.clear()
        values = metrics.clone()
        if world > 1:
            dist.all_reduce(values, op=dist.ReduceOp.SUM)
        if rank == 0:
            # Each update sums per-complex losses / global_batch across ranks; skips never enter metrics.
            means = values / max(1, successful)
            window = {
                "step": state.step,
                "loss_sum_since_log": values[0].item(),
                "x_or_reconstruction_sum": values[1].item(),
                "z_or_sequence_sum": values[2].item(),
                "wall_ms_per_update": elapsed * 1000 / max(1, successful),
                "gradient_norm": float(last_norm),
                "successful_updates": successful,
                "skipped_updates": skipped,
            }
            if args.stage == "ae":
                # Average MSE first, then take its root; this is per-coordinate RMSE, not atom RMSD.
                window.update(
                    coordinate_rmse_angstrom=(10 * means[1].sqrt()).item(),
                    sequence_cross_entropy=means[2].item(),
                    latent_penalty=means[3].item(),
                )
            else:
                window.update(x_loss=means[1].item(), z_loss=means[2].item())
            card["metric_windows"].append(window)
            guard.json(card_path, card)
            print("METRICS " + json.dumps(window), flush=True)
        host_total = torch.tensor(host_peak_bytes(), dtype=torch.int64, device=device)
        if world > 1:
            dist.all_reduce(host_total, op=dist.ReduceOp.SUM)
        if host_total.item() > cfg["host_limit_bytes"]:
            raise RuntimeError(
                "Sum of rank process peak host memory exceeds the host ceiling"
            )
        metrics.zero_()
        last_wall = time.perf_counter()

    while state.step < cfg["updates"]:
        if time.perf_counter() - started > cfg["max_wall_seconds"]:
            raise RuntimeError("Configured wall-time ceiling reached")
        forced = (
            state.cursor // cfg["updates_per_bucket"]
            if cfg.get("updates_per_bucket")
            else state.cursor % len(cfg["buckets"])
            if cfg.get("bucket_schedule") == "round_robin"
            else None
        )
        bucket_index, bucket, microsteps = sampler.draw(
            state.cursor, rank, world, forced
        )
        checkpointing = (
            cfg["model"]["checkpointing"]
            and bucket["length"] >= cfg["model"]["checkpoint_min_length"]
        )
        events = (
            {"microsteps": [], "communication": []} if device.type == "cuda" else None
        )
        cpu_update_start = time.perf_counter()
        if events is not None:
            torch.cuda.reset_peak_memory_stats(device)
```

```
    total_start = torch.cuda.Event(enable_timing=True)
    total_start.record()
start_rng = rng_state(device)
losses = torch.zeros(4, dtype=torch.float32, device=device)
for microstep, indices in enumerate(microsteps):
    batch = collate([reader[i] for i in indices], bucket["length"], device)
    if events is not None:
        before, after = (
            torch.cuda.Event(enable_timing=True),
            torch.cuda.Event(enable_timing=True),
        )
        before.record()
    sync = (
        wrapped.no_sync()
        if world > 1 and microstep < len(microsteps) - 1
        else nullcontext()
    )
    with sync:
        with trunk_autocast(device, cfg["bf16"]):
            if args.stage == "ae":
                result = wrapped(batch)
                loss1, loss2 = result["reconstruction"], result["sequence"]
            else:
                with torch.no_grad():
                    latent = (
                        cache.get(batch, cfg["model"]["latent_width"])
                        if cache is not None
                        else encoder(batch)
                    )
                x, z, tx, tz, target_x, target_z = flow_sample(
                    batch, latent, rotate=bool(cfg.get("rotation", False))
                )
                vx, vz = wrapped(x, z, tx, tz, batch)
                result = flow_loss(vx, vz, target_x, target_z, batch["mask"])
                loss1, loss2 = result["x_loss"], result["z_loss"]
        loss = result["loss"].sum() / cfg["global_batch"]
        loss.backward()
    penalty = (
        result["latent_penalty"].detach().sum() / cfg["global_batch"]
        if args.stage == "ae"
        else loss.detach().new_zeros(())
    )
    losses += torch.stack(
        (
            loss.detach(),
            loss1.detach().sum() / cfg["global_batch"],
            loss2.detach().sum() / cfg["global_batch"],
            penalty,
        )
    )
    if events is not None:
        after.record()
        events["microsteps"].append((before, after))
if events is not None:
    opt_before, opt_after = (
        torch.cuda.Event(enable_timing=True),
        torch.cuda.Event(enable_timing=True),
    )
    opt_before.record()
applied, last_norm = state.finish_update()
if events is not None:
    opt_after.record()
    end = torch.cuda.Event(enable_timing=True)
    events.update(
        {
            "optimizer": (opt_before, opt_after),
            "total": (total_start, end),
            "communication": communication.events[:],
        }
    )
    communication.events.clear()
checkpoint_ms = 0.0
if applied:
    skips = 0
    metrics += losses
    if (
        state.step % cfg["checkpoint_every"] == 0
        or state.step == cfg["updates"]
    ):
```

```
            before = time.perf_counter()
            latest, _ = save_checkpoint(
                state, guard, directory, args.stage, digest, rank, world
            )
            checkpoint_ms = (time.perf_counter() - before) * 1000
    else:
        skips += 1
        restore_rng(start_rng, device)
        if skips >= 3:
            raise RuntimeError(
                "Three consecutive non-finite updates; no training counter advanced"
            )
    if events is not None:
        # Includes host gaps, data work, and checkpoint storage before the end event.
        end.record()
    allocated = (
        torch.cuda.max_memory_allocated(device) if device.type == "cuda" else 0
    )
    reserved = (
        torch.cuda.max_memory_reserved(device) if device.type == "cuda" else 0
    )
    run_peak_allocated = max(run_peak_allocated, allocated)
    run_peak_reserved = max(run_peak_reserved, reserved)
    if reserved > cfg["gpu_limit_bytes"]:
        raise RuntimeError("GPU reserved-memory ceiling exceeded")
    pending.append(
        {
            "step": state.step,
            "bucket": bucket["length"],
            "bucket_index": bucket_index,
            "skip": not applied,
            "checkpoint_ms": checkpoint_ms,
            "peak_allocated_bytes": allocated,
            "peak_reserved_bytes": reserved,
            "update_wall_ms": (time.perf_counter() - cpu_update_start) * 1000,
            "checkpointing": checkpointing,
            "events": events,
        }
    )
    if (applied and state.step % 10 == 0) or state.step == cfg["updates"]:
        flush()
if pending:
    flush()
# Gather observations at the end, rather than synchronising every timing event.
all_records = [None] * world
if world > 1:
    dist.all_gather_object(all_records, records)
else:
    all_records = [records]
memory = {
    "rank": rank,
    "allocated_bytes": run_peak_allocated,
    "reserved_bytes": run_peak_reserved,
    "host_peak_bytes": host_peak_bytes(),
}
all_memory = [None] * world
if world > 1:
    dist.all_gather_object(all_memory, memory)
else:
    all_memory = [memory]
if rank == 0:
    summary = {}
    for bucket in cfg["buckets"]:
        items = [
            r for r in records if r["bucket"] == bucket["length"] and not r["skip"]
        ]
        measured = items[cfg["timing_warmup_per_bucket"] :]
        times = [r.get("update_device_ms", r["update_wall_ms"]) for r in measured]
        summary[str(bucket["length"])] = {
            "updates": len(items),
            "measured_updates": len(measured),
            "median_ms": statistics.median(times) if times else None,
            "p90_ms": percentile90(times) if times else None,
            "mean_ms": statistics.mean(times) if times else None,
            "peak_allocated_bytes": max(
                (r["peak_allocated_bytes"] for r in items), default=0
            ),
            "peak_reserved_bytes": max(
                (r["peak_reserved_bytes"] for r in items), default=0
```

```
                ),
                "wall_mean_ms": statistics.mean(r["update_wall_ms"] for r in measured)
                if measured
                else None,
                "status": "measured" if times else "inconclusive_after_warmup",
            }
        card.update(
            {
                "status": "finished",
                "step": state.step,
                "cursor": state.cursor,
                "rank_records": all_records,
                "peak_memory": all_memory,
                "summary": summary,
                "elapsed_seconds": time.perf_counter() - started,
                "checkpoint": str(latest)
                if latest
                else str(guard.path(args.resume))
                if args.resume
                else None,
            }
        )
        if args.compare_checkpoint:
            baseline = torch.load(
                guard.path(args.compare_checkpoint),
                map_location="cpu",
                weights_only=False,
            )
            candidate = torch.load(latest, map_location="cpu", weights_only=False)
            errors = {
                name: comparisons(baseline[name], candidate[name])
                for name in ("model", "optimizer", "ema", "scheduler", "step", "cursor")
            }
            card["resume_check"] = errors
            print("RESUME_CHECK " + json.dumps(errors), flush=True)
            if max(errors.values()) > 1e-6:
                guard.json(card_path, card)
                raise RuntimeError("Resume comparison exceeds 1e-6")
        guard.json(card_path, card)
        print(
            "SUMMARY "
            + json.dumps(
                {
                    "summary": summary,
                    "peak_memory": all_memory,
                    "elapsed_seconds": card["elapsed_seconds"],
                }
            ),
            flush=True,
        )
    if world > 1:
        dist.destroy_process_group()


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--config", required=True)
    parser.add_argument("--stage", choices=("ae", "denoiser"), required=True)
    parser.add_argument("--device", choices=("cuda", "cpu"), default="cuda")
    parser.add_argument("--backend", choices=("nccl", "gloo"), default="nccl")
    parser.add_argument("--resume")
    parser.add_argument(
        "--allowed-root",
        default=os.environ.get(
            "OMNIVERSO_ALLOWED_ROOT", os.environ.get("OMNIVERSO_ROOT")
        ),
    )
    parser.add_argument("--updates", type=int)
    parser.add_argument("--max-wall-seconds", type=int)
    parser.add_argument("--output-name")
    parser.add_argument("--ae-checkpoint")
    parser.add_argument("--pair-option", choices=("every_layer", "every_2", "none"))
    parser.add_argument("--checkpointing", choices=("on", "off"))
    parser.add_argument("--latent-cache", choices=("on", "off"))
    parser.add_argument("--ema-start", type=int)
    parser.add_argument("--compare-checkpoint")
    args = parser.parse_args()
    if not args.allowed_root:
        parser.error("--allowed-root or OMNIVERSO_ALLOWED_ROOT is required")
```

```
    if (
        args.device == "cpu"
        and args.backend == "nccl"
        and int(os.environ.get("WORLD_SIZE", 1)) > 1
    ):
        parser.error("CPU DDP requires --backend gloo")
    with open(args.config, encoding="utf-8") as handle:
        cfg = json.load(handle)
    for name in ("updates", "ae_checkpoint", "ema_start", "max_wall_seconds"):
        if getattr(args, name) is not None:
            cfg[name] = getattr(args, name)
    for name in ("pair_option", "checkpointing"):
        if getattr(args, name) is not None:
            cfg["model"][name] = (
                getattr(args, name)
                if name == "pair_option"
                else getattr(args, name) == "on"
            )
    if args.latent_cache is not None:
        cfg["latent_cache"] = args.latent_cache == "on"
    if cfg.get("updates") is None or cfg["updates"] < 1:
        parser.error(
            "Set a positive --updates value after the measured budget is available"
        )
    if not cfg["synthetic"] and not cfg.get("manifest"):
        parser.error("No shard manifest present: set manifest in the configuration")
    if cfg["checkpoint_every"] < 1:
        parser.error("checkpoint_every must be positive")
    if args.stage == "ae" and cfg["latent_cache"]:
        parser.error(
            "Latent caching is valid only for the frozen encoder in the denoiser stage"
        )
    if (
        cfg.get("updates_per_bucket")
        and cfg["updates"] > len(cfg["buckets"]) * cfg["updates_per_bucket"]
    ):
        parser.error("Requested updates exceed the per-bucket smoke schedule")
    try:
        run(cfg, args)
    except Exception as error:
        # Preserve a failure receipt without overwriting an earlier run card.
        try:
            guard = WriteGuard(args.allowed_root)
            rank = int(os.environ.get("RANK", "0"))
            guard.json(
                f"failures/{cfg['run_id']}-{time.time_ns()}-rank{rank}.json",
                {
                    "error": str(error),
                    "stage": args.stage,
                    "configuration": cfg,
                    "code_sha256": tree_hashes(),
                },
            )
        except (OSError, ValueError):
            pass
        raise


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

Reference configuration. A run is set by one JSON file. **Supplementary Table S7** gives the keys of `reference_2gpu.json` that set the model, the data and the optimiser, with their values. The files `smoke_node.json` and `smoke_local.json` share the core configuration fields; the reduced configurations add bucket-control fields. The first keeps the reference network with a short run. The second uses a network of width 128 with two blocks per stage and buckets of 12, 16 and 24 residues, for checks on a single machine. The `accumulation` entries of the buckets record the accumulation steps for two ranks; the code computes the accumulation from the global batch, the world size and the microbatch.

