Arc Science
Technical Report I

Omniverso: Substrate, Data Pipeline, Base Model, and Training Code

Part I of the Omniverso Specification (Background, System Architecture, Data Curation, Base Model, and Supplementary Records S2, S4, and S5)

Danil Kotelnikov

Abstract. Omniverso is a system for designing VHH and Fv binders for a chosen antigen and epitope. This volume covers the biological background, system layout, coordinate geometry, immunoglobulin contract, and anti-leakage data pipeline. It specifies the partially latent two-clock base model: a 255,873,676-parameter autoencoder and a 153,102,091-parameter denoiser, configured for training from random initialisation with no released weights loaded. Supplementary Sections S2, S4, and S5 provide data tables, base-model training code, and compute derivations.

Contents

1 Antibodies, nanobodies and the design problem **4**

2 Computational antibody design before generative models **6**

2.1 Structure prediction 7

3 Generative models of protein structure **8**

3.1 Why partially latent flow matching suits antibody-antigen design 11

4 Published antibody and nanobody generators **11**

4.1 Protocols behind the reported rates 16

5 Evaluation, specificity and developability **17**

5.1 Specificity, developability and immunogenicity 19

6 Gaps, aims and scope **20**

7 System overview **21**

8 State and geometry **24**

8.1 Rigid motions and coordinate gauges 25

9 Immunoglobulin contract **27**

9.1 Numbering, regions and the CDR3 prior 28

9.2 Scaffold profiles, cysteines and pairing 29

10 Data **30**

10.1 Sources and licences 32

10.2 Curation, quality control and deduplication 33

10.3 Leakage axes, temporal holdout and split freezing 34

10.4 Shard format and sampling 36

11 Generative model **37**

11.1 Probability paths and the training target 37

11.2 Block loss and independent block clocks 39

11.3 Autoencoder and decoder factorisation 40

11.4 Architecture and parameter counts 41

12 Training of the base model **44**

13 Computing, software and availability **47**

14 Data tables **48**

15 Code **53**

16 Computational budget and derivations **96**

17 Scope, Ethics, Licence, and Declarations **101**

17.1 Scope, Dual Use, and Licensing 101

17.2 Declarations 101

1. Antibodies, nanobodies and the design problem

Antibodies are the largest class of biotherapeutics [1]. Each one binds its target through a few loops on its variable domains. A design that aims at a chosen patch of a chosen antigen has to produce those loops, the scaffold that holds them, and a molecule that a cell can make and a patient can tolerate.

Variable-domain architecture. A conventional antibody variable domain folds from antiparallel β-strands. They form four framework regions (FR1 to FR4) and carry three hypervariable loops, the complementarity-determining regions CDR1 to CDR3 [2]. A heavy-chain domain (VH) and a light-chain domain (VL) pair to form the variable fragment (Fv), so its binding site has six loops [3]. The loops are the main route to the antigen, and other residues also take part. Loop definitions from Kabat, Chothia or the international ImMunoGeneTics information system (IMGT) may miss about 20 percent of the antigen-binding residues, and framework residues contact the antigen in many complexes [3]. In the IMGT analysis, 15 of 19 canonical loop shapes of CDR1 and CDR2 correspond to a single loop length. The same analysis finds no such correspondence for CDR3, whose length, sequence and flexibility vary too much [2]. A generator therefore cannot draw CDR3 from a short list of known shapes.

Camelids also make antibodies that consist of heavy chains only [4]. Their antigen-binding domain, the VHH, works without a partner chain [5]. A VHH has the fold of a VH with three CDRs and a mass of about 12 to 15 kDa [6, 7]. Its CDR3 averages 18 residues, against 12 in mouse and 14 in human antibodies. Some VHH add a disulfide between CDR3 and CDR1 (camels) or CDR2 (llamas) [6].

In a conventional antibody, one face of the heavy-chain domain meets VL. Four FR2 positions (IMGT 42, 49, 50 and 52) lie on that face and usually carry Val 42, Gly 49, Leu 50 and Trp 52 in VH. In heavy-chain antibodies, where the face is exposed, they carry smaller or more hydrophilic residues, typically Phe or Tyr 42, Glu 49, Arg 50 and Gly 52 [2, 6]. IMGT uses the residues at these four positions to tell a VHH from a VH. The word nanobody is also used for constructs with more than one variable domain [2]; this manuscript uses VHH for the single domain. A VHH paratope draws on the framework more than a conventional one. Gordon et al. analysed 345 single-domain complexes and report that IMGT positions 66, 52, 55 and 42 lay in the interaction-defined paratope in 50.4, 31.6, 27.2 and 24.1 percent of the structures [8].

Numbering the domain. Insertions and deletions make a raw sequence index unreliable for comparing loops of different length. The same index can fall in a framework strand in one antibody and at a loop tip in another. The IMGT scheme gives corresponding positions the same label in heavy, light and VHH domains [9]. It divides the domain into four framework regions and three CDRs [2].

FR1 spans positions 1 to 26 and holds Cys 23. CDR1 spans 27 to 38, with gaps at the apex when it is shorter than 12 positions. FR2 spans 39 to 55 and holds Trp 41 and the four VHH hallmark positions. CDR2 spans 56 to 65, with gaps at the apex when it is shorter than 10. FR3 spans 66 to 104 and holds a hydrophobic residue at 89 and Cys 104. CDR3 spans 105 to 117. FR4 spans 118 to 128, and Phe or Trp 118 starts the J-segment motif F/W-G-X-G. The Methods give the region limits as the masks use them (Section 9.1).

A CDR3 longer than 13 residues gains positions between 111 and 112, added in the order 112.1, 111.1, 112.2, 111.2 [2]. Along the chain these read 111, 111.1, 111.2, 112.2, 112.1, 112, so sorting the labels as decimal numbers would scramble the sequence. The tool ANARCI assigns IMGT, Kabat, Chothia, Enhanced Chothia or AHo numbers by aligning a sequence to hidden Markov models of germline domains [10].

Formats. A design can take one of three formats. The format fixes how many variable domains the design must produce, whether a partner chain constrains them, and what a later measurement refers to. A VHH is one heavy-chain variable domain, so the design supplies three CDRs on a framework suited to a single domain. An Fv is a non-covalent VH/VL pair. It needs six CDRs, two frameworks and a consistent pairing. A single-chain variable fragment (scFv) joins VH and VL in one chain with a linker, and adds the linker length and the domain order to the design. The full system starts with VHH and adds Fv and scFv after their acceptance gates (Section 9).

Isolated VH or VL fragments are reported to be sticky and poorly soluble, with reduced affinity, although some keep their specificity. In an Fv, FR2 residues influence the VH/VL contact and so the relative orientation of the CDRs, which makes the pairing part of the design [3]. An scFv joins the domains with a flexible glycine- and serine-rich linker that has to span about 3.5 nm, and either domain can come first [11]. Antigen-binding fragments (Fab) and immunoglobulin G (IgG) add constant domains. In some systems these change the affinity or specificity of an unchanged variable domain [3]. A result for one construct needs justification before it is applied to another.

Epitope and paratope. The paratope is the binding site on the antibody. The epitope is the binding site on the antigen [3]. A complex structure supports a contact-based definition: an antigen residue belongs to the epitope when any of its heavy atoms lies within a cut-off distance of a heavy atom of the antibody. The set grows with the cut-off and depends on the conformation and assembly of the antigen. The design uses a cut-off of 4.5 Å.

An epitope is defined relative to the antibody that recognises it. The amino-acid composition of epitopes is nearly the same as that of other surface residues, and the epitope depends to a great extent on the antibody [3]. Registering an epitope states the surface patch a design should contact. It does not show that a candidate recognises that patch.

Contact is a weaker statement than energy. Binding free energy can concentrate in a few interface residues, the hot-spot hypothesis [3]. Competition between two antibodies does not prove a shared epitope, because antibodies on closely adjacent or minimally overlapping epitopes can displace one another, reversibly and in a dose-dependent way [12]. Antigens may also carry mobile glycans that shield potentially large fractions of the protein surface [13]. An epitope read from a bare protein structure can be partly covered on the native molecule.

What a binder design must satisfy. A binder proposed for a registered antigen and epitope has to meet five requirements together. The full system assesses them through separate computational checks.

- **Sequence acceptance.** Complete numbering, region assignments that fit the format, the two conserved cysteines (23 and 104) and no unpaired cysteine. These rules screen sequences. They do not show that the domain folds or forms a disulfide bond.
- **Target contact.** A predicted pose that contacts the registered epitope and is accessible on the native antigen. Affinity and kinetics need measurement.

- **Specificity.** Low or no binding to a named off-target panel under stated conditions. Polyspecificity is a recognised obstacle to developing antibodies found by in vitro display [14].
- **Developability.** Raybould et al. describe five computed risk metrics: total CDR length, surface hydrophobicity, positive and negative charge in the CDRs, and asymmetry of heavy- and light-chain charge [15]. Asparagine (Asn) deamidation and aspartate (Asp) isomerisation in CDRs depend on local flexibility and structure [16]. A new N-glycosylation sequon, the motif that permits attachment of a glycan, can alter antigen binding [17].
- **Immunogenicity risk.** A repertoire-based humanness score, whose association with clinical immunogenicity its authors reported for their data [18], and a predictor of antigen presentation by major histocompatibility complex (MHC) class II molecules [19]. Both are proxies.

The published developability thresholds derive from clinical-stage antibodies, and one metric compares heavy- and light-chain charge [15]. Single-domain formats have no light chain, so applying the guideline to them needs separate justification.

Why design is attempted and what a computation can show. Discovery by immunisation or by in vitro display selects binders from an animal repertoire or a library. Display allows defined selection conditions and returns the sequence at once [20]. Design can start from the target instead. The epitope is stated first, and liabilities are screened before any molecule is made.

Computed results differ in what they can support. Some are exact checks of a rule, such as a cysteine count, or algorithmic assignments, such as a numbering. Others are model outputs checked against held-out reference data, which needs trustworthy labels and a stated split, or predicted structures with a confidence score. None of these results shows that a designed molecule binds, separates its target from other proteins or can be expressed. Those statements need assays on the exact construct. Confidence scores are easy to over-read: independent benchmarks report that the interface predicted TM-score of structure predictors often fails to separate cognate from shuffled antibody-antigen pairs (Section 2.1).

2. Computational antibody design before generative models

Computational antibody design has long worked with two unequal data sets. The Observed Antibody Space database holds more than one billion antibody sequences from more than 80 studies [21]. The structural database SAbDab2 holds 22,550 antibody structures from 11,664 PDB entries [22, 23]. Both counts are from October 2026. The ratio is more than 40,000 sequences for each structure. The method families below use these two data sets in different ways. For each one we ask what it gives a designer of binders to a chosen epitope, and where it stops.

Physics-based docking and Rosetta design. Docking places a modelled antibody on an antigen and ranks the poses by an energy or a statistical potential. The docking program PIPER, behind the ClusPro server, was much less accurate for antibody-antigen pairs than for other complexes, and an asymmetric potential derived from antibody interfaces improved it [24]. Ambrosetti et al. tested four docking suites on 16 antibody-antigen complexes. HADDOCK, which takes restraints from the CDRs and from the epitope when known, gave the best success rate and model quality, with and without epitope information [25]. SnugDock lets the six CDR loops and the

VH/VL orientation move during docking. On 15 test complexes it gave four medium and seven acceptable lowest-interface-energy predictions on the CAPRI rating [26].

Design adds sequence change to the search. Lippow and colleagues designed single mutations from electrostatic terms and combined them. They report a tenfold gain for cetuximab, to 52 pM, and a 140-fold gain for the model antibody D44.1, to 30 pM [27]. RosettaAntibodyDesign (RAbD) grafts CDRs from canonical structural clusters, designs their sequences from cluster profiles and samples backbones with a flexible-backbone protocol. It starts from an experimental or modelled antibody-antigen structure. On 60 native complexes its design risk ratios for the CDRs other than H3 lay between 2.4 and 4.0, and replacing single CDRs raised the affinity of two test complexes by 10 to 50 fold. The authors name sampling and design of the H3 loop as a major challenge for true de novo design [28].

These methods start from an antibody that is already known to bind, or from restraints that place the epitope. They locate or refine a binder. Proposing a binder from scratch for an epitope chosen in advance lies outside what these studies report.

Antibody language models. Antibody language models learn the distribution of natural sequences. AntiBERTy and IgLM were each trained on 558 million sequences [29, 30], and AbLang on sequences from the Observed Antibody Space database [31]. IgLM re-designs a span of the sequence, such as one CDR loop, using the residues on both sides. Its infilled CDR libraries had improved in silico developability profiles [30]. Sapiens, a language model trained on the same database, was benchmarked on 177 antibodies and gave humanisation results that the authors rate as comparable to those of human experts [18]. General protein language models also help: with no antigen or structure input, they suggested mutations that raised the affinity of four mature antibodies by up to sevenfold and of three unmatured antibodies by up to 160-fold, with 20 or fewer variants screened per antibody [32].

The limit follows from the input. A sequence model has no antigen, so it cannot choose an epitope. Its likelihood scores resemblance to natural sequences, and any link to binding has to be measured. A large part of each variable domain stays germline, which pulls models towards germline residues, yet mutations away from germline are often vital for specific binding. AbLang-2 was built to reduce this bias [33].

2.1. Structure prediction

Two kinds of predictor matter. Antibody-specific models predict the Fv from sequence. DeepAb [34], IgFold [35] and ABodyBuilder2 [36] are examples. IgFold predicts a structure in under 25 s and was run on 1.4 million paired sequences [35]. On 34 recently solved antibodies, the mean CDR-H3 RMSD was 2.81 Å for ABodyBuilder2, 2.90 Å for AlphaFold-Multimer and 3.28 to 3.46 Å for four other tools. CDR H3 was the least accurate loop for every tool in that comparison [36].

General complex predictors read the antibody and antigen sequences together. AlphaFold 2 reached accuracy competitive with experiment for most CASP14 targets [37], and AlphaFold-Multimer extended it to complexes [38]. An independent benchmark found a success rate of 11% for AlphaFold-Multimer on recently released antibody-antigen structures [39]. AlphaFold 3 reports substantially higher antibody-antigen accuracy than AlphaFold-Multimer v2.3 [40]. Boltz-1 and Chai-1 are models of the same class with released weights, and Boltz-2 followed [41–43]. In an independent test with one seed, AlphaFold 3 reached high-accuracy docking for 10.2% of antibodies and 13.3% of nanobodies, and Boltz-1 and Chai-1 for 5% or less [44].

Two further findings matter for design, and the section on evaluation gives the numbers (Section 5). First, a correct model is often among the samples but not at the top [45]. Second, confidence does not follow binding. The interface predicted TM-score (ipTM) of current predictors ranks cognate nanobody-antigen pairs only weakly above shuffled non-cognate pairs [46]. A design pipeline that filters candidates by these scores inherits both weaknesses: the best model is hard to pick, and cognate pairs are hard to separate from non-cognate ones.

Inverse folding. Inverse-folding models predict a sequence for a given backbone. ProteinMPNN recovered 52.4% of native residues on native backbones, against 32.9% for Rosetta, and rescued designs that had failed with Rosetta or AlphaFold [47]. ESM-IF1 recovered 51% on structurally held-out backbones after its training data were extended with AlphaFold2 predictions for 12 million sequences [48]. AntiFold fine-tunes ESM-IF1 on solved and predicted antibody structures. It outperforms existing tools on sequence recovery across the CDRs, and in zero-shot use its predictions correlate more strongly with antibody-antigen affinity [49]. An inverse-folding model needs a backbone, so another method must supply the backbone and its placement on the epitope. Recovery of native residues scores agreement with the natural sequence of a solved structure. Binding to a new antigen is a separate question.

Chained pipelines. The families can be chained. One published pipeline combined Ig-VAE scaffolds, ProteinMPNN and RFantibody for sequence design, tFold-Ab and tFold-Ag for structure prediction, four docking programmes, molecular dynamics and humanisation tools [50]. It was semi-automated and set cut-offs on ipTM, predicted aligned error (pAE) and RMSD. It reports computed binding free energies for three humanised nanobody candidates.

Each family answers part of the design question. Docking and Rosetta design work around a binder that exists. Language models know no antigen. Structure predictors model complexes, and their scores are weak judges of binding. Inverse folding needs a backbone. The studies cited here use different antibodies, antigens, cut-off dates and success definitions, so their numbers cannot be ranked against one another. Several come from preprints without peer review. Generative structure models, covered next, attempt to combine these pieces in one model.

3. Generative models of protein structure

A generative model of protein structure learns the distribution of known structures and draws new samples from it. Two training recipes dominate the field: denoising diffusion and flow matching. Both turn random noise into a structure through many small steps, and they differ in what the network is asked to predict. The families below lead to La-Proteina and Proteina-Complexa, the two published models on which Omniverso builds. These papers supply architecture, dimensions and a reference for training scale.

Diffusion and score models. A diffusion model adds Gaussian noise to a training example in many small steps and trains a network to undo them [51]. In the continuous-time view, the network learns the score, the direction in which the density of noisy data rises fastest. Sampling follows a reverse-time stochastic differential equation, or an equivalent ordinary one, from noise back to data [52]. RFdiffusion brought this recipe to protein backbones. It fine-tuned the structure predictor RoseTTAFold to denoise residue frames and left sequence design to a second network, ProteinMPNN. Its authors tested binders to five targets in the laboratory [53]. Chroma used a

diffusion process matched to polymer statistics and samples structure and sequence together; its authors tested 310 designed proteins experimentally [54]. AlphaFold 3 places a diffusion-based architecture inside a predictor of complexes [40].

Flow matching. Flow matching trains the same kind of network on a simpler target. Draw a noise sample $\bar{q}_0$ and a data sample $\bar{q}_1$, join them by a straight line, and ask the network to predict the velocity along the line. Here $\bar{q}$ is the state of a design made dimensionless as the Methods describe, and $u \in [0, 1]$ is the position on the line. The network output is v_θ, and the expectation $\mathbb{E}$ runs over pairs and positions:

$$\bar{q}_u = (1 - u)\,\bar{q}_0 + u\,\bar{q}_1, \qquad \mathcal{L} = \mathbb{E}\left\|v_\theta(\bar{q}_u, u) - (\bar{q}_1 - \bar{q}_0)\right\|_2^2.$$

The loss is the squared distance between the network output and the velocity of the line, which is $\bar{q}_1 - \bar{q}_0$. Averaged over many pairs, the network learns the field that carries the noise distribution onto the data distribution. Generation integrates the learned field from $u = 0$ to $u = 1$ with an ordinary differential equation solver [55–57].

Gaussian paths include the diffusion paths as special cases [55]. For a straight path from independent Gaussian noise, the score can be read off the velocity v, where p_u is the density of states at position u:

$$\nabla_{\bar{q}} \log p_u(\bar{q}) = \frac{u\,v(\bar{q}, u) - \bar{q}}{1 - u}.$$

A trained flow can therefore also run as a stochastic sampler [58, 59]. Pairing noise and data samples by minibatch optimal transport shortens the lines [60]. Short lines allow few solver steps. FrameFlow used five times fewer steps than the diffusion model it was adapted from [61].

Models with explicit rotations. A backbone can be described by one rigid frame per residue: a position and a rotation. FrameDiff diffuses these frames on SE(3), the group of rotations and translations of three-dimensional space, so that the model does not depend on how a protein is oriented [62]. Riemannian flow matching carries the straight-line recipe over to curved spaces such as the rotation group [63]. FrameFlow and FoldFlow apply it to frames [61, 64]. MultiFlow adds a discrete flow over the twenty amino-acid types, a continuous-time Markov chain, and generates sequence and backbone together; it leaves side chains out [65]. FoldFlow-2 conditions on sequence through a frozen protein language model and reports a scaffolding task around the CDR loops of VHH nanobodies [66].

Proteina takes the opposite route. It models C-alpha positions only and uses a transformer with no built-in rotation symmetry. Symmetry is taught by centring each protein and rotating it at random during training. The network keeps a table of features for every residue pair, the pair representation, which biases attention. Triangle layers that update this table are optional. Proteina trains models of 200 M and 400 M parameters on up to 20,874,485 predicted structures [58].

Latent models. A latent model runs the generative process in the compressed space of an autoencoder, as latent diffusion does for images [67]. For all-atom proteins the compression answers two difficulties. The number of atoms per residue depends on the residue type, so an atom array that follows the sequence has no fixed shape. The sequence is also categorical, while the flow above acts on continuous variables. A fixed-width continuous latent per residue removes both [68]. PLAID runs diffusion in a shared sequence-structure latent space taken from a pretrained predictor.

Its diffusion model is trained on sequences only, so it can use far more data than structure databases hold [69]. LSD generates a coarse latent that decodes to a contact map, then diffuses coordinates conditioned on that map [70]. La-Proteina differs from both: it moves an explicit C-alpha trace and a residue latent together in one flow.

La-Proteina. La-Proteina keeps the C-alpha coordinates $x \in \mathbb{R}^{n_e \times 3}$ explicit and gives each residue a latent $z \in \mathbb{R}^{n_e \times 8}$ of eight continuous numbers, for n_e transported residues. Together they form 11 numbers per residue. A variational autoencoder learns the latent. Its encoder reads the sequence and all atoms. Its decoder reads x and z and returns residue-type probabilities and Atom37 coordinates, the 37 atom slots per residue, of which the decoded residue type selects the valid ones [68].

A denoiser then learns a flow over (x, z) with the autoencoder frozen. Each block has its own interpolation time, t_x and t_z, drawn independently, and its own velocity head. Both heads see both times. Subscript 0 marks noise and subscript 1 data:

$$x_{t_x} = (1 - t_x)\, x_0 + t_x\, x_1, \qquad z_{t_z} = (1 - t_z)\, z_0 + t_z\, z_1,$$
$$\mathcal{L} = \mathbb{E}\Big[\big\|v_\theta^x - (x_1 - x_0)\big\|_2^2 + \big\|v_\theta^z - (z_1 - z_0)\big\|_2^2\Big].$$

Each block follows its own straight line, and the loss adds the two squared errors. The Methods divide each term by the size of its block. Two clocks let sampling move the backbone ahead of the latent. In the authors' ablations on monomers, sampling the backbone faster than the latent was needed for good all-atom co-designability, and a latent that also encoded the C-alpha trace gave worse results.

The paper gives the encoder and decoder 12 transformer blocks and about 130 M parameters each, and the denoiser 14 blocks and about 160 M. The main models use no triangle layers. The training data are cluster representatives of the AlphaFold Database, about 350 thousand to 550 thousand structures of up to 512 residues. A long-chain model draws on 46,942,694 structures grouped into 4,035,594 clusters. The paper reports unconditional monomers of up to 800 residues and atomistic motif scaffolding, and no antibody data.

Proteina-Complexa. Proteina-Complexa extends La-Proteina to binder design [71]. The target enters the denoiser as extra tokens: Atom37 coordinates, residue identity and a flag that marks the hotspot residues near which the binder should form. These tokens are joined to the binder tokens, and the pair features cover binder and target together. The autoencoder is unchanged and encodes the binder alone, so one autoencoder serves every target. Training also adds a random global translation to the binder, with a standard deviation of 0.2 nm, so that the model must learn where the binder sits relative to the target.

Training runs in stages. The autoencoder learns on predicted monomers and then on PDB chains. The denoiser learns first on predicted monomers and then on Teddymer, a set of synthetic dimers made by splitting multidomain AlphaFold Database proteins (10 M dimers in 3.5 M clusters), mixed with PDB multimers at a Teddymer to PDB ratio of 8 to 2. At generation the model samples with reduced noise and adds test-time search: best-of-N, beam search, Feynman-Kac steering and Monte Carlo tree search, scored by the confidence of a structure predictor. The paper evaluates protein targets, small-molecule targets and enzyme design by in silico criteria only and tests no binder in the laboratory. It names antibodies only as a possible modality for future work.

3.1. Why partially latent flow matching suits antibody-antigen design

This is an argument from the published constructions. Five properties of the construction match the problem.

First, an antibody-antigen interface is made of side chains. Which residue sits at a CDR position, and how its side chain points, decide the contacts. A backbone-only generator leaves sequence to a second network. The partially latent state yields sequence and all-atom coordinates from one flow [68].

Second, the state has a fixed size. Eight latent numbers plus the three C-alpha numbers give 11 numbers per residue whatever the sequence, and every number is continuous. The main generator needs no categorical flow.

Third, the C-alpha trace stays explicit. Proteina-Complexa shows how target tokens, hotspot flags and translation noise place a binder against a target, and an antigen can enter the denoiser the same way.

Fourth, the autoencoder does not depend on the antigen. In Proteina-Complexa only the denoiser sees the target, so one autoencoder serves every antigen.

Fifth, the networks avoid triangle layers. The all-atom baseline P(all-atom), which uses them, ran out of 140 GB of GPU memory beyond 500 residues [68]. An antibody-antigen complex spans several hundred residues, so memory sets a limit.

Neither La-Proteina nor Proteina-Complexa reports an antibody or nanobody result, and neither handles IMGT numbering, framework and CDR roles or heavy-light pairing. Omniverso therefore applies a published construction to a new domain. Antibody generators built on related constructions are surveyed in the next section.

Published training compute. The reports state hardware, optimiser steps and batch size but no floating-point operation count for training. Table 1 lists what each paper states.

The SE(3) backbone models trained in days on two to four cards, FoldFlow-2 with a language model pretrained elsewhere. The denoiser of Proteina-Complexa trained in two stages on 32 and then 96 A100-80GB cards and saw 374.4 M samples in total. Test-time search adds inference budgets: Proteina-Complexa uses 16 and 32 GPU-hours per target in its scaling tests and up to 1,100 GPU-hours in one case study. The RFdiffusion row gives the figure listed by the FoldFlow-2 authors.

The base model of this work has 255,873,676 autoencoder parameters and 153,102,091 denoiser parameters (Section 11.4). Its budget rule is given in Section 12. A sample count measures data seen, and compute also depends on sample length, so the comparison marks scale only.

4. Published antibody and nanobody generators

Since 2022, several groups have published generators that propose antibodies or nanobodies on a computer, and most report laboratory tests. We read the primary paper, preprint or technical report of each system in Table 2 in October 2026. The numbers below are those the authors report; none was re-measured. Where versions of a paper differ, the version is named. Several sources

Table 1: Training scale stated by the papers behind the generative models in this section. Samples seen are steps × GPUs × batch per GPU, in millions, assuming one microbatch per update. The Proteina-Complexa and La-Proteina runs used A100-80GB cards; the card model of the Proteina runs is not stated. n/r, not found in the passages read. AFDB, AlphaFold Database.

System and stage	Trainable parameters	Training set	Steps, GPUs, batch per GPU	Samples seen (millions)
FrameFlow [61]	17.4 M	3,938 SCOPe examples below 128 residues	steps n/r; 1 day on 2 A100 GPUs	n/r
MultiFlow [65]	n/r	18,684 PDB proteins of 60 to 384 residues	200 epochs; 3 days on 4 A6000	n/r
FoldFlow-2 [66]	21 M (frozen ESM2-650M excluded)	PDB plus filtered synthetic structures	500 k steps; 4 days on 2 A100-40GB (192 GPU-hours)	n/r
RFdiffusion, as listed in [66]	n/r	PDB	25 k steps; 64 + 8 GPUs; 28 + 3 days, with RoseTTAFold initialisation	n/r
Proteina M_FS, 5 triangle layers [58]	200 M	588,318 AFDB clusters, 32 to 256 residues	200 k × 128 × 4	102.4
Proteina M_FS, no triangle layers	200 M	same	360 k × 96 × 10	345.6
Proteina M_21M, 4 triangle layers	400 M	20,874,485 AFDB structures, up to 256 residues	180 k × 128 × 4	92.2
La-Proteina autoencoder, 3 stages [68]	about 130 M each, encoder and decoder	AFDB clusters, up to 256, 512, 896 residues	500 k × 16; 140 k × 32; 180 k × 32; batch n/r	n/r
La-Proteina denoiser, 3 models	about 160 M	AFDB clusters up to 512 residues; long-chain pool of 46,942,694	390 k × 48 (no triangle layers); 120 k × 96 (triangle layers); 140 k × 128 (long chains); batch n/r	n/r
Proteina-Complexa autoencoder, AFDB stage [71]	256 M (encoder plus decoder)	AFDB monomers, up to 256 residues	500 k × 16 × 14	112.0
Proteina-Complexa autoencoder, PDB stage	256 M	110,976 PDB chains	40 k × 16 × 12	7.7
Proteina-Complexa denoiser, pretraining	159 M	AFDB monomers, up to 256 residues	540 k × 32 × 12	207.4
Proteina-Complexa denoiser, fine-tuning	159 M	Teddymer plus PDB multimers, 8 to 2	290 k × 96 × 6	167.0

are preprints or company reports without peer review. Figure 6 summarises the systems by family and publication year beside the fixed-framework and de novo redesign scopes on PDB 4KRL.