Key	Meaning	Value
model.ae_layers, model.denoiser_layers	Blocks per network	12, 14
model.width, model.heads	Trunk width; attention heads	768, 12
model.latent_width	Latent numbers per residue	8
model.pair_width	Pair representation width	256
model.cond_width, time_width, index_width, separation_width	Widths of the condition, time and index encodings and of the separation embedding	256, 256, 256, 128
model.pair_option	Pair update in every layer, every second layer or none	every_layer
model.checkpointing, model.checkpoint_min_length	Activation checkpointing; padded length from which it applies	true, 300
model.latent_penalty	Weight of the latent penalty	0.01
buckets	Padded length, microbatch per rank, accumulation steps	300/8/3, 400/6/4, 500/4/6
global_batch	Complexes per update	48
bf16, tf32	BF16 trunk; TF32 matrix products	true, false
learning_rate, warmup_steps	Peak rate; linear warm-up length in updates	10^{-4}, 500
clip_norm	Clip of the global gradient 2-norm	1.0
ema_decay, ema_start	EMA decay; step after which the copy is taken	0.999, 1000
latent_cache, canonical_encoder_input, rotation, crop	Latent cache switch and its preconditions	false, true, false, false
synthetic, synthetic_per_bucket	Use generated complexes; number per bucket	false, 16
manifest, ae_checkpoint	Shard manifest; autoencoder checkpoint	null, null
updates	Total successful updates (set with --updates)	null
checkpoint_every	Checkpoint interval in updates	1000
timing_warmup_per_bucket, cpu_threads	Updates per bucket excluded from timing; CPU threads	5, 1

Supplementary Table S7. Model, data and optimiser keys of the reference configuration reference_2gpu.json and their values.

Learning rates are per update; lengths are in residues.

The three complete configuration files are supplied in the code archive. The table above reports the model, data and optimisation settings required to reproduce the reference architecture.

Tests. The package ships unit tests that check parameter counts, units, padding, gradient accumulation, resume and the skip rule. The resume test compares an uninterrupted six-update run with its three-plus-three-update resumed counterpart. They run on the CPU with python

`code/tools/run_tests.py`, and the two files that carry the unit contract and the parameter counts are printed below.

`code/training/tests/test_units.py` (555 lines)

```
from test_support import (
    batch,
    config,
    flow_update,
    fresh_state,
    guard,
    relative,
    run_test,
)
from argparse import Namespace
import copy
from contextlib import redirect_stdout
import hashlib
import io
import json
import math
import os
from pathlib import Path
import time
from unittest.mock import patch

import torch

from models import (
    Autoencoder,
    FlowDenoiser,
    NM_PER_ANGSTROM,
    TRANSLATION_NOISE_NM,
    centre_atoms,
    draw_coordinate_noise,
    flow_interpolate,
    flow_sample,
)
from data import ShardReader, sha256
from precision import trunk_autocast
from train import (
    COORDINATE_CONTRACT,
    LatentCache,
    TrainingState,
    comparisons,
    configure_environment,
    load_checkpoint,
    load_frozen_autoencoder,
    run,
    save_checkpoint,
)


EXPECTED_CONTRACT = {
    "coordinate_unit": "nm",
    "nm_per_angstrom": 0.1,
    "translation_noise_nm": 0.2,
    "distance_bins": "30 centres, 1 to 30 Angstrom after conversion",
    "reconstruction_metric": "per-coordinate RMSE in Angstrom = 10*sqrt(MSE in nm^2)",
}


def check_centring():
    data = batch(count=2, padded=7)
    # Known Angstrom geometry, including nonzero missing and padded atom slots.
    data["atom37"] = (torch.arange(2 * 7 * 37 * 3, dtype=torch.float64) % 97).reshape(
        2, 7, 37, 3
    ) / 4 + 10
    mask = data["atom_mask"] & data["mask"].unsqueeze(-1)
    with trunk_autocast("cpu"):
        centred, present = centre_atoms(data)
    assert centred.dtype == torch.float32 and torch.equal(present, mask)
    assert torch.count_nonzero(centred[~mask]) == 0
    # Centre in nm independently in double precision; the implementation centres in Angstrom first.
    reference = torch.zeros_like(data["atom37"])
```

```
    for i in range(len(reference)):
        nm = data["atom37"][i][mask[i]] / 10
        reference[i][mask[i]] = nm - nm.mean(0)
    error = float((centred.double() - reference).abs().max())
    assert error <= 1e-6, error
    centroid = (centred * mask.unsqueeze(-1)).sum((1, 2)) / mask.sum((1, 2)).unsqueeze(
        -1
    )
    centroid_error = float(centroid.abs().max())
    assert centroid_error <= 1e-6, centroid_error
    scaled, _ = centre_atoms({**data, "atom37": 10 * data["atom37"]})
    torch.testing.assert_close(scaled, 10 * centred, rtol=1e-5, atol=1e-6)
    scaling_error = relative(scaled, 10 * centred)
    assert scaling_error <= 1e-5, scaling_error
    return {
        "reference_max_abs_nm": error,
        "centroid_max_abs_nm": centroid_error,
        "scaling_relative_error": scaling_error,
    }


def check_noise():
    shape = (120000, 2, 3)
    seed = 741
    with torch.random.fork_rng(devices=[]):
        torch.manual_seed(seed)
        noise, translation = draw_coordinate_noise(torch.zeros(shape))
    generator = torch.Generator().manual_seed(seed)
    independent_reference = torch.randn(shape, generator=generator)
    translation_reference = 0.2 * torch.randn(shape[0], 1, 3, generator=generator)
    torch.testing.assert_close(translation, translation_reference, rtol=0, atol=0)
    torch.testing.assert_close(
        noise, independent_reference + translation_reference, rtol=0, atol=0
    )
    assert translation.shape == (shape[0], 1, 3)
    shared = noise - independent_reference
    torch.testing.assert_close(shared, translation.expand_as(shared), rtol=0, atol=1e-6)
    torch.testing.assert_close(shared[:, 0], shared[:, 1], rtol=0, atol=1e-6)
    independent_std = float((noise - translation).std())
    translation_std = float(translation.std())
    combined_std = float(noise.std())
    assert abs(independent_std - 1.0) <= 0.01, independent_std
    assert abs(translation_std - 0.2) <= 0.003, translation_std
    assert abs(combined_std - math.sqrt(1.04)) <= 0.01, combined_std
    assert noise.dtype == translation.dtype == torch.float32
    # Exercise the caller as well: a separate unused draw helper would not suffice.
    with patch("models.draw_coordinate_noise", wraps=draw_coordinate_noise) as draw:
        flow_sample(batch(count=2), torch.zeros(2, 7, 8))
        assert draw.call_count == 1
    return {
        "complex_draws": shape[0],
        "independent_std_nm": independent_std,
        "translation_std_nm": translation_std,
        "combined_std_nm": combined_std,
    }


def check_bins():
    midpoints = torch.arange(1.5, 30.0, dtype=torch.float32)
    angstrom = torch.cat(
        (
            torch.arange(36, dtype=torch.float32),
            midpoints - 1e-4,
            midpoints,
            midpoints + 1e-4,
        )
    )
    x = torch.zeros(len(angstrom), 2, 3)
    x[:, 1, 0] = angstrom * NM_PER_ANGSTROM
    data = {
        "mask": torch.ones(len(x), 2, dtype=torch.bool),
        "residx": torch.arange(2).expand(len(x), -1),
        "chain": torch.zeros(len(x), 2, dtype=torch.long),
    }
    model = FlowDenoiser(config()["model"]).eval()
    captured = []
    bucketize = torch.bucketize
```

```
    def observe(distances, boundaries):
        result = bucketize(distances, boundaries)
        torch.testing.assert_close(
            boundaries, midpoints * NM_PER_ANGSTROM, rtol=0, atol=0
        )
        captured.append(result[:, 0, 1])
        return result

    with torch.no_grad(), patch("models.torch.bucketize", observe):
        model(
            x, torch.zeros(len(x), 2, 8), torch.zeros(len(x)), torch.ones(len(x)), data
        )
    # Independent nearest-centre reference; argmin chooses the lower centre on ties.
    reference = (angstrom[:, None] - torch.arange(1, 31)).abs().argmin(-1)
    assert len(captured) == 1
    mismatch = captured[0] != reference
    assert not mismatch.any(), {
        "angstrom": angstrom[mismatch].tolist(),
        "actual_bins": captured[0][mismatch].tolist(),
        "expected_bins": reference[mismatch].tolist(),
    }
    tie_start = 36 + len(midpoints)
    assert torch.equal(
        captured[0][tie_start : tie_start + len(midpoints)], torch.arange(29)
    )
    assert captured[0][0] == 0 and captured[0][35] == 29
    return {"distances": len(angstrom), "lower_bin_ties": len(midpoints)}


def check_path():
    data = torch.arange(24, dtype=torch.float32).reshape(4, 2, 3) / 10
    noise = -torch.ones_like(data)
    times = torch.tensor([0.0, 1.0, 0.25, 0.75])
    interpolated, velocity = flow_interpolate(data, noise, times)
    reference = torch.stack(
        [noise[i] * (1 - float(t)) + data[i] * float(t) for i, t in enumerate(times)]
    )
    torch.testing.assert_close(interpolated, reference, rtol=0, atol=1e-6)
    torch.testing.assert_close(velocity, data - noise, rtol=0, atol=1e-6)
    assert torch.equal(interpolated[0], noise[0]) and torch.equal(
        interpolated[1], data[1]
    )
    # Fixed draws test the actual sampler's endpoint, target and latent wiring.
    sample_data = batch(count=4)
    atoms, _ = centre_atoms(sample_data)
    latent = torch.arange(4 * 7 * 8, dtype=torch.float32).reshape(4, 7, 8) / 100
    coordinate_noise, latent_noise = (
        torch.full((4, 7, 3), -0.5),
        torch.ones_like(latent),
    )
    with (
        patch(
            "models.draw_coordinate_noise",
            return_value=(coordinate_noise, torch.zeros(4, 1, 3)),
        ),
        patch("models.torch.randn_like", return_value=latent_noise),
        patch("models.torch.rand", side_effect=[times, times.flip(0)]),
    ):
        x, z, tx, tz, target_x, target_z = flow_sample(sample_data, latent)
    for state, endpoint, initial, t, target in (
        (x, atoms[:, :, 1], coordinate_noise, tx, target_x),
        (z, latent, latent_noise, tz, target_z),
    ):
        keep = sample_data["mask"].unsqueeze(-1)  # padded residues carry exact zeros
        expected = (
            (1 - t[:, None, None]) * initial + t[:, None, None] * endpoint
        ) * keep
        torch.testing.assert_close(state, expected, rtol=0, atol=1e-6)
        torch.testing.assert_close(
            target, (endpoint - initial) * keep, rtol=0, atol=1e-6
        )
    return {"times": times.tolist(), "tolerance": 1e-6}


def check_reconstruction():
    data = batch(count=2)
    atoms, mask = centre_atoms(data)
    model = Autoencoder(config()["model"])
```

```
    def constant_error(ca, z, batch):
        return atoms + 0.1, torch.zeros(*ca.shape[:2], 21)

    with (
        torch.no_grad(),
        patch.object(model.decoder, "forward", constant_error),
        trunk_autocast("cpu"),
    ):
        result = model(data)
    assert (
        result["reconstruction"].dtype
        == result["coordinate_rmse_angstrom"].dtype
        == torch.float32
    )
    torch.testing.assert_close(
        result["reconstruction"], torch.full((2,), 0.01), rtol=0, atol=1e-7
    )
    torch.testing.assert_close(
        result["coordinate_rmse_angstrom"], torch.ones(2), rtol=0, atol=1e-5
    )
    atom_rmsd = (
        10
        * (
            ((result["coordinates"] - atoms).square().sum(-1) * mask).sum((1, 2))
            / mask.sum((1, 2))
        ).sqrt()
    )
    torch.testing.assert_close(
        atom_rmsd, math.sqrt(3) * result["coordinate_rmse_angstrom"], rtol=0, atol=1e-5
    )
    torch.testing.assert_close(
        atom_rmsd, torch.full((2,), math.sqrt(3)), rtol=0, atol=1e-5
    )
    return {
        "mse_nm_squared": result["reconstruction"].tolist(),
        "coordinate_rmse_angstrom": result["coordinate_rmse_angstrom"].tolist(),
        "atom_rmsd_angstrom": atom_rmsd.tolist(),
    }


def check_checkpoints(policy):
    refused = 0
    for stage in ("ae", "denoiser"):
        if stage == "ae":
            state = TrainingState(Autoencoder(config()["model"]), config(), "cpu")
            state.model(batch())["loss"].mean().backward()
            assert state.finish_update()[0]
        else:
            state = fresh_state()
            assert flow_update(state)[0]
        directory = policy.mkdir("tests/units/checkpoints/" + stage)
        path, _ = save_checkpoint(state, policy, directory, stage, "test-units")
        payload = torch.load(path, map_location="cpu", weights_only=False)
        assert payload["coordinate_contract"] == EXPECTED_CONTRACT
        for label, contract in (
            ("missing", None),
            ("wrong_unit", {**EXPECTED_CONTRACT, "coordinate_unit": "angstrom"}),
            ("wrong_noise", {**EXPECTED_CONTRACT, "translation_noise_nm": 2.0}),
        ):
            invalid = copy.deepcopy(payload)
            if contract is None:
                del invalid["coordinate_contract"]
            else:
                invalid["coordinate_contract"] = contract
            # Invalid weights prove that the contract error comes before weight loading.
            invalid["model"] = {}
            bad_path = policy.save(directory / (label + ".pt"), invalid)
            before = copy.deepcopy(state.payload(stage, "test-units", 1))
            loaders = [
                lambda: load_checkpoint(state, policy, bad_path, stage, "test-units")
            ]
            if stage == "ae":
                loaders.append(
                    lambda: load_frozen_autoencoder(state.model, policy, bad_path)
                )
            for loader in loaders:
                try:
```

```
                    loader()
                except ValueError as error:
                    assert "coordinate contract is missing or incompatible" in str(
                        error
                    ), str(error)
                    assert "retrain both stages" in str(error), str(error)
                else:
                    raise AssertionError(
                        "Incompatible coordinate checkpoint was accepted"
                    )
                assert comparisons(before, state.payload(stage, "test-units", 1)) == 0
                refused += 1
        # Valid resume and frozen-AE paths still accept current checkpoints.
        load_checkpoint(state, policy, path, stage, "test-units")
        if stage == "ae":
            load_frozen_autoencoder(state.model, policy, path)
    return {
        "refused_loads": refused,
        "valid_stage_resumes": 2,
        "valid_frozen_ae_loads": 1,
    }


def check_run_evidence(policy):
    run_id = "units-" + str(time.time_ns())
    cfg = config()
    cfg.update(
        run_id=run_id,
        updates=10,
        synthetic_per_bucket=4,
        checkpoint_every=10,
        buckets=[{"length": 6, "microbatch": 2}, {"length": 8, "microbatch": 2}],
    )
    cfg["model"]["checkpointing"] = False
    finish_update, ae_forward = TrainingState.finish_update, Autoencoder.forward
    from train import flow_loss

    output = {}
    for stage in ("ae", "denoiser"):
        cfg["latent_cache"] = stage == "denoiser"
        attempts, microsteps = [], []
        skipped = False

        def observe_result(result):
            names = (
                ("loss", "reconstruction", "sequence", "latent_penalty")
                if stage == "ae"
                else ("loss", "x_loss", "z_loss")
            )
            microsteps.append(
                {
                    name: float(result[name].detach().sum()) / cfg["global_batch"]
                    for name in names
                }
            )
            return result

        def observed_ae(model, data):
            return observe_result(ae_forward(model, data))

        def observed_flow(*args):
            return observe_result(flow_loss(*args))

        def skip_once(state):
            nonlocal skipped
            if state.step == 2 and not skipped:
                next(
                    p for p in state.model.parameters() if p.grad is not None
                ).grad.flatten()[0] = float("nan")
                skipped = True
            applied, norm = finish_update(state)
            attempts.append(
                {
                    "applied": applied,
                    "metrics": {
                        name: sum(m[name] for m in microsteps) for name in microsteps[0]
                    },
                }
```

```
    )
    microsteps.clear()
    return applied, norm

args = Namespace(
    stage=stage,
    device="cpu",
    backend="gloo",
    allowed_root=str(policy.root),
    output_name=run_id,
    resume=None,
    compare_checkpoint=None,
)
log = io.StringIO()
with (
    patch.dict(os.environ, {"RANK": "0", "WORLD_SIZE": "1", "LOCAL_RANK": "0"}),
    patch.object(TrainingState, "finish_update", skip_once),
    patch.object(Autoencoder, "forward", observed_ae),
    patch("train.flow_loss", observed_flow),
    redirect_stdout(log),
):
    run(cfg, args)
with open(
    policy.path(f"runs/{run_id}/{stage}/run_card.json"), encoding="utf-8"
) as handle:
    card = json.load(handle)
assert card["coordinate_contract"] == EXPECTED_CONTRACT
assert len(attempts) == 11 and sum(a["applied"] for a in attempts) == 10
assert sum(r["skip"] for r in card["rank_records"][0]) == 1
windows = [
    json.loads(line[len("METRICS ") :])
    for line in log.getvalue().splitlines()
    if line.startswith("METRICS ")
]
assert windows == card["metric_windows"] and len(windows) == 1
window = windows[0]
assert window["successful_updates"] == 10 and window["skipped_updates"] == 1
totals = {
    name: sum(a["metrics"][name] for a in attempts if a["applied"])
    for name in attempts[0]["metrics"]
}
expected = {"loss_sum_since_log": totals["loss"]}
if stage == "ae":
    expected.update(
        x_or_reconstruction_sum=totals["reconstruction"],
        z_or_sequence_sum=totals["sequence"],
        coordinate_rmse_angstrom=10 * math.sqrt(totals["reconstruction"] / 10),
        sequence_cross_entropy=totals["sequence"] / 10,
        latent_penalty=totals["latent_penalty"] / 10,
    )
else:
    expected.update(
        x_or_reconstruction_sum=totals["x_loss"],
        z_or_sequence_sum=totals["z_loss"],
        x_loss=totals["x_loss"] / 10,
        z_loss=totals["z_loss"] / 10,
    )
for name, value in expected.items():
    assert math.isclose(window[name], value, rel_tol=1e-5, abs_tol=1e-6), (
        stage,
        name,
        window[name],
        value,
    )
checkpoint = torch.load(
    policy.path(card["checkpoint"]), map_location="cpu", weights_only=False
)
assert checkpoint["coordinate_contract"] == EXPECTED_CONTRACT
output[stage] = {
    "successful_updates": 10,
    "skipped_updates": 1,
    "metric_keys": sorted(expected),
}
cfg["ae_checkpoint"] = (
    card["checkpoint"] if stage == "ae" else cfg["ae_checkpoint"]
)
if stage == "denoiser":
    encoder_digest = sha256(policy.path(cfg["ae_checkpoint"]))
    expected_cache = {
```

```
                "encoder_sha256": encoder_digest,
                "manifest_sha256": sha256(card["resolved_paths"]["manifest"]),
                "coordinate_contract": EXPECTED_CONTRACT,
                "bf16": cfg["bf16"],
                "tf32": cfg["tf32"],
                "code_sha256": {
                    name: sha256(Path(__file__).resolve().parents[1] / name)
                    for name in ("models.py", "precision.py", "data.py")
                },
            }
            identity = hashlib.sha256(
                json.dumps(expected_cache, sort_keys=True).encode()
            ).hexdigest()
            manifest = policy.path(f"latents/{identity}/manifest.json")
            with open(manifest, encoding="utf-8") as handle:
                record = json.load(handle)
            assert record["contract"] == expected_cache
            record["contract"]["coordinate_contract"]["coordinate_unit"] = "angstrom"
            policy.json(manifest, record)
            reader = ShardReader(card["resolved_paths"]["manifest"])
            encoder = Autoencoder(cfg["model"]).encoder.eval()
            try:
                LatentCache(encoder, reader, cfg, policy, "cpu", 0, encoder_digest)
            except ValueError as error:
                assert "Latent cache contract differs" in str(error), str(error)
            else:
                raise AssertionError(
                    "Latent cache from another coordinate unit was accepted"
                )
            output[stage]["incompatible_cache_refused"] = True
    return output


def check_padding():
    """Padded residues carry exact zeros in the sampler outputs and in the decoder outputs."""
    data = batch(count=2, padded=7)
    mask = data["mask"].clone()
    mask[:, -2:] = False
    data["mask"] = mask
    latent = torch.randn(2, 7, 8)
    x, z, tx, tz, target_x, target_z = flow_sample(data, latent)
    for name, tensor in (
        ("x", x),
        ("z", z),
        ("target_x", target_x),
        ("target_z", target_z),
    ):
        assert torch.count_nonzero(tensor[~mask]) == 0, name
        assert torch.count_nonzero(tensor[mask]) > 0, name
    model = Autoencoder(config()["model"]).eval()
    atoms, _ = centre_atoms(data)
    with torch.no_grad():
        coords, logits = model.decoder(atoms[:, :, 1], latent, data)
    assert (
        torch.count_nonzero(coords[~mask]) == 0
        and torch.count_nonzero(logits[~mask]) == 0
    )
    return {
        "padded_residues": int((~mask).sum()),
        "zero_outputs": ["x", "z", "target_x", "target_z", "coordinates", "logits"],
    }


def check():
    # Match the existing resume/DDP fixtures: the caller exports the allowed root and caches before Python.
    policy = guard()
    configure_environment(policy)
    assert NM_PER_ANGSTROM == 0.1 and TRANSLATION_NOISE_NM == 0.2
    assert COORDINATE_CONTRACT == EXPECTED_CONTRACT
    checks = {
        "centring": check_centring,
        "noise": check_noise,
        "bins": check_bins,
        "flow_path": check_path,
        "padding": check_padding,
        "reconstruction": check_reconstruction,
        "checkpoints": lambda: check_checkpoints(policy),
        "run_evidence": lambda: check_run_evidence(policy),
```

```
    }
    results, failures = {}, {}
    for name, function in checks.items():
        try:
            results[name] = function()
        except Exception as error:
            failures[name] = str(error)
    # Retain evidence for independent checks even if a midpoint criterion fails.
    assert not failures, json.dumps(
        {"failures": failures, "passed": results}, sort_keys=True
    )
    return results


if __name__ == "__main__":
    run_test("units", check)
```

code/training/tests/test_counts.py (44 lines)

```
from test_support import run_test
from pathlib import Path
import json
import torch
from models import Autoencoder, FlowDenoiser, parameter_report


def check():
    with open(
        Path(__file__).resolve().parents[1] / "configs" / "reference_2gpu.json",
        encoding="utf-8",
    ) as handle:
        cfg = json.load(handle)["model"]
    # Meta allocation counts tensors without allocating reference-size storage.
    with torch.device("meta"):
        autoencoder = Autoencoder(cfg)
        counts = {
            "encoder": parameter_report(autoencoder.encoder)["total"],
            "decoder": parameter_report(autoencoder.decoder)["total"],
            "autoencoder": parameter_report(autoencoder)["total"],
        }
        for option in ("every_layer", "every_2", "none"):
            cfg["pair_option"] = option
            counts["denoiser_" + option] = parameter_report(FlowDenoiser(cfg))["total"]
    counts["reference_difference_from_159000000"] = (
        counts["denoiser_every_layer"] - 159000000
    )
    assert counts["autoencoder"] == counts["encoder"] + counts["decoder"]
    # Regression values for the gated feed-forward design of cycle 1 (lock v2, section 3).
    expected = {
        "encoder": 127949832,
        "decoder": 127923844,
        "autoencoder": 255873676,
        "denoiser_every_layer": 153102091,
        "denoiser_every_2": 151258123,
        "denoiser_none": 149414155,
        "reference_difference_from_159000000": -5897909,
    }
    assert counts == expected, counts
    return counts


if __name__ == "__main__":
    run_test("A2", check)
```

Licence and citation. The code is released under the GNU Affero General Public License, version 3. The licence covers the code and grants no rights on any dataset, whose terms attach to the data object (main text, Materials and Methods, Data). The code loads no released weights and uses no code or weights of other generative models. Users of the code cite this article.

S5 Computational budget and derivations

The parameter counts of the base model, its compute per complex, the memory of the optimiser state and the rule that sets the number of updates are derived below. Coordinates are in nanometres inside the model and shards are in ångströms. Section S4 describes the code that these numbers come from.

Parameter counts. The counts come from building each network on the PyTorch meta device, which records shapes without allocating storage, and summing the tensor sizes. The reference configuration has width $w = 768$, 12 heads of width 64, condition width 256, latent width 8, pair width $c = 256$, 12 encoder and 12 decoder blocks and 14 denoiser blocks. A sequence block holds a modulation layer, two layer norms, a fused query-key-value map, an attention output map and a gated feed-forward pair of maps. A map from a to b numbers with a bias has $b(a + 1)$ parameters, and a layer norm has $2w$. **Supplementary Table S8** gives each group and its formula.

Tensor group	Shape	Formula	Parameters per block
Modulation, condition 256 to six vectors of 768	4608 by 256 and bias	$6w\,(256 + 1)$	1,184,256
Two layer norms	2 by (768 and 768)	$4w$	3,072
Query, key, value	2304 by 768 and bias	$3w\,(w + 1)$	1,771,776
Attention output	768 by 768 and bias	$w\,(w + 1)$	590,592
Feed-forward in (value and gate)	6144 by 768 and bias	$8w\,(w + 1)$	4,724,736
Feed-forward out	768 by 3072 and bias	$w\,(4w + 1)$	2,360,064
Block without pair path		sum	**10,634,496**
Pair bias (denoiser only)	12 by 256, no bias	$12c$	3,072
Pair update, layer norm	256 and 256	$2c$	512
Pair update, 256 to 512	512 by 256 and bias	$2c\,(c + 1)$	131,584
Pair update, 512 to 256	256 by 512 and bias	$c\,(2c + 1)$	131,328
Pair update		sum	**263,424**

Supplementary Table S8. Parameters per sequence block and per pair update.