Table 2: Published antibody and nanobody generators, as the primary texts report them (Germinal: version 3). Compute is training unless stated, in GPU-hours where cards and time are given. n/r, not found in the passages read. CDR, complementarity-determining region; FR, framework region; Fv, variable fragment; scFv, single-chain Fv; SPR, surface plasmon resonance; BLI, biolayer interferometry; OAS, Observed Antibody Space. †, technical report without a registry identifier.

System	Year	Conditioning	Framework	Training data	Laboratory test	Compute
DiffAb	2022	Antigen and bound-antibody framework; one CDR at a time	Fixed, from the native complex	34,781 pseudo-CDR complexes added (v1); size otherwise n/r	None	One A100; hours n/r
Absci 2023	2023	HER2, epitope, trastuzumab framework and CDR lengths	Fixed	n/r	About 10^6^ display variants; 421 SPR-confirmed binders; one target	n/r
AbDiffuser	2023	None; HER2 task on binder-labelled CDR-H3 mutants	Generated	n/r; 169 M parameters	16 variants of trastuzumab; 37.5% bound HER2	One A100 80 GB; hours n/r
IgDiff	2024	None, or all CDRs on a fixed framework	Generated	n/r	28 sequences expressed; no binding assay	144 A10G GPU-hours (fine-tuning)
IgGM	2024	Antigen, framework sequence, optional epitope	Sequence fixed	Structures to 2022-12-31	7 of 60 scFv bound PD-L1 (2025 preprint)	1,920 A100 GPU-hours (two stages)
RFantibody	2024	Target, hotspots, framework, CDR lengths	Fixed	Antibody complexes; size n/r	Six targets; binders to four epitopes; cryo-EM	n/r
Germinal	2025	Predicted antigen structure, epitope, framework	Fixed by sequence bias	No network trained	Four targets; BLI binders; one cryo-EM	200 to 500 H100 GPU-hours per 200 to 400 designs
mBER	2025	Target, one hotspot, partly masked VHH	Fixed (IGHV3-23 based)	No network trained	145 targets by pooled phage display	n/r
BoltzGen	2025	Target, scaffold, optional binding site	Fixed (four scaffolds)	PDB to 2023-06-01	Nine targets (v1); nanomolar binders for six	n/r
Chai-2	2025	Target, up to four epitope residues	Fixed (five VHH, five VH-VL)	Structures before 2021-09-30	52 targets; 15.5% of tested designs bound; five cryo-EM structures	n/r
JAM-2†	2025	Target; epitope in the second set	CDRs plus a few FR2 and FR3 residues	n/r	16 targets; mean success 39% (VHH-Fc), 18% (antibody)	n/r
Latent-X2	2025	Target, one to four hotspots, framework, CDR lengths	Fixed (five VHH, five Fv)	n/r	18 targets; validated binders for nine	n/r
Origin-1	2026	Antigen, epitope, framework, CDR lengths	Fixed (four therapeutic)	Structures before 2024-01-01	Ten targets; hits for four; cryo-EM	Search up to about 1,005 H200 GPU-hours per target (v2)
tFold system	2025	Antigen, epitope, framework	Fixed germline	Structures to 2021-12-31	Four targets; pooled phage libraries	n/r

Continued on next page

System	Year	Conditioning	Framework	Training data	Laboratory test	Compute
GeoFlow-V3	2025	Antigen, up to 10 epitope residues	Fixed (h-NbBcII10 FGLA)	n/r	Five targets; up to 50 designs per campaign	n/r
GaluxDesign	2025	Target, two to six epitope residues	n/r	n/r	Six targets, eight epitopes; ELISA and BLI	n/r
UniMoMo	2025	Binding-site residues; optional framework	Fixed	9,473 complexes	None	n/r
AgForce	2026	Antigen, epitope, framework	Fixed	2,338 complexes (CHIMERA-Bench split)	None	About 2 H100 GPU-hours
Ophiuchus-Ab	2026	None	Fixed for infilling, generated for humanisation	2,511,612 OAS pairs	None	8 A100 GPUs, 100 k steps; hours n/r

CDRs on a given complex. The first generators work on a complex or scaffold that already exists. DiffAb diffuses amino-acid types, positions and orientations to redesign one CDR at a time, given the antigen and the framework of the antibody bound to it. Its final test set holds 19 complexes from five CDR-H3 clusters, and it reports no laboratory test [72]. AbDiffuser and IgDiff generate the paired variable region, framework included, and take no antigen as input [73, 74]. Both tested few variants. For AbDiffuser, 37.5% of 16 submitted variants of trastuzumab bound HER2. For IgDiff, 28 sequences were expressed, and the paper has no binding assay. Absci's zero-shot study redesigned the heavy-chain CDRs of trastuzumab against HER2 with the epitope and framework given. The top 1,000 designs bound at 10.6% for CDR-H3 alone and 1.8% for all three heavy CDRs (version 4) [75].

IgGM generates CDR sequences and the structure of the whole variable region from the antigen structure, the framework sequence and an optional epitope [76]. Its benchmark uses 60 antibodies and 27 nanobodies released in the second half of 2023, after removal of sequences similar to the training set. A design counts as a success when DockQ exceeds 0.23, and the success rates are 0.433 for antibodies and 0.415 for nanobodies. A 2025 preprint adds one laboratory test: seven of 60 scFv designs against the PD-1 site of PD-L1 bound [77].

Epitope-conditioned design with public models. RFantibody fine-tunes RFdiffusion on antibody complexes. The user supplies the target structure, hotspot residues, a framework and the CDR lengths. ProteinMPNN designs the CDR sequences, and a fine-tuned RoseTTAFold2 filters the designs [78, 79]. Six targets were designed and binders to four epitopes were confirmed. Yeast display screened about 9,000 designs per target, and single-concentration SPR tested 95 per target. Success rates were 0 to 2% per target and the best affinities ran from 78 nM to 5.5 μM. A cryo-EM structure of one influenza haemagglutinin VHH matched its design at 3.0 Å. The authors classify the structure of one RBD VHH as a design failure. Later versions add scFvs, and the working scFvs came from heavy and light chains recombined across designs.

Germinal and mBER train no network. They optimise CDR sequences on a given framework by gradient descent through AlphaFold-Multimer. Germinal adds an antibody language-model term and a sequence-redesign step [80, 81]. It tested 43 to 101 nanobodies for each of four targets chosen by the authors. In version 3, BLI confirmed 7 of 25 PD-L1 designs taken to BLI, 2 of 11 for IL3, 4 of 11 for IL20 and 5 of 20 for BHRF1.

mBER designed 1,153,241 VHH sequences against 436 human cell-surface proteins and measured binding on 145 targets by pooled phage display [82]. On-design hit rates were significantly raised for 65 of the 145 targets (45%), and the median hit rate per design was about 0.4% among the 78 targets with any on-design binding. The paper reports no affinity and no structure.

BoltzGen is a binder-design model built on the Boltz-2 architecture. For nanobodies it fixes one of four default scaffolds and replaces the three CDRs with loops of random length [83]. In version 1, up to 15 nanobodies were tested for each of nine targets that the authors call novel, and nanomolar binders were found for six of the nine. Version 2 separates a screening hit from a confirmed binder and adds a tenth target.

Company reports. Chai Discovery, Nabla Bio, Latent Labs and Absci report antibody design. The first three offer access only through partnerships or a platform. Chai-2 designs VHH and scFv from a target structure and up to four epitope residues, and designs all CDR residues and their lengths on a scaffold chosen by the user [84]. On 52 targets, with up to 20 designs tested per target and format, 15.5% of tested designs bound by BLI (20.0% VHH, 13.7% scFv). At least one binder was found for 26 targets. A second report reformats scFv hits as 88 IgGs, compares their developability with clinical antibodies, and shows five Fab-antigen cryo-EM structures at 2.9 to 3.9 Å in version 2 [85].

JAM-2† is a technical report of Nabla Bio, a PDF on the company website (`nabla-public.s3.us-east-1.amazonaws.com/2025_Nabla_JAM2.pdf`). It has no DOI, arXiv number or PubMed entry, is named here by its location and is not in the reference list. It tests VHH-Fc and full-length antibodies. On 16 targets with 45 designs per format, the report gives a mean per-target success of 39% for VHH-Fc and 18% for antibody. A second set of 10 targets with 20 chosen epitopes each was screened by yeast display, with about 600 VHH designs per epitope. For the GPCRs CXCR4 and CXCR7, 86 of 736 and 28 of 728 designs bound. The first JAM paper tested pooled libraries of about 10,000 designs per target, with hit rates of 0.03 to 0.19% [86].

Latent-X2 designs VHH and scFv from the target structure, one to four hotspot residues and one of five VHH or five Fv frameworks [87]. On 18 targets, 4 to 24 designs per modality and target gave validated binders for nine. The preprint ties "validated" to a reproducible five-point BLI or SPR fit. It also reports immunogenicity assays on one target, with four VHH and ten donors.

Origin-1 combines an all-atom diffusion model fine-tuned from Boltz-1, a sequence-design model and a co-folding score. It designs six CDRs on four therapeutic frameworks and tests full-length antibodies [88]. Its ten targets have no complex structure in the PDB and at most 60% identity to any protein that has one. With at most 95 antibodies tested per target, hits were found for four targets, one to four each. Cryo-EM at 3.0 to 3.1 Å matched two designs (version 1). The authors state that their hit rates are lower than others have recently reported.

Other groups report related systems. The tFold system of Wu and colleagues selected scFv from pooled phage libraries of 10,000 designs for each of four targets [89]. GeoFlow-V3 and GaluxDesign report five and six targets, with up to 50 designs per campaign or epitope [90, 91]. Ma and colleagues describe an adaptive deep-learning method for pocket-targeted CDR design [92]. TFDesign-sdAb adds Protein A binding to existing single-domain antibodies by mutating framework residues; it modifies a known antibody and proposes no binder from scratch [93]. UniMoMo, AgForce and Ophiuchus-Ab report computational benchmarks only, and the last has no antigen input [94–96].

Framework handling. The systems differ most in how they treat the framework. Of the 19 systems in Table 2, twelve keep it fixed outright, and two more, IgGM and Germinal, hold it by its sequence or by a bias. Several draw it from a short list: four scaffolds for BoltzGen and Origin-1, five VHH and five Fv frameworks for Latent-X2, and five VHH and five VH-VL frameworks for Chai-2. Germinal's bias still permits mutation. IgGM takes the framework sequence from the user and generates its structure. JAM-2† generates the CDRs and a few adjacent framework 2 and 3 residues that contact the antigen; the first JAM paper does not state how the framework is chosen. AbDiffuser and IgDiff generate the framework but take no antigen. Ophiuchus-Ab fixes the framework for infilling and generates it for humanisation, and GaluxDesign does not report framework handling.

The primary texts report no epitope-conditioned generation of the whole framework, and none compares fixed and generated frameworks at matched compute. A fixed framework lets the authors pick one with known properties; Chai-2 used VH3-23 and VH3-66 for their thermostability. The primary texts do not say whether a fixed scaffold limits the reachable epitopes.

Validation and specificity. Binding is the usual readout, by BLI, SPR, yeast or phage display, or ELISA. Structures of designs are reported for RFantibody, Germinal (one scFv at 3.9 Å), Chai-2 (five complexes) and Origin-1 (two in version 1, three in version 2). No primary text reports a self-proteome panel, and the specificity tests are narrow. Chai-2 tested six antigens, and one of 23 designs bound above background. BoltzGen tested human serum albumin, JAM-2† ovalbumin, and Origin-1 two non-antigen proteins together with cross-design checks. The pooled mBER screen gives the widest view: among the 331,578 designs whose targets were screened, it found 2,207 on-design hits and 11,040 off-design hits. Chai-2, JAM-2†, Latent-X2 and Origin-1 report developability panels on expression yield, size-exclusion profile, hydrophobicity and polyspecificity. Compute is rarely stated, and GPU-hours do not transfer between card types; Table 2 lists what each paper gives.

4.1. Protocols behind the reported rates

Six features of the study protocols determine the size of a reported rate.

- **Targets.** Each study chose its own, and overlap is rare. The examples span one-target studies and the pooled screen of mBER on 145 targets. Target counts must be read with the assay format and the candidate-selection procedure. JAM-2† tests 16 targets that overlap the Chai-2 study, in other formats. Epitopes were chosen by the authors, often from known complexes.
- **Novelty and cut-off.** Chai-2 excludes antigens above 70% identity over 80% coverage to antibody antigens released before 30 September 2021. BoltzGen excludes proteins above 30% identity in any bound PDB structure and trained on structures up to 1 June 2023. Origin-1 allows at most 60% identity to any protein with a complex and trained on structures before 1 January 2024. The tFold system trained on structures to 31 December 2021, and IgGM to 31 December 2022. The main texts of RFantibody, Germinal, mBER and Latent-X2 state no such criterion.
- **Designs tested.** Counts run from 4 to 24 per target and modality (Latent-X2) through 15 (BoltzGen) and up to 20 (Chai-2) to 43 to 101 (Germinal), and pooled libraries test 10,000 or more. A rate on individually tested designs and a rate on a pool are on different scales.
- **Selection before testing.** Tested designs are the best-ranked by structure-predictor confi-

dence, so a rate depends on the filter and on the pool it drew from. The pass rate before filtering is rarely given.

- **Success definition.** Thresholds include a BLI response above 0.1 nm at one concentration (Chai-2 at 5 or 10 μM, JAM-2† at 500 nM), a reproducible five-point kinetic fit (Latent-X2) and an ELISA signal three times the negative control (tFold system). Origin-1 requires SPR above 10 RU at 2 μM with no off-target binding, and mBER uses pooled enrichment tested by Fisher's exact test. Formats also differ: monovalent VHH, dimeric VHH-Fc, scFv and IgG.
- **Level of the rate.** A rate per design, per epitope or per target answers different questions. At least one binder was found for 26 of 52 targets (Chai-2), 9 of 18 (Latent-X2), 6 of 9 (BoltzGen) and 4 of 10 (Origin-1).