The encoder and decoder use the block without pair path. The denoiser adds the pair bias to every block and a pair update where the pair option asks for one.

Supplementary Table S9 adds the parts outside the blocks. The input map of the encoder reads a packet of 169 numbers per residue (111 centred coordinates, 37 atom flags and 21 residue categories); the decoder and the denoiser read 11 numbers, three coordinates and eight latent values. Each module has a map from the 256-wide sinusoidal residue index to 768. The encoder and decoder carry a learned constant condition of 256 numbers, and the denoiser derives its condition from two time encodings through two maps. The pair builder exists only in the denoiser. It maps 30 distance bins to 256 numbers, embeds 258 signed separations in 128 numbers and maps these to 256, and holds a two-row chain embedding of width 256.

Part	Formula	Encoder	Decoder	Denoiser, every layer
Blocks without pair path	$n_b \times 10{,}634{,}496$	127,613,952	127,613,952	148,882,944
Pair bias	$14 \times 3{,}072$	0	0	43,008
Pair updates	$14 \times 263{,}424$	0	0	3,687,936
Input map	$768\,(n_{\text{in}} + 1)$	130,560	9,216	9,216
Residue-index map	768×257	197,376	197,376	197,376
Learned constant condition	256	256	256	0
Time maps, 512 to 256 and 256 to 256	$131{,}328 + 65{,}792$	0	0	197,120
Pair builder (bins 7,936; separation 33,024 and 33,024; chain 512)	sum	0	0	74,496
Final layer norm	$2w$	1,536	1,536	1,536
Output heads	see caption	6,152	101,508	8,459
Total		**127,949,832**	**127,923,844**	**153,102,091**

Supplementary Table S9. Parameters per module of the reference size.

The block counts n_b are 12, 12 and 14. Heads: encoder latent 768 to 8 (6,152); decoder coordinates 768 to 111 (85,359) and residues 768 to 21 (16,149); denoiser velocities 768 to 3 (2,307) and 768 to 8 (6,152). The autoencoder total is 127,949,832 plus 127,923,844, which is 255,873,676.

The pair-update option changes only the third row. The default puts an update in all 14 layers, the second option in layers 2, 4, ..., 14 and the third in none. With n_u updates the denoiser has $149{,}414{,}155 + 263{,}424\,n_u$ parameters: 153,102,091 for $n_u = 14$, 151,258,123 for $n_u = 7$ and 149,414,155 for $n_u = 0$. The pair features bias attention in every layer in all three cases. The published Proteina-Complexa denoiser has about 159 M parameters [11]; the default count is 5,897,909 below it, or 3.71 percent. The pair updates hold 3,687,936 of 153,102,091 parameters, or 2.4 percent.

Compute per complex. Only matrix products are counted, at two operations per weight (one multiplication and one addition); additions, normalisation, the softmax and the gates are not. For a complex of L residues, a denoiser of $K = 14$ blocks has the forward cost

$$F_{\mathrm{fwd}} = K[32\, w^2 L + 4\, w\, L^2 + (8c^2 + 2ch)\, L^2],$$

with $h = 12$ heads. The first term is the sequence maps: query, key and value ($3w^2$ weights), output (w^2), feed-forward in ($8w^2$) and out ($4w^2$), which is $16w^2$ weights at two operations each. The second is attention, $2L^2w$ for the scores and $2L^2w$ for the weighted sum. The third is the pair update, two maps of $2c^2$ weights each on L^2 pairs. The fourth is the projection of the pair representation to the attention bias. We estimate the matrix-product training work in checkpointed blocks as four times their forward work: one forward pass, one recomputed forward pass and a backward pass of twice the forward cost. Operations outside these blocks are not recomputed by block checkpointing. The denoiser estimates below exclude the frozen-encoder forward pass, which is evaluated for every batch in the reference configuration; its work is about 0.0715, 0.0968 and 0.1228 TFLOP at lengths 300, 400 and 500. The four terms give four-times-forward estimates of 3.006, 5.203 and 7.998 TFLOP per complex at lengths 300, 400 and 500.

Three further terms of the implemented network lie outside the formula. The pair builder applies a 30-to-256 map and a 128-to-256 map on each pair. The input, index and output maps and the time network act once per residue or once per complex. The modulation layers act once per complex and add $K \cdot 2 \cdot 256 \cdot 6w = 33{,}030{,}144$ operations per forward pass. The first two terms add 1.0 percent. Including the modulation gives four-times-forward estimates of 3.036, 5.256 and 8.079 TFLOP. The autoencoder has no pair path; its 24 blocks, input and output maps and modulation layers give the last rows of **Supplementary Table S10**. The script `code/tools/flop_derivation.py` reproduces the table except for the modulation term, which it leaves out; it therefore prints 5.255 where the table gives 5.256. The core of the denoiser calculation is

```
def denoiser_terms(L, K=14, w=768, c=256, h=12):
    return {"sequence": K * 32 * w * w * L, "attention": K * 4 * w * L * L,
            "pair_update": K * 8 * c * c * L * L, "pair_bias": K * 2 * c * h * L * L}
for L in (300, 400, 500):
    print(L, round(4 * sum(denoiser_terms(L).values()) / 1e12, 3))   # 3.006 5.203 7.998
```

Term	Formula	300	400	500
Denoiser sequence maps (GFLOP)	$32Kw^2L$	79.27	105.70	132.12
Denoiser attention (GFLOP)	$4Kw\,L^2$	3.87	6.88	10.75
Denoiser pair update (GFLOP)	$8Kc^2L^2$	660.60	1,174.41	1,835.01
Denoiser pair bias (GFLOP)	$2Kch\,L^2$	7.74	13.76	21.50

Term	Formula	300	400	500
Denoiser forward cost F_{fwd} (GFLOP)	sum of the four rows above	751.5	1,300.7	1,999.4
Denoiser four-times-forward estimate (TFLOP)	$4F_{\mathrm{fwd}}$	3.006	5.203	7.998
Pair builder (GFLOP)	$2\,(30+128)\,c\,L^2$	7.28	12.94	20.22
Input, index and output maps, time network (GFLOP)	$2w\,(11+256+3+8)\,L+2\,(512\cdot 256+256\cdot 256)$	0.13	0.17	0.21
Modulation layers (GFLOP)	$K\cdot 2\cdot 256\cdot 6w$	0.033	0.033	0.033
Denoiser four-times-forward estimate with the three rows above (TFLOP)	$4\times$ (forward cost plus the three rows)	3.036	5.256	8.079
Autoencoder forward cost (GFLOP)	$24\cdot 32\,w^2L+24\cdot 4\,w\,L^2$ plus input and output maps and modulation	143.0	193.6	245.6
Autoencoder four-times-forward estimate (TFLOP)	$4\times$ forward cost	0.572	0.774	0.982

Supplementary Table S10. Floating-point operations per complex at padded lengths 300, 400 and 500 residues. The autoencoder input and output maps use the widths 169 (encoder packet), 256 (index encoding), 8 (latent), 11 (decoder input), 256, 111 and 21 (decoder outputs).

Optimiser state. For each trainable parameter each card holds the FP32 value (4 bytes), the gradient (4 bytes) and the two Adam moments (8 bytes), which is 16 bytes. The moving average adds 4 bytes. The frozen encoder of the denoiser stage adds 4 bytes per parameter and nothing else, because it has no gradient or optimiser state. A checkpoint holds the parameters, the two moments and the average. With P_{train} trainable and P_{frozen} frozen parameters, the state on each card is $20\,P_{\mathrm{train}}+4\,P_{\mathrm{frozen}}$ bytes, and a run on R ranks that holds S bytes of shards in memory needs $R\,(S+20P_{\mathrm{train}}+4P_{\mathrm{frozen}})$ bytes of host memory. **Supplementary Table S11** gives the figures at the reference size. Activations are not included.

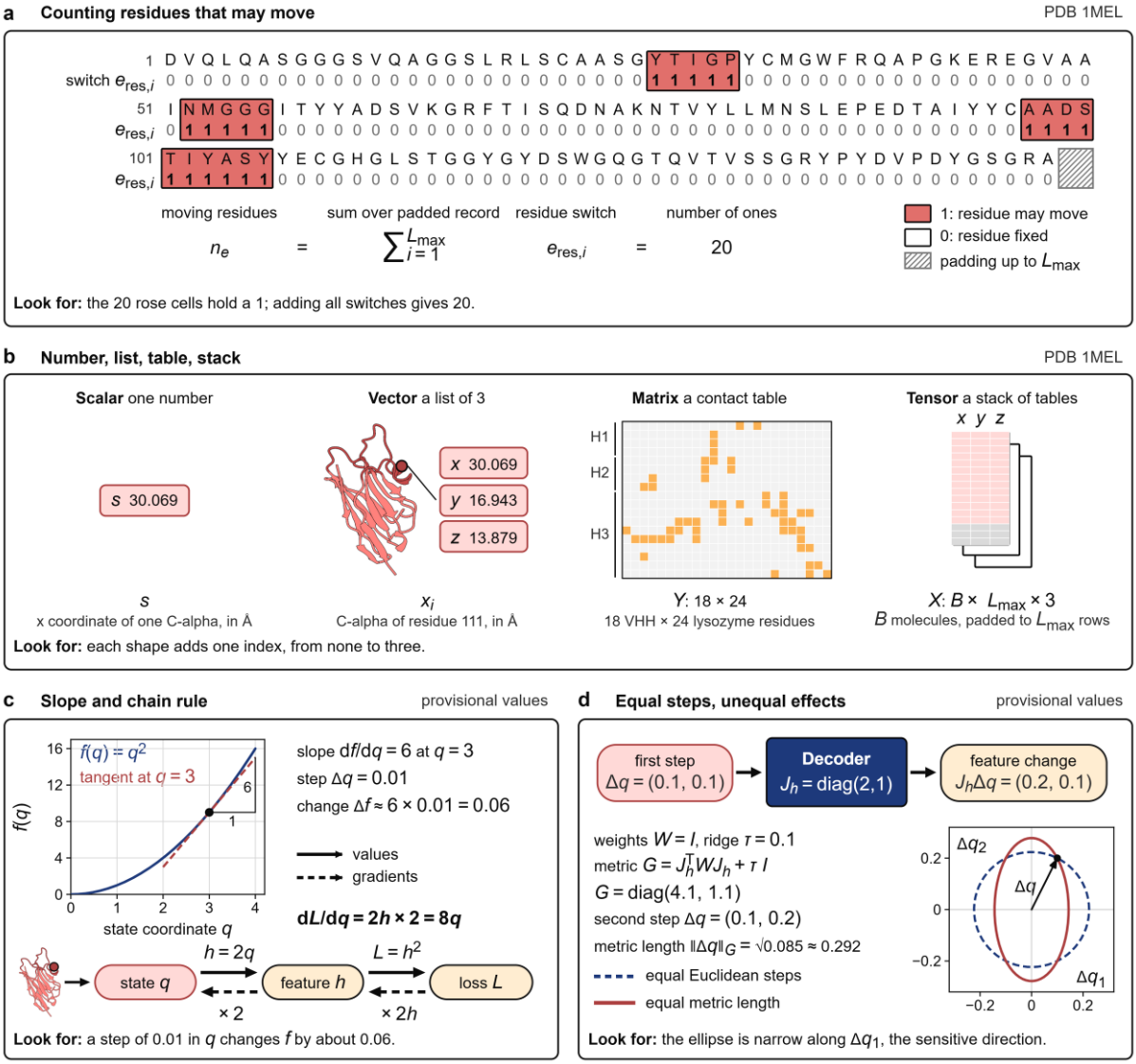
Quantity	Formula	Autoencoder stage	Denoiser stage
Trainable parameters P_{train}	**Supplementary Table S9**	255,873,676	153,102,091
Frozen parameters P_{frozen}	encoder	0	127,949,832
Weights, gradients and moments (bytes)	$16\,P_{\mathrm{train}}$	4,093,978,816	2,449,633,456
Frozen encoder (bytes)	$4\,P_{\mathrm{frozen}}$	0	511,799,328
Moving average (bytes)	$4\,P_{\mathrm{train}}$	1,023,494,704	612,408,364
State per card (bytes)	$20\,P_{\mathrm{train}}+4\,P_{\mathrm{frozen}}$	5,117,473,520	3,573,841,148

Supplementary Table S11. Memory of the optimiser state at the reference size, in bytes.

Update-count rule. For allocated stage time T_s, set $N_s = \lfloor T_s / \bar{t}_s \rfloor$, where $\bar{t}_s = \frac{1}{3}\sum_b t_{s,b} + c_s/I$. Here $t_{s,b}$ is the mean time of one complete update in length bucket b, c_s is the time to write one checkpoint and $I = 1{,}000$ is the checkpoint interval of the reference configuration. Compute each bucket mean from complete-update wall times after subtracting the corresponding `checkpoint_ms`, converting milliseconds to seconds. The supplied timing-summary means include checkpoint writes and must not be used directly. The term c_s/I then adds the amortised checkpoint cost once. N_s is calculated before launch and supplied through `--updates`. Multiplying the full forward matrix-product count, including modulation, by four gives estimates of 145.7, 252.3 and 387.8 TFLOP per update of 48 complexes at lengths 300, 400 and 500; these estimates exclude the frozen-encoder forward pass.

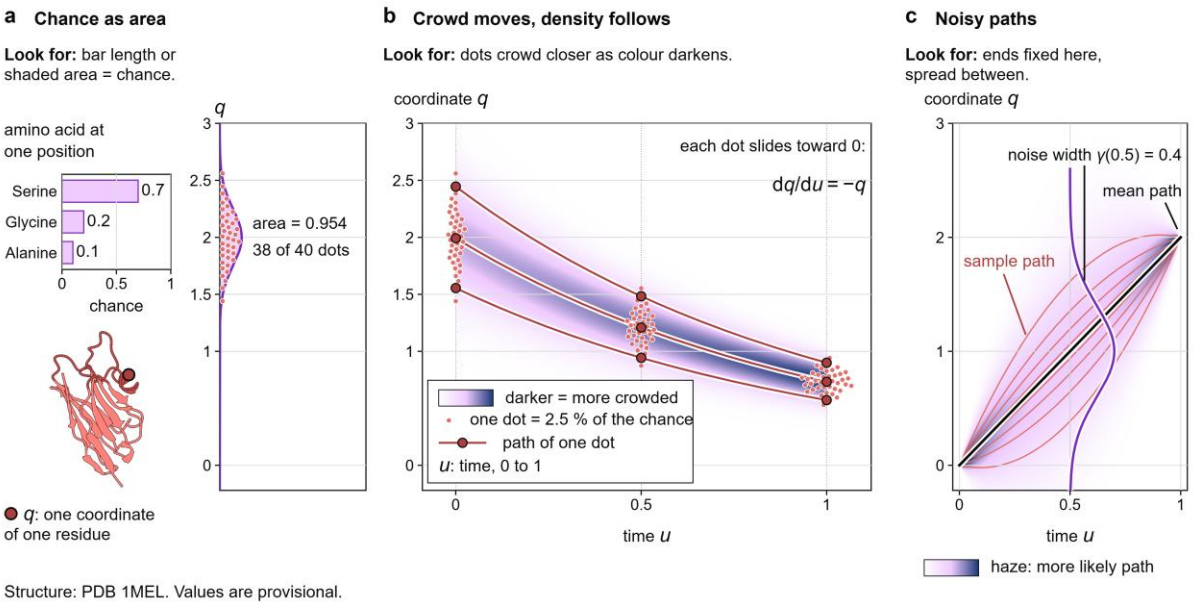
S6 Supplementary figures

The eight plates below carry the published landscape, the proposed interface layer, the mathematics and the evaluation protocol. Each caption names what is read from a deposited entry or a published source, what is a chosen example and what the full system specifies as a design.



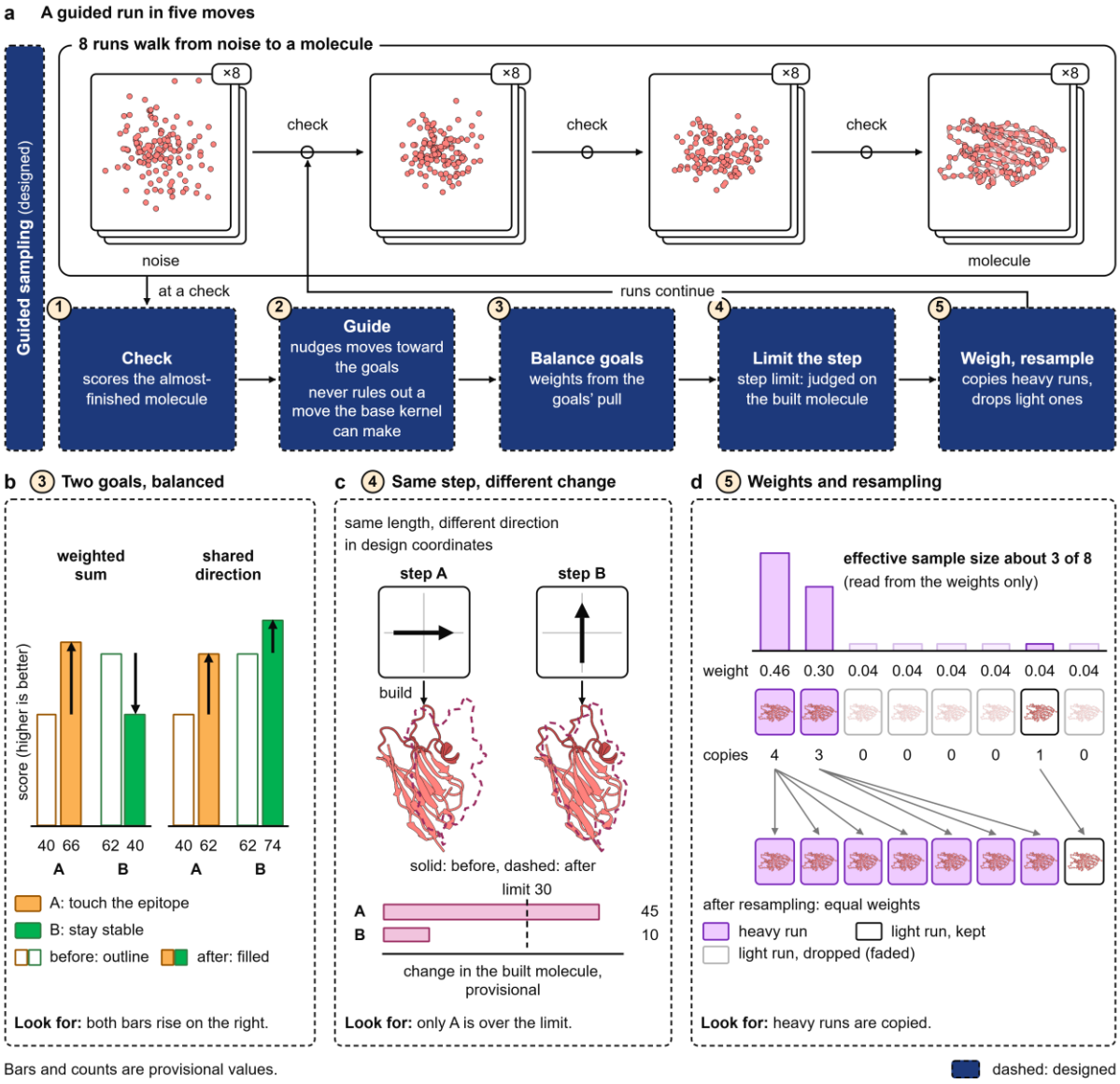
Supplementary Figure S1. Worked examples fix the notation for counts, array shapes, derivatives, decoder sensitivity and step cost.

a, In $n_e = \sum_{i=1}^{L_{\max}} e_{\mathrm{res},i} = 20$, the binary switch $e_{\mathrm{res},i}$ selects residue i, the sum runs over the padded record and n_e counts the transported residues. The strip shows the 148 residues of the VHH of PDB 1MEL chain A, with 20 chosen residues switched to 1, then two padding cells; the base model pads its records to 300, 400 or 500 positions. **b,** Four shapes of data. A scalar s is the x coordinate of one Cα, and the Cα position x_i is a vector. The contact table Y of PDB 1MEL is a matrix with entries y_{ij}, as in section 207: 18 VHH residues by 24 lysozyme residues, with 55 of 432 pairs in contact. A pair is in contact when two atoms lie closer than 4.5 Å. The padded batch of coordinates X is a tensor. The scalar and the vector are the Cα of residue 111 (30.069, 16.943, 13.879 Å). The dot on the render marks the CDR-H3 end of the VHH and is not the exact position of that atom, and the number of sheets and rows of the stack is schematic. **c,** For $f(q) = q^2$ the slope at $q = 3$ is 6, so $\Delta q = 0.01$ changes f by about 0.06. The chain rule carries a loss gradient back to the state; $h = 2q$, $L = h^2$ gives $8q$. Solid arrows, values; dashed arrows, gradients. **d,** For $J_h = \mathrm{diag}(2,1)$, the step $\Delta q = (0.1,0.1)$ gives $J_h \Delta q = (0.2,0.1)$. With $W = I$ and $\tau = 0.1$, $G = \mathrm{diag}(4.1,1.1)$. The dashed circle holds steps of equal Euclidean length, the solid ellipse steps of equal metric length; $\Delta q = (0.1,0.2)$ has metric length $\sqrt{0.085} \approx 0.292$. Panels a and b use coordinates and contacts of PDB 1MEL and a chosen set of residues, and panels c and d are worked examples with chosen numbers. PDB, Protein Data Bank; VHH, variable domain of a heavy-chain-only antibody.



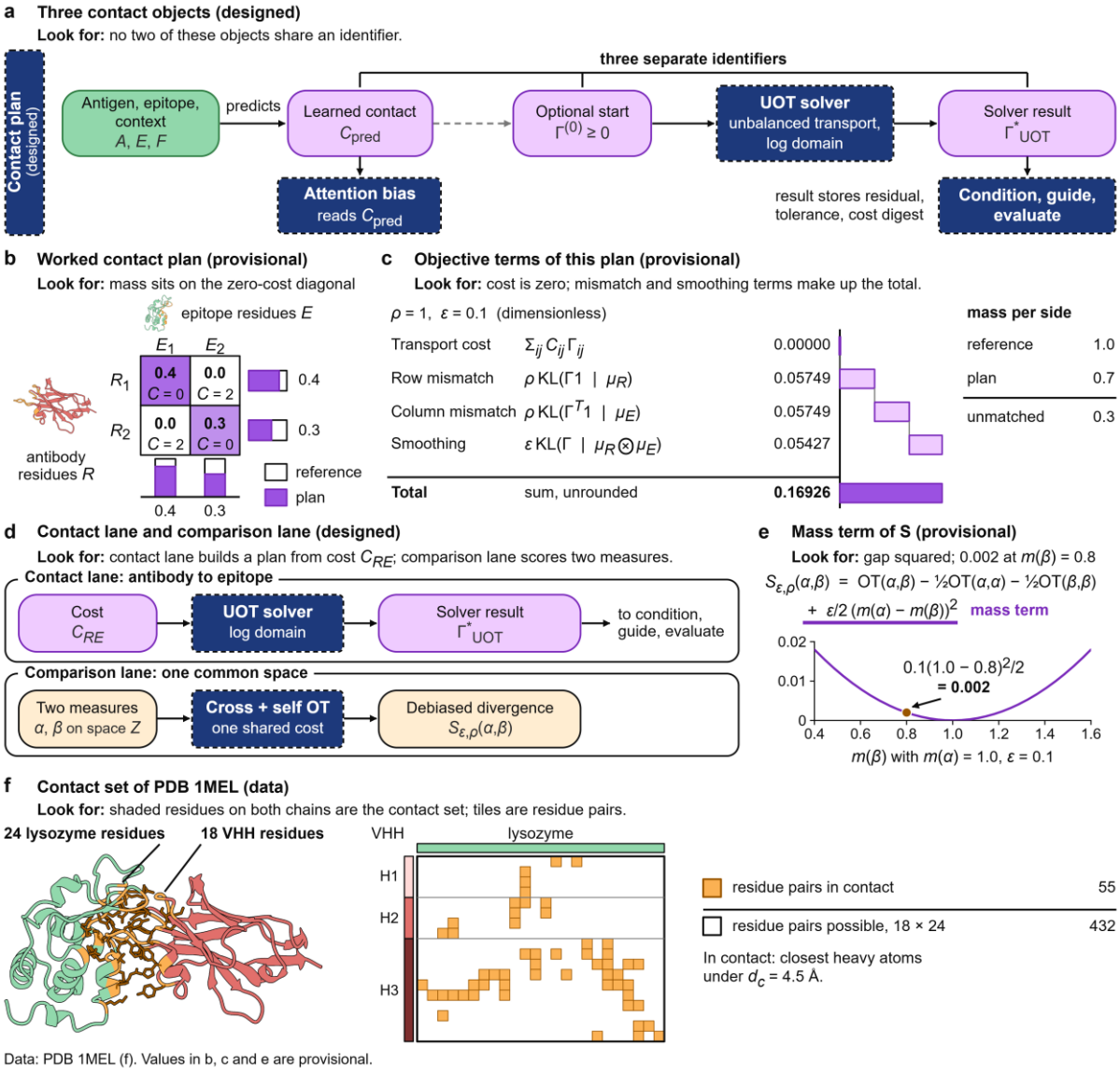
Supplementary Figure S2. Chance is a share of a crowd, a moving crowd carries its density, and noise on a path spreads the paths in the middle.

a, Left: the chances of three amino acids at one position, serine 0.7, glycine 0.2 and alanine 0.1, where bar length is the chance. Right: the density $p_0 = N(2, 0.25^2)$ of one coordinate q of one residue, with 40 dots placed at equal steps of chance, so one dot is 2.5 % of the chance. The shaded area between 1.5 and 2.5 is 0.9545, printed as 0.954, and 38 of the 40 dots lie there. The dots are quantile points and not random draws, so the count is exact and differs from the area. The render is PDB 1MEL chain A with the C-alpha of position 111 marked. It points to what one coordinate of one residue is, and no coordinate of 1MEL is plotted. **b,** The same 40 dots move by $\mathrm{d}q/\mathrm{d}u = -q$. The heat map is the density $p_u(q) = p_0(q\mathrm{e}^u)\,\mathrm{e}^u$, drawn as the square root of the density with the top of the scale at the peak of p_1, so darker means more crowded. Three dots are followed along their paths. **c,** The noise bridge $q = I(u) + \gamma(u)\zeta$ with $I(u) = 2u$ and $\gamma(u) = 1.6\,u(1-u)$. The eight sample paths use balanced draws $\zeta_k = \Phi^{-1}((k-0.5)/8)$, so their mean is the mean path $I(u)$. The haze is the Gaussian with mean $I(u)$ and width $\gamma(u)$, and the bell is that width at $u = 0.5$, where $\gamma(0.5) = 0.4$. The width is zero at both ends. One pair of end states is fixed here and only the noise varies, whereas the base state of the model is itself random. Panels b and c are two separate analytic constructions computed from the equations of Supplementary Section S1, and neither uses a learned field. The chances 0.7, 0.2 and 0.1, the density p_0, $I(u)$ and $\gamma(u)$ are example values of this plate. PDB, Protein Data Bank.