Reported numbers also change between versions. Germinal's BHRF1 count fell from 11 of 20 in version 1 to 5 of 20 in version 3. BoltzGen redefined its hit between versions. The cryo-EM resolutions in the second Chai-2 report changed from 3.5 to 5.7 Å in version 1 to 2.9 to 3.9 Å in version 2. A rate is therefore quoted with its version.

Independent comparisons. The AIntibody challenge is a blinded, prospective comparison. It used a single antigen, the SARS-CoV-2 receptor-binding domain, asked for CDR redesign on fixed frameworks, and collected 511 sequences from 29 organisations [97, 98]. Its authors report that design outside the library was highly variable and that many submissions did not outperform standard selections. The main text of that paper names none of the generators in Table 2. CHIMERA-Bench retrains eleven earlier CDR generators on shared epitope-group, antigen-fold and temporal splits, and could not evaluate several methods under leakage control, RFantibody among them [99]. No shared benchmark connects the epitope-conditioned systems.

Weights and licences. Code and weights are released for DiffAb (Apache-2.0), IgGM, RFantibody, BoltzGen and mBER (MIT). Germinal's code is under Apache-2.0 but needs PyRosetta and IgLM, which carry non-commercial terms. The tFold code is under a non-commercial licence. Chai-2, JAM, JAM-2† and Latent-X2 are available only through partners or a platform. The Origin-1 repository holds designs and data; model weights were not found. The base model described here loads no released weights.

5. Evaluation, specificity and developability

A generated antibody is typically judged in three steps. A structure-prediction model scores the designed sequence against the antigen, property predictors screen the survivors, and a small set goes to the laboratory. The first step does most of the filtering, and the scoring model is a trained network with its own error rate. One benchmark paper describes the common workflow as generating thousands to millions of candidates and filtering them on internal confidence scores such as pLDDT, ipTM and PAE [46]. Such filters help: using AlphaFold2 or RoseTTAFold to assess whether a designed sequence folds and binds as designed raised de novo binder success nearly 10-fold [100].

Leakage and temporal holdouts. A result is only as informative as the separation between the test set and everything the model has seen. One analysis reports major leakage under common splits by sequence or metadata similarity and recommends splitting on the three-dimensional similarity

of interfaces [101]. A model can also meet the test set in pretraining: for protein melting-point prediction, a split that accounts for the pretraining corpus changed the measured performance consistently against a naive split [102]. For antibody design, CHIMERA-Bench states that methods are evaluated on different SAbDab snapshots, non-overlapping test sets and incompatible metrics, and provides splits for unseen epitopes, unseen antigen folds and later-deposited targets [99]. A temporal holdout works only when its date belongs to the right model. One benchmark of 110 antibody-antigen complexes was cut at 30 September 2021, the training cut-off its authors state for all four methods tested [45]. Another used public entries deposited after May 2024 plus confidential structures [103]. A date alone does not exclude an old antibody redeposited with a new antigen variant, so identity and structure checks accompany it.

This study uses four leakage axes. Antigen sequence identity is limited to 40% at 80% coverage and antigen structure to a TM-score of 0.50 at 80% coverage. Epitope overlap is limited to a Jaccard index of 0.30 on residues within 4.5 Å of the partner, and CDR-H3 identity to 80% at 80% coverage. Records linked on any axis go to the same side of the split, and a temporal cut-off of 1 January 2025 by PDB deposition date sends records deposited on or after it to the holdout (Section 10.3). These checks are a floor. They miss shared interface geometry, overlap with a foundation model's pretraining corpus, and group-level links such as lineages, assay plates and campaigns. A claim that a model generalises beyond its training records needs a prospective cohort collected after the model is frozen.

Co-folding judges and their calibration. A co-folding judge predicts the antibody-antigen complex and returns a confidence score. Three questions decide whether it can be used: does it find a correct pose, can it pick that pose among its own samples, and does its confidence separate binders from non-binders?

Pose accuracy for antibodies is low. FoldBench reports antibody-antigen failure rates above 50% for current methods [104]. On a benchmark of structures deposited after May 2024, the success rates of seven predictors rarely exceed 25% [103].

Picking the pose is the second weakness. For AlphaFold 3 on 110 antibody-antigen complexes unseen in training, the mean DockQ of the best of 200 models rose above 0.5, while the model ranked first by the judge's own confidence reached about 0.37. The authors trace the gap to errors in the predicted aligned error [45]. SNAC-DB finds the same pattern for Boltz-2 on 101 nanobody complexes. Across 1 to 1,000 samples per target, oracle selection improves from 11.9% to 50.5% success, while confidence ranking stays between 10.9% and 14.9% [103]. More samples raise the best pose the judge could return and leave the pose it does return largely unchanged.

Separating binders from non-binders is the third. Smorodina and colleagues paired 106 cognate nanobody-antigen complexes with 11,342 shuffled pairings. The average precision with which ipTM ranked cognate above shuffled pairs was 0.187 for AlphaFold 3, 0.067 for Chai-1 and 0.026 for Boltz-2, against a random baseline near 0.011. More sampling improved the structures without improving the separation [46]. The authors assume the shuffled pairs do not bind and note that a small share may. The score also depends on the input construct. ipTM falls when disordered or accessory regions are included although the interface is unchanged, and an interface-restricted score, ipSAE, separates true from false complexes better than ipTM does [105]. Across 3,766 designed binders against 15 targets, ipSAE outperformed ipAE and ipTM, and performance varied by target [106]. On 4,184 antibody-antigen pairs, Liang and colleagues separate within-antigen ranking from cross-antigen ranking and show that interface confidence scores lose calibration across distinct antigens

[107].

Designs chosen by a model are not an independent sample of the data that model was fitted on, and finite-sample confidence sets exist for that dependence [108]. Calibration is therefore measured on the population being judged. A judge report states the model, version, seeds, construct, score and threshold, with its error rate on decoys and on designs. The share of all designs that the judge admits and an external assay then fails is the quantity of interest (Technical Report III (Section "Judging and specificity")).

5.1. Specificity, developability and immunogenicity

Specificity. Specificity is relative to a panel. A candidate is specific if it binds the target and does not bind a named set of other proteins under a stated assay. Panels hold orthologs from the pharmacology species, paralogs, abundant circulating proteins and, for a systemic drug, membrane and secreted human proteins. A panel samples the proteome, so a clean result holds for its members only. Polyreactivity belongs to the candidate alone. The polyspecificity-reagent assay is a flow-cytometric readout compatible with yeast display, and it correlates with cross-interaction chromatography, a solubility surrogate, and with a baculovirus-particle ELISA, a surrogate for clearance [14]. For nanobodies the reagent was a detergent-solubilised insect-cell membrane preparation, and a sequence model trained on the readout reached an AUC above 0.8; over 90% of its predicted substitutions reduced polyreactivity in three scaffolds [109].

Pair-specific off-target binding belongs to a candidate and one named protein. A clean polyreactivity score does not show that a given paralog is unbound. A pair-specific panel does not detect general stickiness. A report needs both.

Computed panels are younger. SurfDiff scores target surface patches against related proteins before any binder exists, which ranks epitopes by selectivity without scoring any candidate [110]. OdinMulti, for protein binders, treats cross-reactivity and specificity as explicit design objectives [111]. Co-folding confidence is a weak pair-specific oracle, as the shuffled-pair results above show, and a pair oracle needs matched assay labels for candidate-antigen pairs and a split by antigen family. The full system reports panel coverage beside every risk estimate (Technical Report III (Section "The signed panel and pair-specific losses")).

Developability. Developability is a vector of measurements: expression, monomeric purity, thermal stability, self-association, polyreactivity, viscosity, chemical stability and immunogenicity risk. Reference distributions come from clinical-stage antibodies. Jain and colleagues expressed 137 clinical-stage antibodies as IgG1 and measured them in a dozen assays to set empirical boundaries [112]. The Therapeutic Antibody Profiler derives five computed guideline metrics from clinical-stage therapeutics, one of which is the asymmetry between heavy- and light-chain surface charge [15]. A single-domain VHH has no light chain, so that metric has no counterpart, and IgG1 boundaries do not carry over to other formats. A sequence motif only nominates a chemical liability: for Asn deamidation and Asp isomerisation in CDRs, conformational flexibility, the size of the next residue and secondary structure characterise the hotspots [16]. FLAb2 compiles public therapeutic-property datasets for expression, thermostability, aggregation, polyreactivity and pharmacokinetics [113]. In the blinded 2025 Ginkgo Antibody Developability Competition, sequence and structure predictors ranked hydrophobicity and polyspecificity better than affinity or self-association [114].

Immunogenicity. The humanness score OASis was reported to correlate with clinical immunogenicity [18]. NetMHCIIpan predicts peptide binding to MHC class II [19], yet an industry commentary names the missing correlation between predicted epitope content and anti-drug-antibody outcome as a clear limit on its use [115]. Scans cover the final assembled construct, because a junction creates peptides that neither domain contains.

Glycans. Glycans act on both sides of the interface. On the antigen, mobile glycans can shield a large share of the protein surface [13], so an epitope that looks accessible on a bare structure may be blocked on the glycoprotein. On the binder, an N-linked glycan in the variable domain can raise or lower affinity depending on its position [116] and adds chemical heterogeneity [117]. A designed binder with an Asn-X-Ser/Thr sequon (X is any residue except Pro) therefore needs a sequon-free sequence or measured occupancy before its contacts are read (Technical Report III (Section "Liability, immunogenicity and glycan channels")).

What a benchmark reports. Two hit rates can be compared only if both state the targets, with their identifiers, deposit dates and similarity to the training sets of generator and judge. Both must also state the cut-off date, the model it is measured against, and the numbers of designs generated, scored and tested. They also state the judge (model, version, seeds, construct, score and threshold) and the success rule (assay, construct, concentration, threshold and replicate rule). Controls include shuffled and non-cognate decoys, an antigen swap and a reference binder. Specificity needs a panel manifest, the assay and the polyreactivity readout, and developability needs assays and references in the tested format. The statistics state the resampling unit, the interval and the number of targets. With 30 targets, the normal-approximation 95% margin near a 50% hit rate is about 18 percentage points. Any comparison with a published number states which of these differences apply and that the panels are not matched (Technical Report III (Section "Evaluation design")).

6. Gaps, aims and scope

Gaps in the published record. Generators of antibodies and nanobodies exist, and most report laboratory tests (Section 4). Among the systems we read, four features of the record bear on this study. The first is the framework. Twelve of the 19 systems in Table 2 keep it fixed outright, and several draw it from a short list of scaffolds. IgGM takes the framework sequence from the user [76], TFDesign-sdAb mutates framework residues of a known antibody [93], and the two systems that generate the framework, AbDiffuser and IgDiff, take no antigen [73, 74]. The primary texts report no epitope-conditioned generation of the framework, the CDRs and their placement together, and no comparison of fixed and generated frameworks at matched compute.

The second is the generator. La-Proteina and Proteina-Complexa report no antibody result [68, 71] (Section 3.1), and UniMoMo, a latent diffusion model that includes antibodies among its benchmark domains, reports computational benchmarks only [94]. The third is specificity. No primary text reports a self-proteome panel, the specificity tests that exist are narrow, and co-folding confidence is a weak pair-specific oracle (Section 5.1). Odin-Multi treats cross-reactivity as a design objective and SurfDiff ranks epitopes by selectivity [110, 111].

The fourth is evaluation. Reported hit rates come from different targets, cut-off dates, numbers of designs, judges and success rules, and no shared benchmark connects the epitope-conditioned systems (Section 4.1). CHIMERA-Bench retrains earlier CDR generators on shared splits and

covers CDR generators only [99]. Training compute is rarely stated, and Chai-2, JAM-2† and Latent-X2 are available only through partners or a platform.

Aims. This study has three aims. The first is to specify an unconditional generative base model, with its data format, autoencoder, denoiser, training procedure and compute rule, in enough detail to repeat the work, and to supply its code. The second is to specify the full system that would turn the base model into an antigen-conditioned design tool for VHH and Fv binders, with a free framework, typed evidence channels and a fixed-size selection. The third is to fix the evaluation design in advance, so that a later result is read against a protocol set beforehand: four leakage axes, a temporal holdout, a rule for comparison with published numbers, and the hypotheses H1 to H6 (Technical Report III (Section "Hypotheses, ablations and validation levels")).

Scope. This paper specifies a system and releases the code of its base model. The base model is an autoencoder and a flow-matching denoiser configured for training from random initialisation on an unconditional task (Section 7). It is implemented, and its code is supplied. Antigen-conditioned generation, guidance, maturation, judging, liability channels and portfolio selection are specified as the design of the full system. The work is computational and contains no laboratory step. The binding, specificity, safety and accuracy of a designed molecule are outside its scope. Other systems enter through their published numbers only, and a comparison states the protocol differences between the studies (Technical Report III (Section "Comparison with published numbers")). The Supplementary Information holds the mathematical derivations, the data tables, the evaluation protocol and the explanation of the code.

7. System overview

Omniverso is a computational system for the de novo design of antibody and nanobody binders against a registered antigen and epitope. The design connects molecular generation, decoding into sequences and structures, evaluation in typed channels and the selection of a fixed-size portfolio. It ties every output to the data, the model state, the numerical settings and the evaluation procedure that produced it. All outputs are computational predictions.

Scope. This paper specifies the full system and releases the code of its base model. The base model (cycle 1) is an autoencoder and a flow-matching denoiser, configured for training from random initialisation on an unconditional task. It takes no antigen, no hotspot and no format token, and loads no released weight. The full system adds antigen conditioning, guidance, maturation, judging, liability channels and portfolio selection. These parts are specified as a design, and the code package covers the base model. The binding, specificity, safety and accuracy of a designed molecule or of the model are outside the scope of the manuscript.

Figure 1 draws the antigen patch and illustrative rigid-body pose decoys on one deposited complex (panel a) beside four polar interface anchors on the native VHH (b).

Base model. The autoencoder maps each residue to a deterministic latent of 8 numbers. Its encoder reads 169 numbers per residue: centred atom37 coordinates (111), an atom mask (37) and a residue category (21). Its decoder reads the C-alpha coordinates (3) and the latent (8), and returns atom37 coordinates and residue logits. The shards store coordinates in ångströms. The code centres each complex and multiplies the coordinates by 0.1, so the networks, the noise and

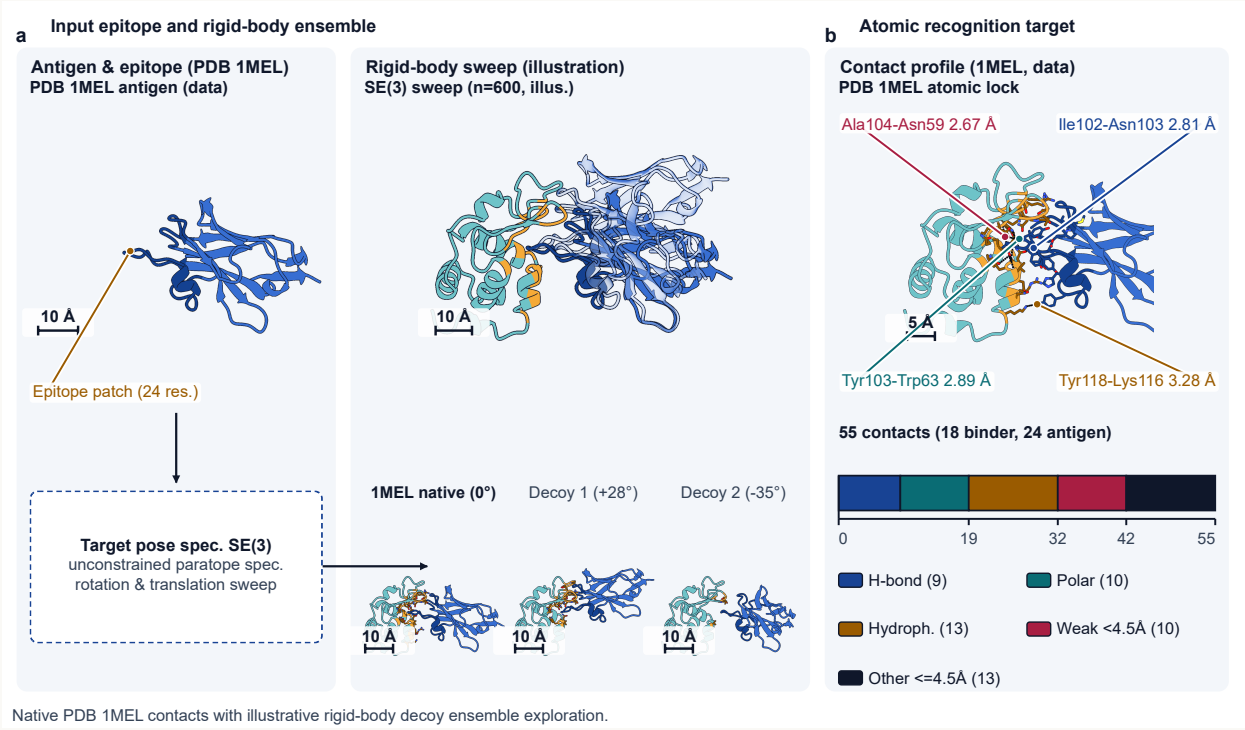

Figure 1: An antigen contact patch, illustrative pose decoys and native polar anchors define the binder design problem on PDB 1MEL. a, Deposited hen egg-white lysozyme (chain L, teal) with its 24-residue 4.5 Å contact patch (amber) bound to VHH cAb-Lys3 (chain A, blue) across three illustrative rigid-body poses: native (0°) and two rotated decoys (+28° and −35°). **b,** Close-up of the PDB 1MEL VHH interface marking four polar anchors (Ala104–Asn59 at 2.67 Å, Ile102–Asn103 at 2.81 Å, Tyr103–Trp63 at 2.89 Å, and Tyr118–Lys116 at 3.28 Å) alongside the 55 interface pairs split across CDR-H3 (41), CDR-H1 (9), and CDR-H2 (5). Coordinates and contact counts come from PDB 1MEL; decoy rotations and candidate pose status labels illustrate the scoring task without experimental measurement. CDR, complementarity-determining region; PDB, Protein Data Bank; VHH, variable domain of a heavy-chain-only antibody.

the coordinate losses work in nanometres (Section 8). Encoder and decoder each have 12 blocks of width 768, and the autoencoder has 255,873,676 parameters (encoder 127,949,832, decoder 127,923,844).

The denoiser learns a velocity field on the flow state, which holds 11 numbers per residue: the C-alpha coordinates (3) and the latent (8). It has 14 blocks of width 768 with 12 heads, a pair stream of width 256 and no triangle layers. With a pair update in every layer it has 153,102,091 parameters. The reference denoiser of the same family has about 159 M parameters [118], so the count here is 3.7 percent lower. The construction follows partially latent flow matching [68]: coordinates and latent are transported from noise to data on independent times. The base model is a design of the same kind as the published networks; a comparison of its architecture and quality with theirs is outside the scope of this paper.

Training has two stages. The encoder and decoder train first. The encoder is then frozen, and the denoiser trains on its latents (Section 12). Training losses describe training behaviour and are not measures of model quality (Technical Report III (Section "Evaluation design")).

Full system. The full system is designed to turn the base model into a campaign tool. A campaign defines the antigen, the epitope, the antibody format, the movable and sequence-design regions, a specificity panel and the size N of the final portfolio. The generator draws many molecular proposals, each as one numerical state. Independent channels then answer different questions about each decoded proposal. The selector returns a set of exactly N candidates, or the typed outcome `INFEASIBLE_PORTFOLIO` when no set meets the constraints.

The base model is specified in Section 8 to Section 12. The parts that only the full system has are the immunoglobulin contract (Section 9), antigen conditioning and Interface-JEPA (Technical Report II (Section "Antigen conditioning and Interface-JEPA")), guidance and sequential Monte Carlo (Technical Report II (Section "Guidance and sequential Monte Carlo")), maturation (Technical Report II (Section "Maturation and editing")), the judge and cross-reactivity instrument (Technical Report III (Section "Judging and specificity")), the liability, presentation and glycan channels (Technical Report III (Section "Liability, immunogenicity and glycan channels")), the decision layer (Technical Report III (Section "Decisions under ambiguity and portfolio selection")) and the evaluation design (Technical Report III (Section "Evaluation design")).

Design inputs. Six design inputs act through generation, steering and selection: the antigen, its topology, its physicochemistry, its surroundings, the shape of the paratope and the regime of the self-proteome panel. The first five enter the generator, and the self-proteome panel acts through steering and selection (Technical Report II (Section "Antigen conditioning and Interface-JEPA")).

Typed outputs. Each biological question has its own output type. The eight types are the pose score, the epitope metrics, the pair-binding probability, the affinity value, a generic polyspecificity readout, a named off-target result, a chemical liability and a predicted MHC-II presentation load. Each answers one question about one required object, for example one antibody-antigen pose or one named antibody-antigen pair. Types are validated before ranking or aggregation, and no report converts one type into another.

Evidence channels. The design condition (what the campaign asked the generator to satisfy), the model prediction (what a computational judge reported) and the experimental observation (what a physical measurement recorded) stay distinct. A report marks a missing measurement as missing.

Formats. VHH de novo is the first format admitted. Paired Fv de novo is added when its gates pass, and editing is inactive in version 1. A component returns a typed failure when its inputs are absent and never substitutes another dataset, model, numbering scheme, threshold or unit.

8. State and geometry

A molecule enters the model as a list of residues, and each residue carries a fixed set of numbers. The base model reads one protein complex at a time, with no antigen, no hotspot and no format token. Figure 2 draws the three nested residue masks on one deposited nanobody fold alongside their Euler scope counts (panel a) and per-residue sequence tracks with the CDR-H3 interface closeup (panel b).

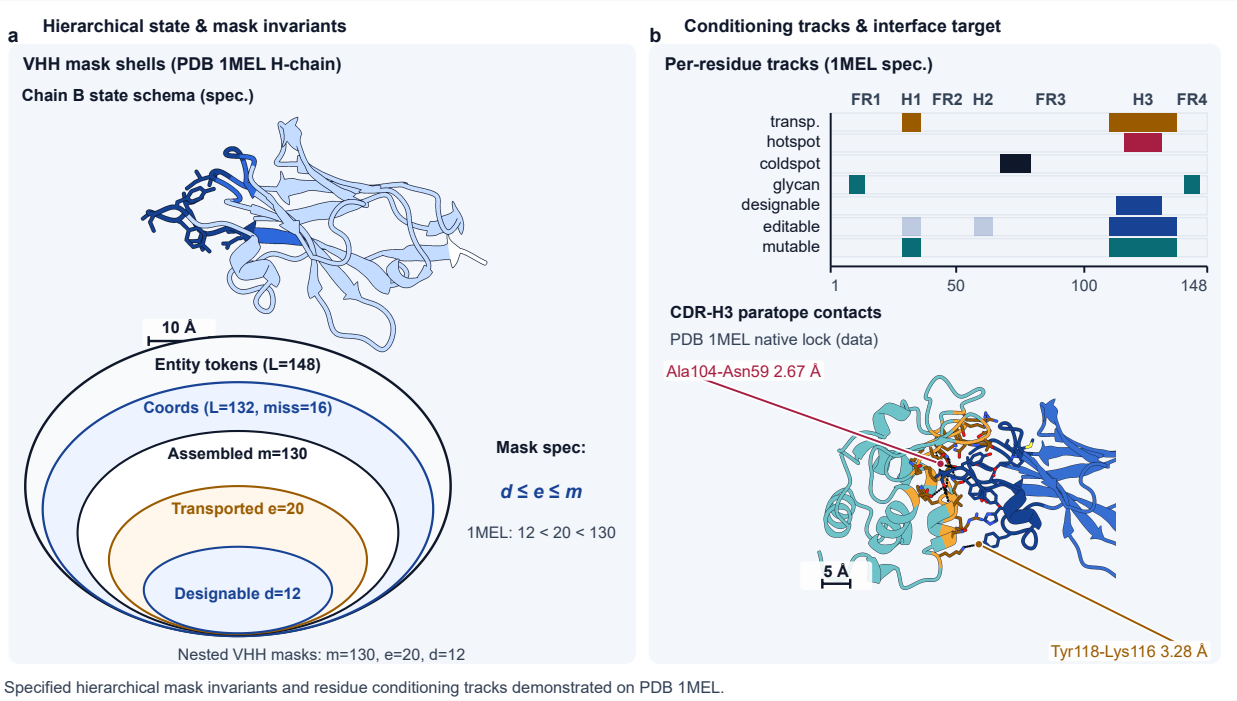

Figure 2: Three nested residue sets partition PDB 1MEL chain A between fixed context, mobile state and sequence-mutable design positions. a, Deposited VHH structure of PDB 1MEL chain A beside its nested Euler scopes: entity length $L = 148$, resolved coordinate rows (132), assembled mask $m = 130$, transported state $e = 20$, and mutable residues $d = 12$ obeying $d \preceq e \preceq m$. **b,** Seven aligned per-residue tracks (observed states, CDR1–3 windows, m, e, and d) across FR1 to FR4 alongside the PDB 1MEL CDR-H3 lock (Ala104–Asn59 at 2.67 Å). Coordinate availability comes from PDB 1MEL; example counts $m = 130$, $e = 20$ and $d = 12$ illustrate the full-system mask schema, whereas the implemented base model sets $e = d = m$. CDR, complementarity-determining region; FR, framework region; PDB, Protein Data Bank; VHH, variable domain of a heavy-chain-only antibody.

Residues and atoms. Let $L_{\max}$ be the padded number of residue rows in one batch entry. Rows beyond the real length hold zeros and are excluded by the masks below. Each residue has a category s coded 0 to 20, a block of atom coordinates and a latent vector $z \in \mathbb{R}^8$. Atoms follow the Atom37 layout of La-Proteina [68]. Each residue owns 37 atom slots, and a slot holds three coordinates and a presence flag M_{atom}. The C-alpha atom is slot 1 and is present in every residue. An absent atom has coordinates of zero after centring.

The coordinate block has $37 \times 3 = 111$ numbers per residue, and the category is a one-hot vector of width 21. Coordinates are stored in ångströms in the shards and used in nanometres inside

the model. The chain index and the residue index identify each residue uniquely. The encoder computes the latent.

Encoder, decoder and flow state. The encoder reads, for each residue, the 111 centred coordinates in nanometres, the 37 presence flags and the 21-wide category vector, which is 169 numbers. It is deterministic: one input gives one latent $z_i \in \mathbb{R}^8$ per residue, computed in FP32. The decoder reads the C-alpha position in nanometres and the latent, $3 + 8$ numbers, and writes 37×3 coordinates in nanometres and 21 category logits. The flow transports only the C-alpha position and the latent of each residue,

$$q_{\text{raw},i} = (x_{\text{C}\alpha,i},\, z_i), \qquad x_{\text{C}\alpha,i} \in \mathbb{R}^3, \quad z_i \in \mathbb{R}^8,$$

which is 11 numbers per residue. The decoder is trained to rebuild the other atoms and the category from these 11 numbers. The latent channels are learned and carry no assigned chemical meaning. The denoiser also receives the chain index and the residue index of every row. It encodes the index with sinusoids and adds a pair feature, the signed index difference clipped to -128 through 128, with one extra category for residues on different chains.

Masks and the state schema. Three nested masks select residues:

$$d_{\text{res}} \preceq e_{\text{res}} \preceq m_{\text{res}}, \qquad d_{\text{res}},\, e_{\text{res}},\, m_{\text{res}} \in \{0,1\}^{L_{\max}}.$$

The symbol $\preceq$ compares two masks entry by entry, and the sums of the masks are n_d, n_e and n_m. The assembled-residue mask m_{res} marks the residues of the assembled molecule, and rows outside it are padding. The transported mask e_{res} marks the residues of the transported state, and only these move. The design mask d_{res} marks the residues whose category may change.

A residue in e_{res} and outside d_{res} moves while its category stays clamped. A residue in m_{res} and outside e_{res} is context: it is stored, carries zero velocity and zero loss, and adds no dimension to the state. Such a residue reaches the model through the condition c. Figure 2a and b draw the three masks on one VHH domain with example counts $n_m = 130$, $n_e = 20$ and $n_d = 12$. The rules that derive the masks from IMGT regions are in Section 9.1.

The base model sets $e_{\text{res}} = d_{\text{res}} = m_{\text{res}}$ for every complex. Every real residue moves and every category is decoded, so the code stores only m_{res}, as a padding mask. Clamped framework residues need $e_{\text{res}} \subsetneq m_{\text{res}}$, which the full system introduces.