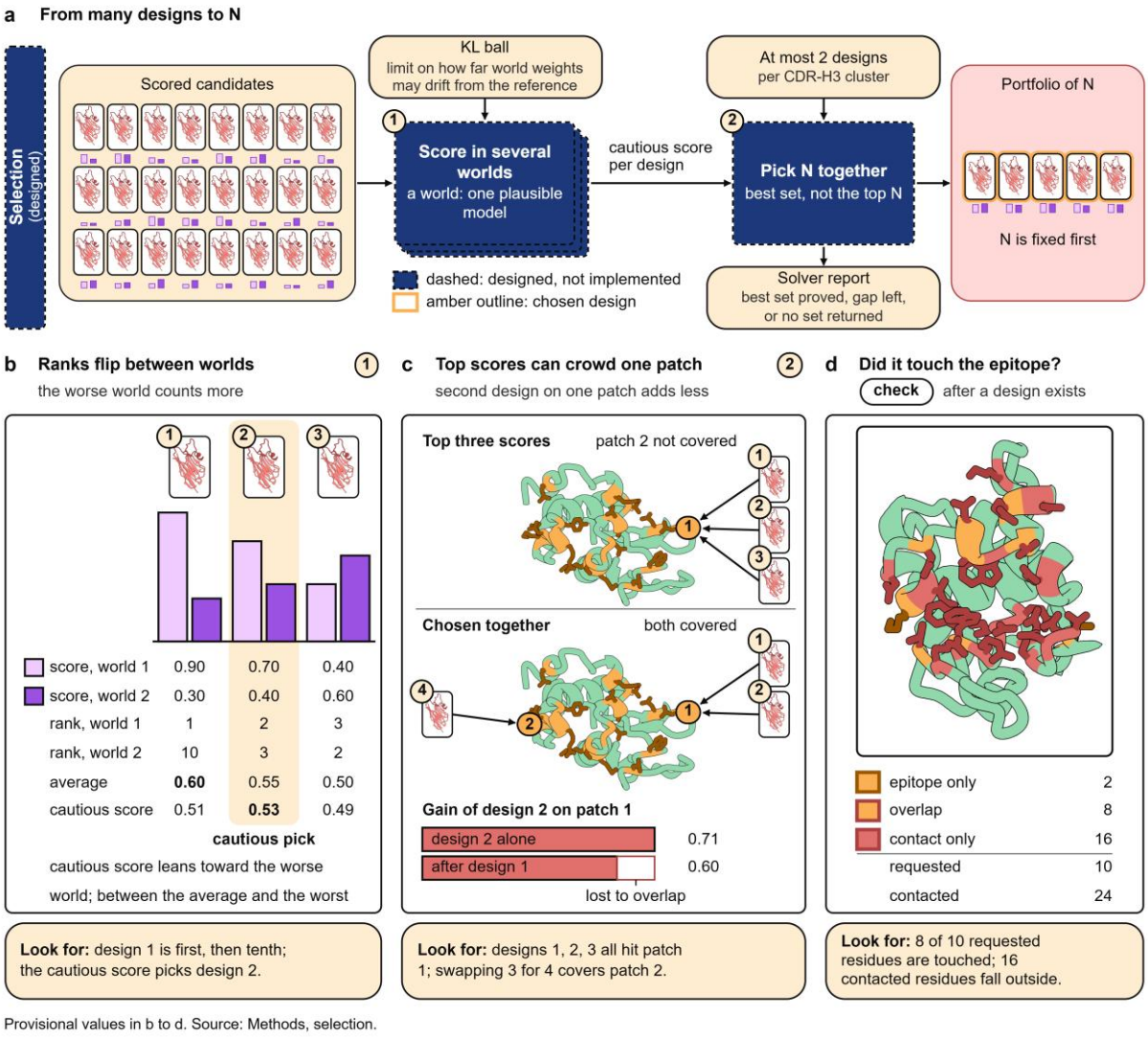
Supplementary Figure S3. Guided sampling is a run in five moves, and three of the moves are drawn as worked examples.

a, Eight runs walk from noise to a molecule, and a check interrupts them. At a check the five moves run in order. Move 1, check, scores the almost-finished molecule. Move 2, guide, nudges the move towards the goals and leaves open every move the base kernel can make. Move 3, balance goals, takes its weights from the pull of the goals. Move 4, limit the step, judges a step by its effect on the built molecule. Move 5, weigh and resample, copies heavy runs and drops light ones. The runs then continue. The pictures are frames of the straight interpolation path towards PDB 1MEL chain A, and the stacked cards mark a population of eight. **b,** Move 3, with two goals on a score scale of 0 to 100: A, touch the epitope, and B, stay stable. A fixed weighted sum raises A from 40 to 66 and lowers B from 62 to 40. The shared direction raises A from 40 to 62 and B from 62 to 74, so both goals rise. **c,** Move 4. Two steps of equal length in different directions of the design coordinates change the built molecule by 45 and 10 on an illustrative scale, against a step limit of 30, so only step A is over the limit. The solid picture is PDB 1MEL chain A before the step, and the dashed outline is its silhouette displaced in proportion to the change, which is a drawing and not a set of moved coordinates. **d,** Move 5. Eight weights, 0.46, 0.30 and six of 0.04, give an effective sample size of about 3 of 8 by $1/\sum_k (\bar{w}^{(k)})^2$, read from the weights only, which is below half of the population, so the runs are resampled. One stratified draw gives 4, 3, 0, 0, 0, 0, 1 and 0 copies, and the weights are equal afterwards. The scores, the change values, the step limit and the weights are example values of this plate, and the five moves are specified as a design. ESS, effective sample size; PDB, Protein Data Bank.



Supplementary Figure S4. Unbalanced optimal transport turns exposure and cost into a soft contact plan, and a debiased statistic compares two measures.

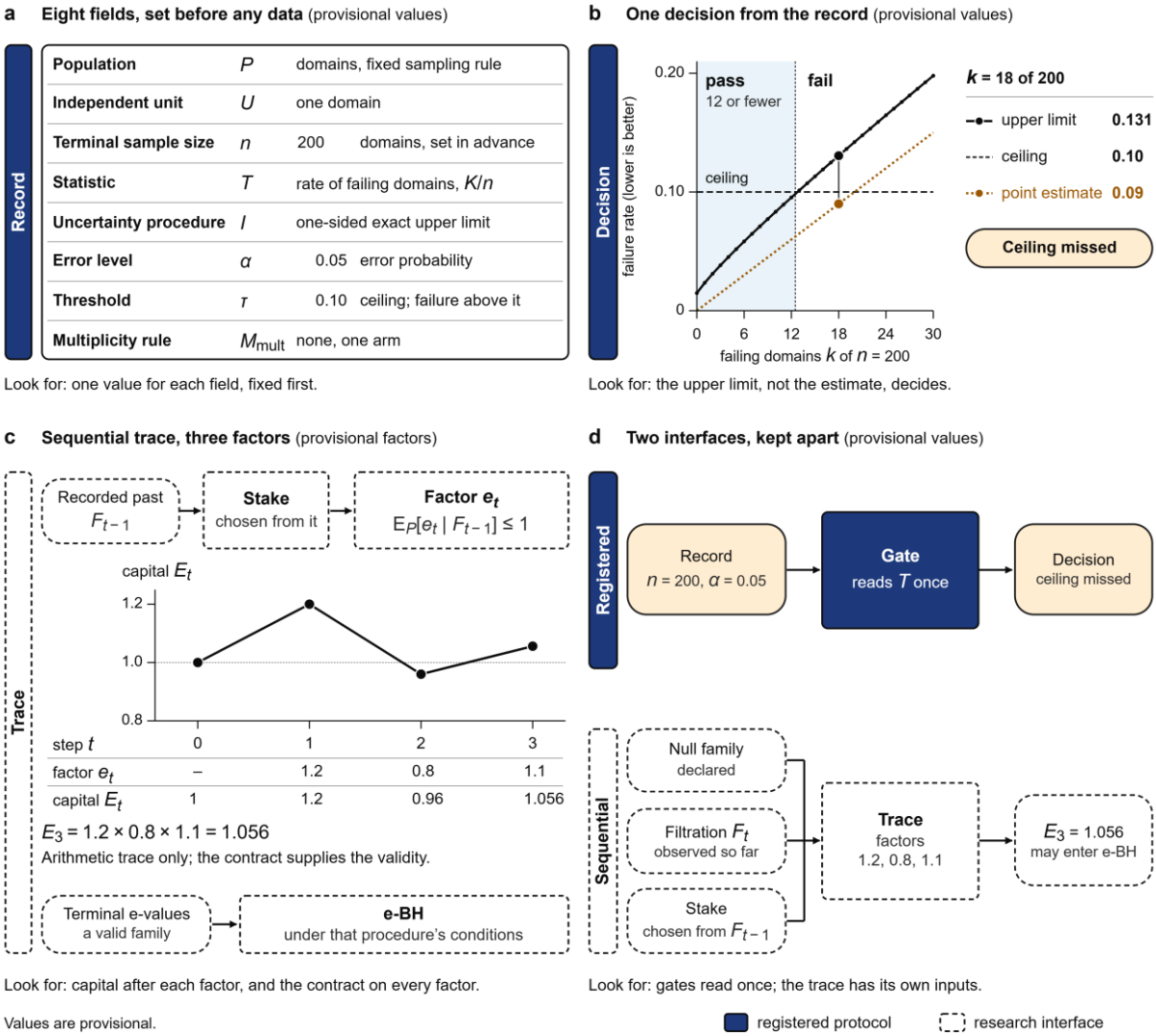
a, The learned contact C_{pred}, an optional nonnegative start $\Gamma^{(0)}$ and the solver result $\Gamma^{\star}_{\text{UOT}}$ carry separate identifiers; the solver result stores its residual, tolerance and cost digest. Dashed outlines mark components of the full system, which is specified as a design. **b,** A candidate two-by-two plan $\Gamma = \text{diag}(0.4,0.3)$ with reference masses $\mu_R = \mu_E = (0.5,0.5)$ and cost C (0 on the diagonal, 2 off it). Rows stand for antibody residues and columns for epitope residues; the bars show the row and column masses of the plan against the reference mass. The cells of b are not the pairs of f. **c,** Objective terms of this plan for $\rho = 1$ and $\varepsilon = 0.1$ (dimensionless): transport cost 0.00000, row mismatch 0.05749, column mismatch 0.05749 and smoothing 0.05427. The total, 0.16926, is computed from the unrounded terms. The reference carries mass 1.0 per side, the plan 0.7, and 0.3 is unmatched. **d,** The contact lane builds a plan from the cost C_{RE}; the comparison lane reports the debiased divergence $S_{\varepsilon,\rho}(\alpha,\beta)$ of Supplementary Section S1 for two measures in one common space. **e,** The mass term $\varepsilon/2\,(m(\alpha) - m(\beta))^2$ of $S_{\varepsilon,\rho}$ for $m(\alpha) = 1.0$ and $\varepsilon = 0.1$; at $m(\beta) = 0.8$ it equals 0.002. **f,** The interface of the deposited complex PDB 1MEL, first copy: VHH cAb-Lys3 (chain A, rose) and hen egg-white lysozyme (chain L, mint). A residue pair is in contact when the minimum heavy-atom distance is below $d_c = 4.5$ Å; shaded residues form the contact set. The set has 24 lysozyme residues and 18 VHH residues (4 in H1, 4 in H2 and 10 in H3) in 55 pairs, out of 18 × 24 = 432 possible pairs; each tile of the map is one pair. The values in b, c and e are example values, and panel f is computed from the deposited coordinates. H1 to H3, heavy-chain complementarity-determining regions 1 to 3; KL, Kullback–Leibler; OT, optimal transport; PDB, Protein Data Bank; UOT, unbalanced optimal transport; VHH, variable domain of a heavy-chain-only antibody.