The random state is packed from the rows of e_{res} alone. The padded arrays are storage. Packing gathers the selected rows into one vector, and scattering returns the updated rows to their places beside the fixed rows. Fixed and padding rows therefore never become random dimensions. The padded length, the three masks and a coordinate profile g (next subsection) form the state schema

$$\mathcal{S}_{\text{state}} = (L_{\max},\, m_{\text{res}},\, e_{\text{res}},\, d_{\text{res}},\, g).$$

The schema, the generation condition c and the packing order together fix the probability space. Conditioning variables stay outside the transported density.

8.1. Rigid motions and coordinate gauges

A rigid motion applies one rotation $R \in \mathrm{SO}(3)$ and one translation $t \in \mathbb{R}^3$ to every atom, $x_i \mapsto Rx_i + t$. The set of such motions is the group SE(3). Reflections are excluded because a mirror

image turns L-amino acids into D-amino acids. A quantity is invariant when a rigid motion leaves it unchanged, as it does an interatomic distance. A quantity is equivariant when it moves with the molecule, as a displacement vector does. Physical laws give no preference to a laboratory frame, so a gauge, a rule that fixes the frame, gives one physical object one numerical form.

Two situations need different gauges. For an isolated molecule, global translation is redundant and the gauge removes it. When an antigen is present, the position of the antibody relative to the antigen is the quantity being designed, and the gauge keeps it. Three coordinate profiles result. In the centred frame of the base model, positions are expressed relative to the atom centroid, the coordinate dimension is $d_x = 3n$ for a complex of n residues and the state dimension is $D = 11n$. The unanchored monomer profile and the context-anchored profile are used by the full system, and n_e denotes the number of transported residues.

The unanchored monomer profile applies when $e_{\text{res}} = m_{\text{res}}$ and the solve has no antigen, fixed framework or external frame. It removes the centroid of the transported C-alpha positions:

$$y_x = (B_e^\top \otimes I_3)\ \mathrm{vec}(x_{\mathrm{C}\alpha,e}), \qquad d_x = 3n_e - 3.$$

Here $x_{\mathrm{C}\alpha,e}$ holds the n_e transported C-alpha positions and vec stacks them into one column. The matrix B_e has n_e rows and $n_e - 1$ orthonormal columns that span the residue patterns summing to zero. I_3 is the 3×3 identity, and the Kronecker product $\otimes$ applies the same centring to the x, y and z components. A registered Helmert or QR construction with fixed column signs makes B_e deterministic. In a one-dimensional example, three residues at 1, 4 and 7 Å have a centroid of 4 Å and centred positions of -3, 0 and 3 Å. Three numbers carry two degrees of freedom.

The context-anchored profile serves every solve with an antigen or a fixed framework, and every solve with $e_{\text{res}} \subsetneq m_{\text{res}}$. A registered context frame supplies an origin $a_c \in \mathbb{R}^3$ and a rotation $R_c \in \mathrm{SO}(3)$. Each transported position becomes

$$y_{x,i} = R_c^\top (x_i - a_c), \qquad d_x = 3n_e.$$

The origin is subtracted first, and the result is rotated into the axes of the frame. The relative placement of antibody and antigen stays in the state. If antibody and context move together, a_c and R_c move with them and $y_{x,i}$ is unchanged (proof in Supplementary Section S1). The frame identifier and the rule for degenerate epitope geometry are recorded with the campaign.

The base model uses neither chart. The shard reader returns coordinates as stored, in ångströms, and the model code centres every complex whenever a batch is used. The reference point is the mean position of all present atoms of the complex. Centring runs in FP32 in ångströms, and the code then multiplies the coordinates by 0.1, so the model works in nanometres. The atom centroid differs slightly from the C-alpha centroid, so the C-alpha positions have close to, but generally not exactly, zero mean.

The coordinate noise of the base model is a Gaussian with a standard deviation of 1 nm per coordinate, plus one translation per complex with a standard deviation of 0.2 nm (2 Å) per axis. The shared translation follows the training recipe of Proteina-Complexa, which gives it in nanometres. That recipe uses it to place a binder at an interface and describes it as irrelevant for monomer generation [118]. No likelihood is computed in the base model, so the full-dimensional support that the reduction by three secures for densities does not arise there.

The base-model networks have no symmetry guarantee. The denoiser projects raw coordinates into its feature stream and builds its pair features from distances, which are invariant. The architecture is not equivariant by construction, and the code applies no rotation augmentation. The full system requires the velocity field to satisfy $f_u(g \cdot x,\, g \cdot c) = \rho_g f_u(x, c)$, where ρ_g rotates the velocity of every residue by R and $u \in [0, 1]$ is the master clock of the flow. With an invariant base law, such a field gives a flow that commutes with every rigid motion, so moving the antigen moves the generated antibody with it (proof in Supplementary Section S1). Equivariance is a consistency condition on the model and is not evidence of binding.

Scaling and the dimension of the state. Coordinates arrive in ångströms and latents have learned scales. The full system converts both to dimensionless numbers, so that one loss can weigh them:

$$\bar{y}_x = \frac{y_x}{\ell_0}, \qquad \bar{z}_{ir} = \frac{z_{ir} - \mu_{z,r}}{\sigma_{z,r}}.$$

Coordinates are divided by a registered length $\ell_0 > 0$ in ångströms. For channel r of residue i, the latent has the training-set mean $\mu_{z,r}$ subtracted and is divided by the training-set standard deviation $\sigma_{z,r}$. With $\ell_0 = 3$ Å, a coordinate of 6 Å becomes 2. A latent value of 5 with mean 3 and standard deviation 1 also becomes 2. The packed dimensionless state is $\bar{q} = (\bar{y}_x, \bar{z}_e) \in \mathbb{R}^D$. Its default base law is a standard Gaussian in each block, $\bar{y}_{x,0} \sim \mathcal{N}(0, I_{d_x})$ and $\mathrm{vec}_e(\bar{z}_0) \sim \mathcal{N}(0, I_{8n_e})$.

The base model applies one fixed length scale. The factor of 0.1 from ångströms to nanometres equals $\ell_0 = 10$ Å, so a coordinate of 6 Å enters the model as 0.6 nm. The latent is not standardised, which equals $\mu_z = 0$ and $\sigma_z = 1$, and the output of the encoder enters the flow unchanged. A penalty of weight 0.01 on the mean squared latent controls its scale. Each profile of the full system registers its own ℓ_0.

The state dimension adds the coordinate block to the latent block, $D = d_x + 8n_e$, so

$$D = \begin{cases} 11n_e - 3, & \text{unanchored monomer,} \\ 11n_e, & \text{context anchored or base-model frame.} \end{cases}$$

For $n_e = 130$ the latent block has $8 \times 130 = 1{,}040$ numbers. The unanchored coordinate block has $3 \times 129 = 387$, so $D = 1{,}427$. The anchored block has 390, so $D = 1{,}430$. For $n_e = 20$ the anchored state has $D = 220$. Rows in m_{res} outside e_{res} add nothing. The base model pads complexes to 300, 400 and 500 residues, so a full bucket holds 3,300, 4,400 and 5,500 flow numbers per complex. These figures give storage size. Padded rows are masked, so they add nothing to the state dimension. Whether 8 latent numbers per residue keep enough side-chain and category information is a property of the trained autoencoder and is assessed with the reconstruction diagnostics of Technical Report III (Section “Evaluation design”).

9. Immunoglobulin contract

The immunoglobulin contract is the set of rules that the full system applies to every antibody or nanobody that it reads or generates. The base model trains on an unconditional task and does not number, mask or admit residues by these rules.

9.1. Numbering, regions and the CDR3 prior

Numbering record. Every variable domain keeps three coordinate systems. The raw index is the zero-based offset of a residue in the submitted sequence. The tool position is the pair (base number, insertion letter) that the numbering tool returns. The IMGT label is the canonical site name, with insertions written as decimal ranks. The raw index shifts with every insertion, whereas an IMGT label names the same site in every domain. The tool output is kept, and normalisation appends a field and the converter version.

Numbering tools. The numbering of record comes from ANARCII, version 2.0.8 [119]. ANARCI [10] with HMMER runs as an independent cross-check and keeps its own record. We set the independent ANARCI/HMMER cross-check threshold to a bit score of 100. The cross-check does not replace the numbering of record, and a difference between the two position maps is stored as a discrepancy. The IMGT scheme [9] annotates 128 positions [10]. Curation rejects and counts every record that cannot be numbered (Section 10.2). The supplied support code (Supplementary Section S4) calls ANARCI with the package defaults to read CDR ranges, and it applies neither ANARCII nor the bit-score rule.

Insertions. The third CDR holds 13 positions, 105 to 117, in the basic scheme. A longer loop receives extra positions between 111 and 112, created in the order 112.1, 111.1, 112.2, 111.2 and so on. The numbering output writes them as letters, such as (112, A). The converter reads a letter as a spreadsheet-style rank (A is 1, Z is 26, AA is 27), so (112, A) becomes 112.1 and (111, B) becomes 111.2. The allocation rank, the order of creation, is

$$\mathrm{alloc}(112.r) = 2r - 1, \qquad \mathrm{alloc}(111.r) = 2r.$$

Four extra residues read 111, 111.1, 111.2, 112.2, 112.1, 112 from the N terminus. Creation order differs from chain order, so records are sorted by chain order; a lexical sort would reverse the right-hand labels. An insertion syntax outside the registered roster returns `numbering_unavailable`, and a round-trip test requires that decoding an encoded position returns the position.

Numbering audit. Gate F-IMGT audits the numbering caller on at least 600 independently annotated solved domains (VH, VL, VHH and documented insertion cases). A domain passes when its position map, region map and conserved anchors all agree with the reference. Acceptance needs a one-sided 95 percent exact-binomial lower limit of 0.995 on the pass fraction, no disagreement outside the documented conventions, and no mask index outside the numbered domain. With 600 domains the lower limit is $0.05^{1/600} = 0.99502$ when all pass and 0.9921 with one failure, so a single failure fails the gate at that size. The thresholds are engineering settings.

Reference identity. For a protein sequence, IgBLAST [120] aligns the V region to translated germline references. The record stores the best reference and the aligned identity, which is the fraction of aligned V-region positions where the two residues match, under the declared gap policy. The result is typed `nearest_translated_V_reference`. An edit toward that reference is named `aa_reference_reversion`. The name `germline_allele_reversion` is reserved for assignments resolved from nucleotide sequence.

Regions and masks. Masks are built from IMGT labels and chain identifiers, never from raw indices (Table 3) [9]. Conserved anchors sit at fixed positions, among them cysteine 23, tryptophan 41, cysteine 104, and phenylalanine or tryptophan 118.

Table 3: IMGT framework (FR) and complementarity-determining (CDR) regions used by the region masks. Positions are IMGT numbers; the counts sum to 128 before insertions.

Region	IMGT positions	Number of positions
FR1	1 to 26	26
CDR1	27 to 38	12
FR2	39 to 55	17
CDR2	56 to 65	10
FR3	66 to 104	39
CDR3	105 to 117, plus insertions	13 plus insertions
FR4	118 to 128	11

The labels select residues into the three nested masks $d_{\text{res}} \preceq e_{\text{res}} \preceq m_{\text{res}}$ for design-mutable, transported and assembled residues (Figure 2, panel b; Section 8). When a profile names a position that the sequence does not contain, the profile returns `edit_profile_unavailable` for that sequence. A record with a missing CDR residue is rejected at curation, so every mask is built from a complete loop.

CDR3 length prior. The prior for format p, chain c and analysis stratum r is a smoothed count of CDR3 lengths in an admitted repertoire:

$$P(\ell \mid p, c, r) = \frac{n_{pcr\ell} + \alpha_0}{\sum_{j \in \mathcal{L}_{pcr}} n_{pcrj} + \alpha_0 |\mathcal{L}_{pcr}|}.$$

Count the clonotypes of length ℓ, add a pseudo-count $\alpha_0 > 0$ to every permitted length so that none has probability zero, and divide by the total. Here $n_{pcr\ell}$ is the clonotype-weighted count, so read depth does not inflate it, and $\mathcal{L}_{pcr}$ is the set of observed, policy-approved lengths. The repertoire manifest records organism, tissue, disease status, filters, deduplication, sequencing technology, chain type and split. The prior applies when its repertoire files and count table exist. A hand-written range is typed `engineering_length_bound`. As an illustration, counts of 20, 50 and 30 at lengths 12, 13 and 14 with $\alpha_0 = 1$ give $(21, 51, 31)/103 \approx (0.204, 0.495, 0.301)$.

9.2. Scaffold profiles, cysteines and pairing

Scaffold profile. A scaffold profile is a versioned record tied to one construct. It names the format (VHH, VH, VL or paired Fv), the source construct, the residues allowed at each required IMGT position, the protected positions, the supported CDR3 lengths, the disulfide topology, the free-cysteine mode, the framework reference set and the assembly requirements. The label VHH selects a format, and the registered profile supplies the constraints. A VHH profile may protect positions that faced the light chain in a conventional VH, such as framework-2 positions 42, 49, 50 and 52 [6], or specify a disulfide between loops. A profile needs its own scaffold experiments and construct definition, and no universal site map is inferred. Gate F-VHH enumerates the whole exported panel against the profile and needs a conformance fraction of 1.000.

Cysteine table. Every decoded molecule carries a residue-level cysteine table. For each cysteine it records the chain and IMGT position, the role (canonical disulfide, profile disulfide, conjugation handle, unresolved or free), the partner position, the sulfur geometry when a structure exists, the source (parent, generated or edit) and a status. The canonical disulfide joins cysteine 23 in

framework 1 to cysteine 104 in framework 3 [9]. A profile declares one of two free-cysteine modes. In `FREE_CYS_PROHIBITED`, every cysteine outside the profile fails admission. In `CONJUGATION_HANDLE`, named positions are reserved for a stated chemistry and every other unpaired cysteine fails; this mode needs assays of accessibility, scrambling, occupancy, heterogeneity and retained function on the assembled construct. The design uses the first mode.

VH/VL pairing. Paired formats add the chain identity, the provenance of the pair, the interface pose and a pairing-likeness score. ImmunoMatch [121] supplies $S_{\text{pair}}(\text{VH}, \text{VL}) \in \mathbb{R}$, typed `pairing_likeness`. It estimates how closely a heavy and light chain resemble the paired chains used in training. Physical association, expression-level pairing and binding keep separate endpoints. ImmunoMatch was trained on paired human repertoires and reports an AUC-ROC of 0.75 on a withheld set and 0.66 on three unseen donors. Paired-Fv admission also needs an interface pose check, a clash and buried-polar analysis and, where relevant, experimental evidence of chain pairing or product purity. A single-chain construct also needs a linker profile. Gate F-PAIR tests cognate-versus-shuffled discrimination and the agreement between pairing-likeness and interface evidence.