Provisional values in b to d. Source: Methods, selection.

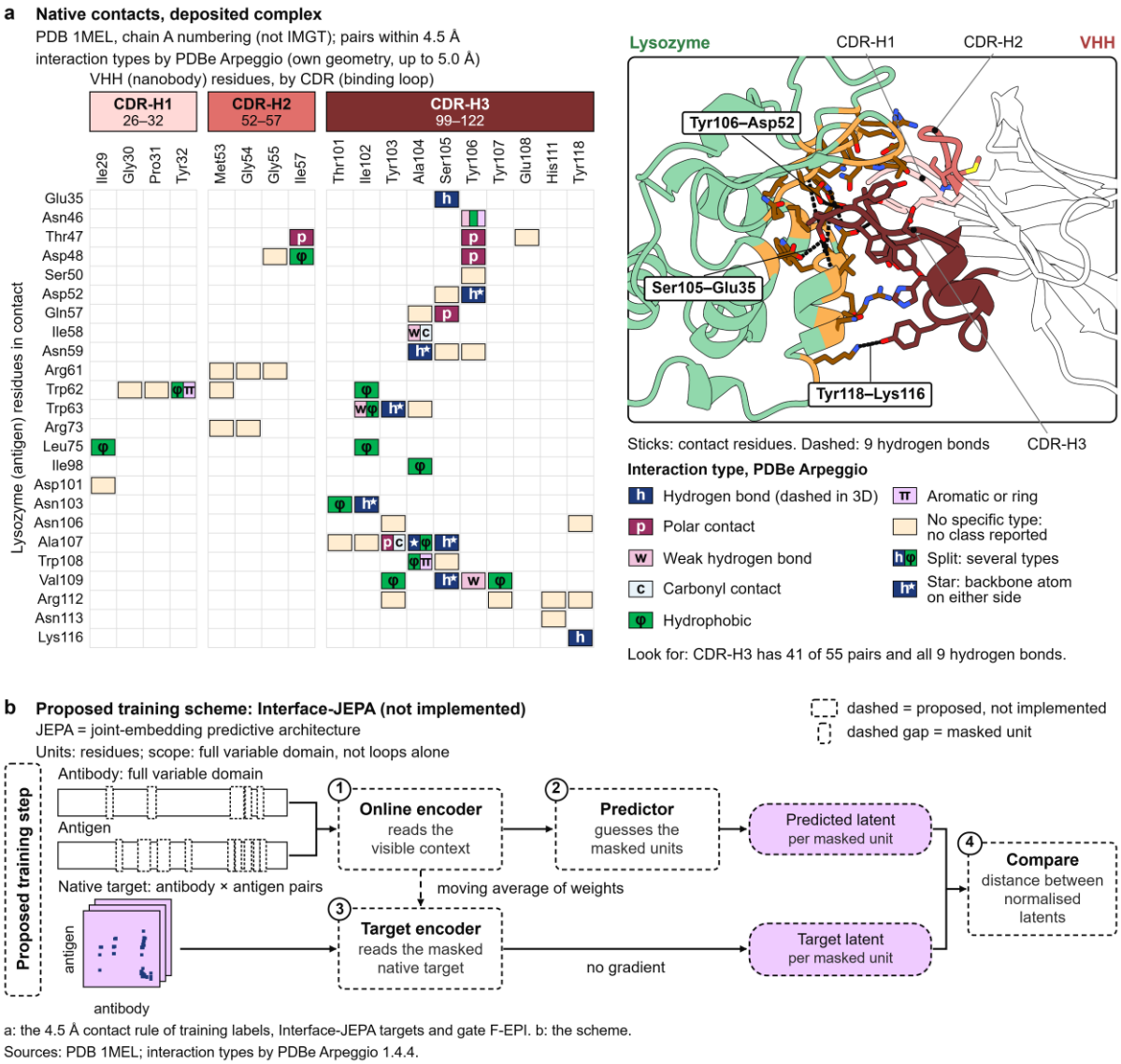
Supplementary Figure S5. Selection scores every candidate in several plausible worlds and then picks a fixed number of designs together.

a, Scored candidates enter two steps. Step 1 scores each design in several worlds under a Kullback-Leibler ball, which limits how far the world weights may drift from the reference weights, and returns one cautious score per design. Step 2 picks N designs together under the constraints of the campaign, here at most two designs per CDR-H3 cluster, and the solver reports a proved best set, a remaining gap, or no set, which is not by itself a proof of infeasibility. The portfolio holds exactly N designs, and N is fixed before scoring. The cards are the deposited VHH of PDB 1MEL chain A, repeated as a stand-in for candidates. **b,** Ranks flip between worlds. With reference weights (0.5, 0.5) and $\eta = 2$, the entropic score of three designs with world scores (0.90, 0.30), (0.70, 0.40) and (0.40, 0.60) is 0.51, 0.53 and 0.49, against plain averages of 0.60, 0.55 and 0.50. The average picks design 1 and the cautious score picks design 2, because the cautious score leans towards the worse world and lies between the average and the worst. **c,** Top scores can crowd one patch. Two chosen patches are painted on the deposited lysozyme of PDB 1MEL chain L. The three top-scoring designs all reach patch 1 and leave patch 2 uncovered; exchanging design 3 for design 4 covers both. With two targets of weight 1 and $p(A) = (0.8,0.1)$, $p(B) = (0.2,0.7)$, design 2 adds 0.71 alone and 0.60 after design 1. **d,** A check made after a design exists. On the same deposited lysozyme, 10 requested residues and the 24 residues that the native VHH contacts within 4.5 Å share 8 residues, so 2 requested residues are untouched and 16 contacted residues fall outside the request. The two patches of c and d are chosen target residues of the deposited lysozyme, and the contacted residues are those of the native VHH, which stand in for the contacts of a design. The world scores, the ranks and the chosen sets of b and c are example values of this plate. Selection is specified as the design of the full system. CDR-H3, third heavy-chain complementarity-determining region; KL, Kullback-Leibler; PDB, Protein Data Bank; VHH, variable domain of a heavy-chain-only antibody.



Supplementary Figure S6. A fixed-horizon test is fully specified before outcomes are read, and sequential evidence stays a separate research interface.

a, The eight fields of a fixed-horizon record, with the values of the chirality-rate example: one domain as the independent unit, a terminal sample size of 200, a one-sided exact upper limit at an error level of 0.05, and a ceiling of 0.10 with failure above it. **b,** One decision from that record. In the worked example, 18 failing domains among 200 give a point estimate of 0.09 and a one-sided 95% exact (Clopper–Pearson) upper limit of about 0.131, so the ceiling of 0.10 is missed. The ceiling is cleared only by 12 or fewer failures. **c,** A sequential trace. Each factor $e_t \geq 0$ obeys $\mathrm{E}_P[e_t \mid \mathcal{F}_{t-1}] \leq 1$ for every P in the declared null family, given the recorded past, and the stake is chosen from that past. The factors 1.2, 0.8 and 1.1 give capital $E_3 = 1.056$. The contract supplies the validity; the product is an arithmetic trace. A family of valid terminal e-values may enter e-BH under the conditions of that procedure. **d,** Registered gates read a fixed-horizon statistic once. Sequential traces belong to a separate interface with its own null family, filtration and stake. The limit and the capital path follow from $n = 200$, $\alpha = 0.05$ and the three factors. Solid navy marks the registered fixed-horizon protocol, which is specified as a design, and white dashed outlines mark the research interface. The numbers are values chosen for this example. Abbreviation: e-BH, e-value Benjamini–Hochberg procedure.

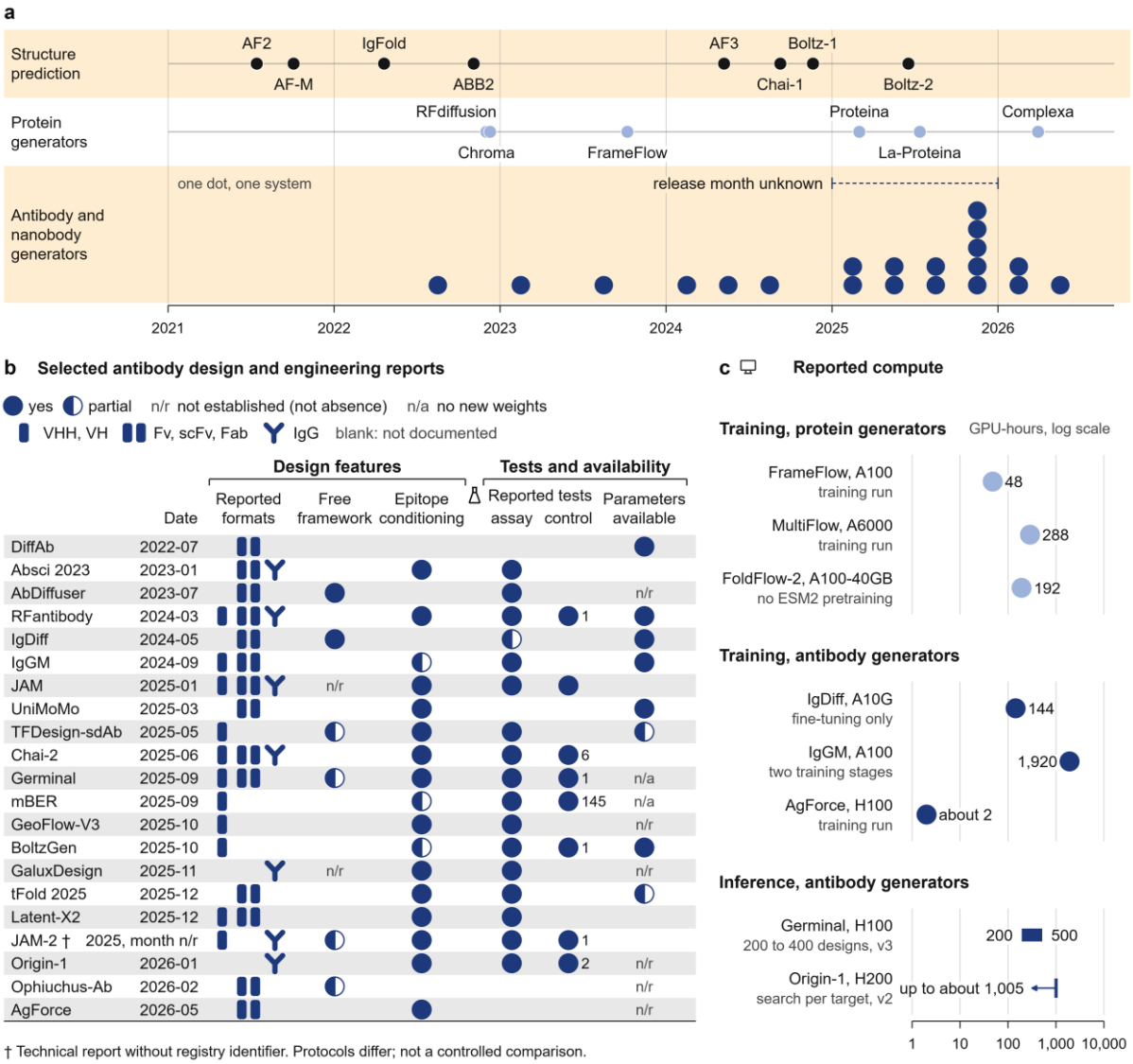


a: the 4.5 Å contact rule of training labels, Interface-JEPA targets and gate F-EPI. b: the scheme.
Sources: PDB 1MEL; interaction types by PDBe Arpeggio 1.4.4.