Admission predicate. The predicate combines five checks for candidate X under profile p, which fixes the format:

$$\text{IgPass}(X; p) = N(X) \wedge R(X; p) \wedge L(X; p) \wedge C(X; p) \wedge A(X; p).$$

Here N is numbering completeness, R reference and scaffold compliance, L CDR3 length support, C cysteine topology and A the assembled-format checks. These letters are local to the predicate. A candidate is admitted only when every required numbering, scaffold, length, cysteine and assembly check is available and passes. A check that the profile excludes is left out of the conjunction, and a descriptive result is labelled as such. A missing check is never read as a pass or a fail.

Each candidate emits its numbering record, reference identity, profile and prior identifiers, cysteine table and the result of every check. A canonical-class field for CDR loops is optional and cannot gate, train or reject a design.

VHH de novo is the first format admitted. Paired Fv de novo is added when its gates pass, and editing is inactive in version 1. Until a registered profile exists, the design screens sequences before scoring with an interim filter. It needs bonded cysteines at IMGT 23 and 104 (sulfur distance 1.9 to 2.2 Å in the decoded structure), no other free cysteine, no N-X-S/T sequon (X other than proline) inside CDR1 to CDR3, and a CDR3 of 5 to 25 residues. A registered profile replaces this filter, and the repertoire count table replaces the CDR3 bound. Figure 3 draws the four structural formats alongside the disulfide and numbering gate checks.

10. Data

The base model is trained on protein structures. Source bytes arrive in quarantine. A source enters curation only after the admission check D0 passes. Curation then removes duplicate structures, numbers the antibody domains, applies structural quality control, annotates contacts, assigns each record to four leakage groups, applies the temporal holdout, freezes the splits and packs shards.

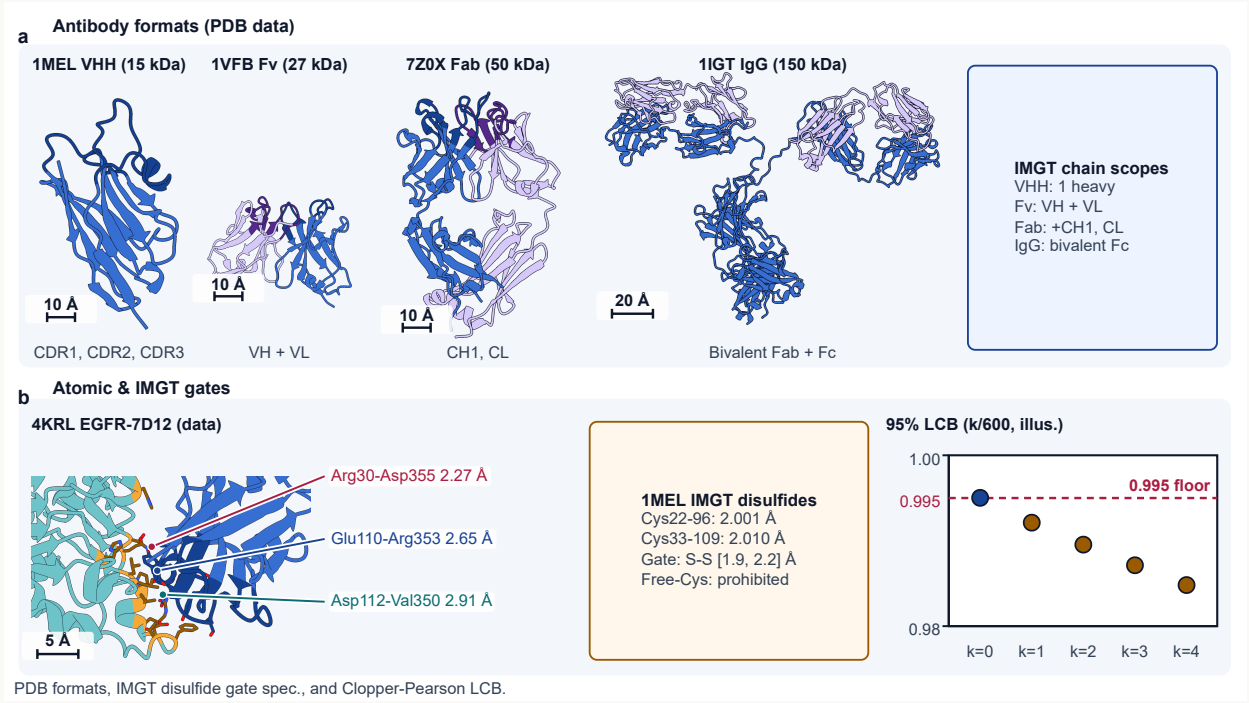

Figure 3: Four immunoglobulin formats map to canonical IMGT contracts audited against PDB 4KRL anchors, PDB 1MEL disulfides and a 0.995 numbering floor. a, Monotonically scaled reference architectures from PDB 1MEL (VHH, 1 chain), PDB 1VFB (Fv, VH+VL), PDB 7Z0X (Fab, 2 domains), and PDB 1IGT (IgG, 4 chains), aligned to IMGT CDR1 (27–38), CDR2 (56–65), and CDR3 (105–117) intervals. **b,** Geometry and numbering gates: PDB 4KRL 7D12–EGFR polar anchors (Arg30–Asp355 at 2.27 Å, Glu110–Arg353 at 2.65 Å, and Asp112–Val350 at 2.91 Å), PDB 1MEL disulfide sulfur separations (Cys22–96 at 2.001 Å and Cys33–109 at 2.010 Å) within the $[1.9, 2.2]$ Å gate, and the one-sided 95% exact Clopper–Pearson lower bound exceeding the 0.995 acceptance floor once $n = 600$ domains pass with zero failures. PDB structures and distances reflect deposited coordinates; gate curves represent deterministic statistical requirements of the full-system specification. Fab, antigen-binding fragment; Fv, variable fragment; IgG, immunoglobulin G; IMGT, ImMunoGeneTics; PDB, Protein Data Bank; VHH, variable domain of a heavy-chain-only antibody.

10.1. Sources and licences

Each dataset has one of six roles. Pretraining sets are unsupervised corpora. Supervised sets serve later tasks. Calibration sets fix a threshold that a gate reads. Benchmarks are held out from training by construction. Panels supply off-target or self background. References are term lists, annotation tables and contracts. Every training role serves a model trained from scratch, and no released weights are loaded.

The sources of the base-model corpus are structures: experimental ones from the Protein Data Bank (PDB) [122], AlphaFold models [123], and synthetic dimers built from them. The Teddymer dimers come from splitting AlphaFold DB monomers at the domain boundaries of the TED annotation [124] and clustering the pairs with Foldseek [71]. Table 4 lists the sources.

Table 4: Sources of the base-model corpus. Counts are numbers of structures, clusters or representatives as reported by each source or by [71]. The licence is the one stated by the source.

Source	Content and count	Reported licence
PDB	Dated mmCIF snapshot of 2026-05-28; first biological assembly of each entry	CC0 1.0
AlphaFold DB, release v6 of 2025-09-15	241,070,489 structures	CC BY 4.0
AlphaFold DB, Foldseek-clustered subset	About 3 M cluster representatives; about 550 k after the pLDDT, length and coil filters; about 350 k after the β-sheet filter	CC BY 4.0
TED domain assignments	Domain boundaries of AlphaFold DB models	CC BY 4.0
Teddymer synthetic dimers	3,556,223 clusters before the quality filter; 510,454 representatives after interface length above 10, interface pAE below 10 and interface pLDDT above 70 [71]	CC BY-NC 4.0
PDB multimers, filtered	45,856 structures after all filters [71]	CC0 1.0

Base-model corpus. The corpus is drawn from the sources of Table 4 through a filter ladder that the corpus manifest records. The temporal cut-off T^* applies to PDB entries by deposition date, and AlphaFold models and TED assignments carry no PDB deposition date. The code package also contains a generator of synthetic complexes that passes through the same reader and carries no biological content. Each complex is a seeded random walk of C-alpha positions with a step of 0.7 Å per axis. Atoms scatter 0.15 Å about the C-alpha, 8 percent of the other atom slots are masked, and categories are drawn at random from 0 to 20.

The antibody-specific datasets of the full system are SAbDab [22], SNAC-DB [103], Chimera-Bench [99], FoldBench [104], the Observed Antibody Space [21], IMGT/GENE-DB, CoV-AbDab, Thera-SAbDab, PLAbDab, SKEMPI 2.0, the polyspecificity reagent corpus [109], Absolut! [125] and the Antibody-Nanobody Design Dataset (ANDD) [126]. Supplementary Section S2 gives their counts, versions, roles and licence classes, together with the cut-off dates of published systems.

Licences were read from the release page of each source on 5 and 6 October 2026. Terms attach to the dataset, so an article under CC BY does not make its dataset CC BY. The AGPL-3.0

licence of the code grants nothing on a dataset. Sources under CC0 1.0 or CC BY 4.0 are admitted with attribution. They are the PDB, AlphaFold DB, TED, SAbDab, OAS, IMGT/GENE-DB, UniProt and its reference proteomes, Human Protein Atlas v25.1, SKEMPI 2.0, CoV-AbDab and the polyspecificity reagent corpus. OAS gathers data from many studies, and the terms of each study apply.

Four classes are excluded from the training lineage of weights that may be released. Non-commercial terms apply to Teddymer, Absolut! data, ASD v2, AbAgym, INDI2, PairedAbNGS, AbNatiV and NanoMelt. No-derivative or share-alike terms apply to SURFY, UniCarbKB, OMA, OpenCell images and the Adaptyv competition data. SNAC-DB reserves training to its authors. Custom or paid terms apply to GDPa1, PDBbind, NetMHCIIpan 4.3k and the IgFold OAS models. Sources with no stated licence serve as references or evaluation material until written terms exist. They are AB-Bind, Mason 2021, the Absolut! repository, TEMPRO, the Jain supplement, PLAbDab, PLAbDab-nano, Thera-SAbDab, cAb-Rep and FoldBench (whose article carries CC BY-NC-ND terms).

Training data for weights that may be released must lie under CC BY 4.0, CC0, MIT, Apache 2.0 or public-domain terms. Before a source is used, its licence is reviewed for the object and for the action (training, evaluation or redistribution). The review ends in one of three outcomes (allow, restrict or blocked) and is valid only with the digest of the governing terms. Every weight file carries a lineage record with its datasets, their licences and their file hashes.

10.2. Curation, quality control and deduplication

For each dataset, record its release, retrieval date, licence terms, inclusion criteria, deduplication rule and retained counts by split. Distinguish counts reported by the source from counts obtained after acquisition and filtering. For deduplication, the input count equals the retained count plus the number of duplicate exclusions.

D0 halts curation of the source on a checksum or schema failure, on a licence decision that does not permit the action, on a count that does not conserve, on a substituted snapshot and on a derivation that disagrees with its filter. Changed upstream bytes produce a new digest and stop D0.

Quality control decides whether a record may enter a contact annotation, a mask or a pretraining example. The rules are fixed before use and bound to a content digest. Records are kept at a resolution of 4.0 Å or better when they serve as supervised or benchmark rows or enter the contact annotation of PDB records. Geometry used only for pretraining is recorded separately. The R-free value is recorded, and no rule uses it as a gate. Atoms enter the 4.5 Å contact set only above an occupancy of 0.50, in all sources. Among alternate locations the one with the highest occupancy is kept.

Missing density that touches the interface rejects a supervised or benchmark row. In pretraining-only rows, where no interface is read, the gap is modelled. A missing CDR residue rejects the record in every source. All antibody records are numbered with the IMGT scheme, and a numbering failure rejects the record, is counted and keeps the invocation.

The numbering of record comes from ANARCII, version 2.0.8 [119]. ANARCI [10] with HMMER runs as an independent cross-check with its own record. A record whose cross-check alignment has a bit score below 100 is rejected and counted, and a difference between the two position maps is

stored as a discrepancy. Section 9 describes the numbering record. Each rejection raises a manifest counter and creates a discrepancy record, and the accepted count enters the conservation check.

Contact annotation applies a 4.5 Å atom-to-atom rule between antibody and antigen atoms whose occupancy passes the gate. It emits a contact-set bitmap per antigen and a rectangular antibody-to-epitope contact matrix. The bitmap seeds the A3 leakage axis below, and the matrix seeds the contact loss of the full system.

Cross-source deduplication. Several structural sources are views of the PDB and may share depositions: SAbDab, SAbDab-nano, SNAC-DB, AbDb, AACDB, Chimera-Bench, FoldBench, the Smorodina panel and the Hitawala–Gray benchmark. Deduplication runs before clustering, so that one deposition receives one set of group identifiers. A structure-derived record is reduced to the key

$$k = \mathrm{SHA256}\big(pdb \parallel asm \parallel Ab \parallel Ag\big),$$

where *pdb* is the entry identifier, *asm* the biological-assembly identifier, and *Ab* and *Ag* the sorted author chain identifiers of the antibody and antigen chains. The key is computed on the biological assembly of the admitted PDB snapshot and never on the identifier of a source. Chain identifiers are normalised to the snapshot, so a renaming by a curator does not change the key. A record without a structure is keyed on the SHA-256 of its numbered VH and VL sequence pair.

Two orders resolve a shared key. Annotation precedence decides which source supplies the record: SNAC-DB, then SAbDab and SAbDab-nano, then AbDb, then AACDB, then the raw PDB snapshot. SNAC-DB leads because its assembly-based curation resolves multi-chain epitopes. It reports multi-chain antigens in 33.5 percent of its complexes against 15.3 percent in its SAbDab comparison [103]. Its records serve for training only where its terms permit that use.

Partition precedence decides which split a shared key lands in, and it overrides annotation precedence. The order is benchmark and temporal holdout first, then calibration, then training. A key claimed by any admitted benchmark is removed from every training and pretraining manifest and from the calibration designs. Each realised manifest lists, for every other source, the shared keys before precedence and those removed by it, with the query that produced them. D0 requires zero benchmark overlap with training after precedence.