Supplementary Figure S7. The contacts of one deposited nanobody-lysozyme complex are typed by residue pair, and a predictive layer is specified to learn from masked interface units.

a, Native contacts of PDB 1MEL, chains A and L, first copy. Columns are the 18 VHH residues with a heavy atom within 4.5 Å of lysozyme, grouped under the loop spans H1 26 to 32, H2 52 to 57 and H3 99 to 122 by chain-A sequence index. These spans are wider than the IMGT windows of the immunoglobulin contract. Rows are the 24 lysozyme residues that they touch, and each of the 55 tiles is one residue pair. The colour and the letter give the interaction class that PDBe Arpeggio 1.4.4 reports at up to 5.0 Å, on a model whose hydrogens were placed by ChimeraX. The classes are h hydrogen bond, p polar contact, w weak hydrogen bond, c carbonyl contact, φ hydrophobic and π aromatic or ring, and the polar class holds both the polar and the weak-polar records of the tool. A cream tile is a pair within 4.5 Å for which the tool reports no class, a split tile carries several classes, and a star marks a hydrogen bond with a backbone nitrogen or oxygen on either side, 7 of the 9 pairs. CDR-H3 holds 41 of the 55 pairs and all 9 hydrogen bonds. The close-up is the same complex with the contact residues as sticks and the nine hydrogen bonds as dashed lines, several of which overlap in this projection. **b,** The training scheme of Interface-JEPA as specified for the full system. An online encoder reads the visible antibody and antigen residues over the full variable domain, and a predictor guesses a latent for each masked unit. A target encoder, an exponential moving average of the online encoder, reads the masked native target and passes no gradient. The target is a tile of antibody against antigen residues, drawn here as the contact indicator of 1MEL; its distance, bin and orientation channels are not drawn. The loss is the distance between the normalised latents of the masked units. Panel a is a measurement on a deposited structure with a published classifier, and the interaction classes are that tool's annotation; the 4.5 Å contact rule is the one the training labels, the Interface-JEPA target and gate F-EPI share. In b the masked units are placed at the interface residues of 1MEL by this drawing, and the Methods

set no masking pattern. IMGT, ImMunoGeneTics numbering; JEPA, joint-embedding predictive architecture; PDB, Protein Data Bank; VHH, variable domain of a heavy-chain-only antibody.



Supplementary Figure S8. Selected public reports of antibody and nanobody design tools are dated and described by what each one reports, and the figure ranks none of them.

a, The date of the assessed report for 35 systems in three families, on one time axis. In the antibody and nanobody lane one dot is one system, placed in the middle of its calendar quarter. JAM-2† is dated to the year alone, so it is drawn as a dashed span without a dot. **b,** Twenty-one selected antibody design and engineering reports, in order of date. The date is the one on which the assessed report first became public as a preprint, a technical report, a release or a journal article (Boltz-1 and Boltz-2 are dated by their preprints and not by their earlier code releases), and capabilities are assessed across the versions listed in the figure tables. Reported formats include experimental reformatting and do not imply that a system generates constant domains. A filled dot is yes and a half dot is partial. Partial means, for the framework, task-dependent, biased towards the input framework, or a limited number of substitutions in an existing single-domain antibody; for the epitope, optional or one hotspot residue; for parameters, one component only; for the assay, expression only. Text n/r means not established from the reviewed sources and is not evidence of absence, n/a means that no new model weights were produced, and a blank cell means not documented in the cited study. Under reported tests, the assay column marks a laboratory assay described in the report. The control column marks a specificity control, a single decoy, an off-target panel or a pooled cross-screen, with the number of proteins or targets where the source gives it, and those numbers count different things; the single decoy of RFantibody covers one of its six targets, TcdB. The categories identify the assay types reported and not the quality of the evidence or the specificity achieved, and no report states a self-proteome panel. **c,** Published compute in GPU-hours on a logarithmic axis, for the stage named under each label. Hardware-hours are not normalised measures of efficiency, and the Origin-1 value is a search budget read from a plot axis. Sixteen of

the 21 entries carry no extracted GPU-hour estimate from the selected source versions. Every mark and value is source-reported, except that the totals of IgDiff (fine-tuning only), IgGM and AgForce are calculated here from the reported GPU count and duration, and nothing was re-measured here. Dates and assessed versions are distinguished, missing information is not evidence of absence, and the published protocols differ, so this is not a controlled comparison. †, technical report without registry identifier. Fab, antigen-binding fragment; Fv, variable fragment; IgG, immunoglobulin G; n/a, not applicable; n/r, not established; scFv, single-chain Fv; VH, heavy-chain variable domain; VHH, variable domain of a heavy-chain-only antibody.

References

[1] Anderson BDO. Reverse-time diffusion equation models. Stochastic Processes and their Applications. 1982;12(3):313–326. doi:10.1016/0304-4149(82)90051-5

[2] Séjourné T, Feydy J, Vialard FX, Trouvé A, Peyré G. Sinkhorn Divergences for Unbalanced Optimal Transport. arXiv:1910.12958. 2019.

[3] Gerber M, Chopin N, Whiteley N. Negative association, ordering and convergence of resampling methods. The Annals of Statistics. 2019;47(4):2236–2260. doi:10.1214/18-aos1746

[4] Kuhn D, Esfahani PM, Nguyen VA, Shafieezadeh-Abadeh S. Wasserstein Distributionally Robust Optimization: Theory and Applications in Machine Learning. arXiv:1908.08729. 2019.

[5] Nemhauser GL, Wolsey LA, Fisher ML. An analysis of approximations for maximizing submodular set functions—I. Mathematical Programming. 1978;14(1):265–294. doi:10.1007/bf01588971

[6] Angelopoulos AN, Bates S, Fisch A, Lei L, Schuster T. Conformal Risk Control. arXiv:2208.02814. 2022.

[7] Ramdas A, Grünwald P, Vovk V, Shafer G. Game-Theoretic Statistics and Safe Anytime-Valid Inference. Statistical Science. 2023;38(4):576–601. doi:10.1214/23-sts894

[8] Berman HM. The Protein Data Bank. Nucleic Acids Research. 2000;28(1):235–242. doi:10.1093/nar/28.1.235

[9] Varadi M, Anyango S, Deshpande M, Nair S, Natassia C, Yordanova G, et al. AlphaFold Protein Structure Database: massively expanding the structural coverage of protein-sequence space with high-accuracy models. Nucleic Acids Research. 2022;50(D1):D439–D444. doi:10.1093/nar/gkab1061

[10] Lau AM, Bordin N, Kandathil SM, Sillitoe I, Waman VP, Wells J, et al. Exploring structural diversity across the protein universe with The Encyclopedia of Domains. Science. 2024;386:eadq4946. doi:10.1126/science.adq4946

[11] Didi K, Zhang Z, Zhou G, Reidenbach D, Cao Z, Cha S, et al. Scaling Atomistic Protein Binder Design with Generative Pretraining and Test-Time Compute. arXiv:2603.27950. 2026.

[12] Dunbar J, Krawczyk K, Leem J, Baker T, Fuchs A, Georges G, et al. SAbDab: the structural antibody database. Nucleic Acids Research. 2014;42(D1):D1140–D1146. doi:10.1093/nar/gkt1043

[13] Capel HL, Vavourakis O, Williams BH, Taylor CR, Deane CM. SAbDab2: The structural antibody database in the age of machine learning. bioRxiv. 2026. doi:10.64898/2026.06.16.732554

[14] Gupta A, Rivero BM, Li R, Roel-Touris J, Nanfack YF, Wendt M, et al. SNAC-DB: An ML-ready database for antibody and NANOBODY® VHH–antigen complexes with expanded

structural diversity and real-world benchmarking. Protein Science. 2026;35(7):e70655. doi:10.1002/pro.70655

[15] Ferdous S, Martin ACR. AbDb: antibody structure database—a database of PDB-derived antibody structures. Database. 2018;2018. doi:10.1093/database/bay040

[16] Ahmed M, Taj N, Khan IU, Venkateswara H, Patterson M. CHIMERA-Bench: A Benchmark Dataset for Epitope-Specific Antibody Design. arXiv:2603.13431. 2026.

[17] Xu S, Feng Q, Qiao L, Wu H, Shen T, Cheng Y, et al. Benchmarking all-atom biomolecular structure prediction with FoldBench. Nature Communications. 2025;17(1):442. doi:10.1038/s41467-025-67127-3

[18] Smorodina E, Ali M, Kropivšek Brumat K, Salicari L, Miklavc S, Kappassov A, et al. Structural Plausibility Without Binding Specificity: Limits of AI-Based Antibody-Antigen Structure Prediction Confidence Scores. bioRxiv. 2026. doi:10.64898/2026.03.02.709004

[19] Hitawala FN, Gray JJ. What does AlphaFold3 learn about antibody and nanobody docking, and what remains unsolved? mAbs. 2025;17:2545601. doi:10.1080/19420862.2025.2545601

[20] Olsen TH, Boyles F, Deane CM. Observed Antibody Space: A diverse database of cleaned, annotated, and translated unpaired and paired antibody sequences. Protein Science. 2022;31(1):141–146. doi:10.1002/pro.4205

[21] Raybould MIJ, Marks C, Lewis AP, Shi J, Bujotzek A, Taddese B, et al. Thera-SAbDab: the Therapeutic Structural Antibody Database. Nucleic Acids Research. 2020;48(D1):D383–D388. doi:10.1093/nar/gkz827

[22] Abanades B, Olsen TH, Raybould MIJ, Aguilar-Sanjuan B, Wong WK, Georges G, et al. The Patent and Literature Antibody Database (PLAbDab): an evolving reference set of functionally diverse, literature-annotated antibody sequences and structures. Nucleic Acids Research. 2024;52(D1):D545–D551. doi:10.1093/nar/gkad1056

[23] Jankauskaitė J, Jiménez-García B, Dapkūnas J, Fernández-Recio J, Moal IH. SKEMPI 2.0: an updated benchmark of changes in protein–protein binding energy, kinetics and thermodynamics upon mutation. Bioinformatics. 2019;35(3):462–469. doi:10.1093/bioinformatics/bty635

[24] Sirin S, Apgar JR, Bennett EM, Keating AE. AB-Bind: Antibody binding mutational database for computational affinity predictions. Protein Science. 2016;25(2):393–409. doi:10.1002/pro.2829

[25] Zhao X, Tang YC, Singh A, Cantu VJ, An K, Lee J, et al. AbBiBench: A Benchmark for Antibody Binding Affinity Maturation and Design. arXiv:2506.04235. 2025.

[26] Robert PA, Akbar R, Frank R, Pavlović M, Widrich M, Snapkov I, et al. Unconstrained generation of synthetic antibody–antigen structures to guide machine learning methodology for antibody specificity prediction. Nature Computational Science. 2022;2(12):845–865. doi:10.1038/s43588-022-00372-4

[27] Raybould MIJ, Marks C, Krawczyk K, Taddese B, Nowak J, Lewis AP, et al. Five computational developability guidelines for therapeutic antibody profiling. Proceedings of the National Academy of Sciences. 2019;116(10):4025–4030. doi:10.1073/pnas.1810576116

[28] Jain T, Sun T, Durand S, Hall A, Houston NR, Nett JH, et al. Biophysical properties of the clinical-stage antibody landscape. Proceedings of the National Academy of Sciences. 2017;114(5):944–949. doi:10.1073/pnas.1616408114

[29] Harvey EP, Shin JE, Skiba MA, Nemeth GR, Hurley JD, Wellner A, et al. An in silico method to assess antibody fragment polyreactivity. Nature Communications. 2022;13(1):7554. doi:10.1038/s41467-022-35276-4

[30] Prihoda D, Maamary J, Waight A, Juan V, Fayadat-Dilman L, Svozil D, et al. BioPhi: A platform for antibody design, humanization, and humanness evaluation based on natural antibody repertoires and deep learning. mAbs. 2022;14(1):2020203. doi:10.1080/19420862.2021.2020203

[31] Chai Discovery Team, Boitreaud J, Dent J, Geisz D, McPartlon M, Meier J, et al. Zero-shot antibody design in a 24-well plate. bioRxiv. 2025. doi:10.1101/2025.07.05.663018

[32] Bennett NR, Watson JL, Ragotte RJ, Borst AJ, See DL, Weidle C, et al. Atomically accurate de novo design of antibodies with RFdiffusion. Nature. 2026;649(8095):183–193. doi:10.1038/s41586-025-09721-5

[33] Mille-Fragoso LS, Driscoll CL, Wang JN, Dai H, Widatalla T, Zhang JL, et al. Efficient generation of epitope-targeted antibodies with Germinal. Nature Biotechnology. 2026. doi:10.1038/s41587-026-03187-0

[34] Stark H, Faltings F, Choi M, Xie Y, Hur E, O'Donnell T, et al. BoltzGen: Toward Universal Binder Design. bioRxiv. 2025. doi:10.1101/2025.11.20.689494

[35] Swanson E, Nichols M, Ravichandran S, Ogden P. mBER: Controllable de novo antibody design with million-scale experimental screening. bioRxiv. 2025. doi:10.1101/2025.09.26.678877

[36] Clopper CJ, Pearson ES. The use of confidence or fiducial limits illustrated in the case of the binomial. Biometrika. 1934;26(4):404–413. doi:10.1093/biomet/26.4.404

[37] Wang R, Ramdas A. False Discovery Rate Control with E-values. Journal of the Royal Statistical Society Series B: Statistical Methodology. 2022;84(3):822–852. doi:10.1111/rssb.12489

[38] Chen T, Xu B, Zhang C, Guestrin C. Training Deep Nets with Sublinear Memory Cost. arXiv:1604.06174. 2016.

[39] Li S, Zhao Y, Varma R, Salpekar O, Noordhuis P, Li T, et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. arXiv:2006.15704. 2020.