10.3. Leakage axes, temporal holdout and split freezing

Leakage means that information about a test record reaches training by a route that the split does not cut. The design uses four axes: antigen sequence (A1), antigen structure (A2), the set of antigen residues in contact with the antibody (A3) and the CDR-H3 sequence (A4). Whole-Fv identity is left out because conserved frameworks dominate that comparison while the antigen-contacting loops may still differ. Table 5 gives the rule and threshold of each axis.

A3 uses the Jaccard index of epitope residue sets after mapping them into a shared antigen residue coordinate system, with contacts defined at 4.5 Å, a threshold of 0.30 and single-linkage clustering. The Jaccard index of two epitopes is the size of the intersection of their residue sets divided by the size of the union. An A3 comparison is defined only when that mapping specifies antigen equivalence, reference alignment, insertion handling, unmapped residues and multichain correspondence. A split cannot be declared A3-disjoint where a required comparison is undefined. The value 0.30

Table 5: Leakage axes: clustering rule and threshold. Identity and coverage are in percent; the Jaccard index and the TM-score are dimensionless.

Axis	Object	Rule	Threshold
A1	Antigen sequence	MMseqs2 clustering [127]	Identity at least 40 percent, coverage at least 80 percent
A2	Antigen structure	Foldseek clustering [128]	TM-score at least 0.50, coverage at least 80 percent
A3	Epitope residue set	Jaccard index over mapped antigen residues within 4.5 Å of the antibody, single-linkage	Jaccard index at least 0.30
A4	CDR-H3	MMseqs2 clustering over IMGT positions 105 to 117, single-linkage	Identity at least 80 percent, coverage at least 80 percent

is a setting chosen for discontinuous epitopes. Chimera-Bench groups only identical residue sets, which is a Jaccard index of 1 [99]. The clustering commands for A1, A2 and A4 are:

```
# A1, antigen sequence
mmseqs easy-cluster antigens.fasta clust/ag_seq tmp/ag_seq \
  --min-seq-id 0.40 -c 0.80 --cov-mode 0 --cluster-mode 0
# A2, antigen structure
foldseek easy-cluster antigen_pdbs/ clust/ag_str tmp/ag_str \
  --alignment-type 1 --tmscore-threshold 0.50 -c 0.80 -e 0.001
# A4, CDR-H3 extracted at IMGT 105-117
mmseqs easy-cluster cdrh3.fasta clust/h3 tmp/h3 \
  --min-seq-id 0.80 -c 0.80 --cov-mode 1 --cluster-mode 1
```

Coverage mode 0 measures coverage against the longer sequence of a pair, and mode 1 against the target. Cluster mode 1 joins sequences through transitive links, which is single-linkage clustering, whereas mode 0 is a greedy set cover.

Each curated record carries four labels, $(g_{A1}, g_{A2}, g_{A3}, g_{A4})$. Two records are joined if they share a label on any axis, and the split operates over the connected components of that union graph. A component is therefore never divided between test and training, and one very large component would reduce the material available for holding out.

A committed split must satisfy a disjointness predicate between a test partition T and the union C of training, pretraining and calibration partitions. Four conditions each hold under a deterministic pass over the frozen split. Every record of T was deposited on or after the cut-off T^*. No pair across T and C exceeds the A1 identity threshold on antigen sequence or the A4 threshold on CDR-H3. No pair exceeds the A2 TM-score threshold. No pair with a defined A3 comparison exceeds the A3 Jaccard threshold, and no required A3 comparison is undefined.

The conditions are conjunctive because each alone misses a route. Identity misses remote structural homologues. Structure misses high-identity sequences with unsolved structures. A date misses re-deposition of older entities.

Three further channels pass all four conditions. Splits by sequence or metadata similarity let pairs with similar interface structure cross the split [101]. A test record can sit in the pretraining corpus of an encoder that the model uses [102]. Lineages, epitope families, assay plates, campaigns and retrieval databases couple records in groups, and leakage-reduced splitting can be posed as an optimisation problem [129]. The design therefore adds interface-structure clustering and a record

of pretraining provenance for every checkpoint. The base model loads no outside weights, so the second channel does not arise there. A claim that a model generalises beyond its training records needs a prospective cohort collected after the model is frozen. Metadata checks are a necessary floor and give no guarantee.

Temporal holdout. The corpus-wide cut-off is $T^* = 2025\text{-}01\text{-}01$ by PDB initial deposition date. A record deposited on or after T^* belongs to the temporal holdout. It is excluded from every training, pretraining and calibration manifest, whichever source supplied it. Each record stores a temporal label and a component label, and a record in the holdout removes its whole A1 to A4 component from training. This is the protocol of the full system. The support code of the base model applies the temporal rule and two local checks, whole-chain identity and similarity of C-alpha distance histograms, and drops the leaking test record instead (Supplementary Section S4). The date is later than the last deposit in the released training list of Proteina-Complexa, 2024-08-02 [71], so its role is to control overlap with the disclosed training set of that system. SNAC-DB builds its public benchmark from PDB deposits of the preceding year, up to 31 March 2025, and lists the training cut-off dates of the models it evaluates [103]. Chimera-Bench and FoldBench keep their own date rules.

Split freezing. The frozen split file stores, for each record, its split, its temporal label and its four component labels. It also stores six named evaluation splits, each with its parent source and its disjointness condition: `pairing-eval-donor-disjoint` (gate F-PAIR), `tts-holdout-12ag` (F-TTS), `jepa-consequence-stream` (F-J7), `chi-ref-holdout` (F-CHI), `imgt-heldout-independent` (F-IMGT) and `glycan-contact-heldout` (F-GLY). The pairing split also carries donor identifiers, so that no donor stands on both sides. The split file is committed and hashed before training, and its digest is recorded with every run, so the exact membership of each split can be reproduced and audited.

10.4. Shard format and sampling

The packer of the full system writes a deterministic sample index with one row per sample. The row holds the sample identifier, the source-record digest, the split, the length bucket, the MSA-depth bucket, the format and coordinate profiles, the digest of the state schema, the shard identifier, the member offset and the payload hash. The payload stores full padded arrays and the three masks of Section 8. The loader assigns samples disjointly to ranks and workers, keeps a bounded shuffle buffer with a recorded generator state, conserves sample counts at epoch boundaries and restarts from a saved sampler state.

A base-model shard is an uncompressed NPZ archive of ragged complexes concatenated along the residue axis. Let N be the total number of residues in the shard and C the number of complexes. The arrays are `atom37` (float32, $N \times 37 \times 3$, in Å), `atom_mask` (bool, $N \times 37$), `aatype` (int8, N, categories 0 to 20), `chain` (int16, N), `residx` (int32, N), `offsets` (int64, $C + 1$, the row range of each complex) and `complex_id` (Unicode, C). The C-alpha sits at slot 1 of the 37 atom slots. A JSON manifest of schema 1, in ångström units, lists the file name and the SHA-256 of every shard. The reader converts nothing: shards and manifests stay in ångströms, and the model multiplies the centred coordinates by 0.1 at its entry (Section 8). The format costs 488 bytes per residue (444 for coordinates, 37 for the mask and 7 for the three index fields), so ten million residues take 4.88 GB.

The reader refuses a manifest if a shard is missing, lies outside the manifest directory or has a

different SHA-256, or if any array has the wrong type or shape. It also refuses non-finite coordinates, categories outside 0 to 20, offsets that do not start at zero or do not increase strictly, a residue without a C-alpha, a repeated pair of chain and residue index within a complex, and a repeated complex identifier. It reads every shard into memory once per rank, so a random draw never rereads a file, and the host memory needed is the shard bytes multiplied by the number of ranks plus the model and optimiser state (Supplementary Section S5).

Complexes fall into three disjoint length buckets: up to 300, 301 to 400 and 401 to 500 residues. Each bucket is padded to its upper limit, and masks exclude the padding. The global batch is 48 complexes in every update. The product of microbatch, accumulation steps and ranks gives 48 in each bucket: $8 \times 3 \times 2$ for 300, $6 \times 4 \times 2$ for 400 and $4 \times 6 \times 2$ for 500. In each update the sampler draws a bucket uniformly at random, unless a smoke schedule forces one, and then draws 48 complexes from it with replacement. The draw is seeded by the run identifier and the update index through the first 32 bits of a SHA-256 hash, so a resumed run repeats the same draws. Both ranks use one bucket per update and take separate slices of the 48 draws. A complex longer than 500 residues falls in no bucket and is never drawn, because the reference configuration applies no crop. Section 12 describes the procedure that consumes these batches.

11. Generative model

The generative model works in two steps. A velocity network moves random numbers towards the numbers that describe a real structure, and a decoder turns the result into amino acids and atom positions. Each residue carries 11 numbers. Three place its C-alpha atom, and eight are a learned latent vector from which the decoder recovers the residue type and the remaining atoms. Moving only the C-alpha atoms and the latent, instead of every atom, is the partially latent design of La-Proteina [68].

The denoiser takes its depth, width and head count from the Proteina-Complexa denoiser [71]. Both networks of the base model are implemented for training from random initialisation. The sampler, the master clock that drives it and the density formula belong to the full system, and the base model draws no sample. Figure 4 shows the straight SE(3) flow path and C-alpha distance maps on PDB 1MEL (panel a) beside the denoiser trunk and IEEE-754 accumulation precision contract (panel b). Figure 5 shows the four-tier software stack and the ångström-to-nanometre centring contract.

11.1. Probability paths and the training target

Transport. Let q be the list of D numbers that describes one complex, and let $u \in [0, 1]$ measure progress along a path. A probability path is a family of densities p_u over states. The density p_0 is a Gaussian that can be sampled, and p_1 is the density of encoded real structures. A velocity field f_u moves each state, so that a state follows the ordinary differential equation

$$\frac{\mathrm{d}q}{\mathrm{d}u} = f_u(q), \qquad q(0) = q_0 \sim p_0.$$

When the field carries p_0 to p_u, the densities obey the continuity equation, which states that probability is neither created nor destroyed:

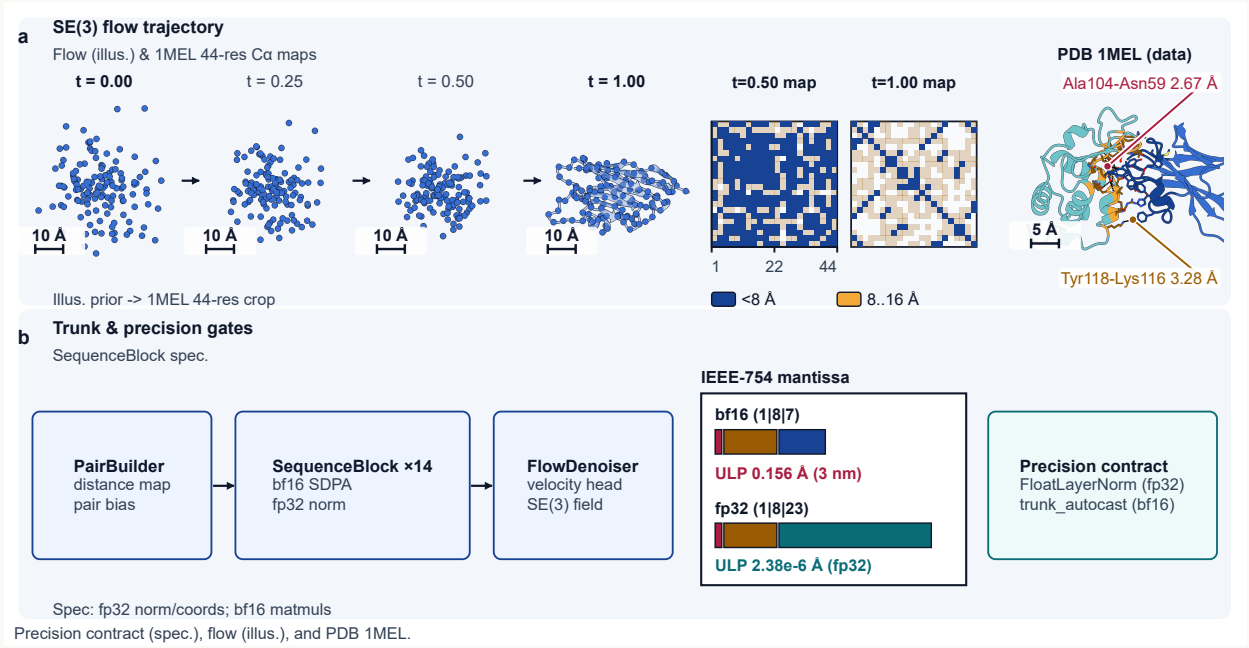

Figure 4: The partially latent SE(3) denoiser trunk couples C-alpha distance geometry with FP32 reduction arithmetic. a, Illustrative straight-path frames at $t = 0.00$, 0.25, 0.50, and 1.00 terminating at PDB 1MEL chain A, paired with computed 44-residue C-alpha distance maps at $t = 0.50$ and 1.00 and the native Ala104–Asn59 interface lock (2.67 Å). **b,** Live denoiser trunk (`PairBuilder` to `SequenceBlock` ×14 to `FlowDenoiser`, 153.1M parameters) alongside proportional IEEE-754 layouts comparing BF16 (‘1|8|7‘, 0.156 Å step at 3 nm) with FP32 (‘1|8|23‘, 2.38×10^{-6} Å step) accumulation precision. Module shapes and arithmetic policy match the live training code; trajectory frames illustrate linear interpolation toward PDB 1MEL, and no trained sampler produced them. PDB, Protein Data Bank.

$$\partial_u p_u + \nabla \cdot \left(p_u f_u\right) = 0.$$

Where the field makes neighbouring states converge, the density rises, and where it makes them separate, the density falls. This is the construction of a continuous normalising flow [130]. For $f_u(q) = -q$ and $q_0 = 2$ the solution is $q_u = 2e^{-u}$, so $q_1 = 2/e \approx 0.736$, nearby states end closer by the factor e^{-1}, and the density rises by the factor e.

Conditional paths. The intermediate densities p_u are unknown, so f_u cannot be fitted to them directly. Flow matching replaces this problem with an easier one [55]. For each data state q_1 one chooses a conditional path $p_u(q \mid q_1)$ whose velocity $b_u(q \mid q_1)$ has a closed form. Averaging over the data gives the marginal path $p_u(q) = \int p_u(q \mid q_1)\, p_{\text{data}}(q_1)\, \mathrm{d}q_1$. The field that generates it is the mean of the conditional velocities over every data state that could have produced q at progress u. The network v_θ is trained to match the conditional velocity at a sampled state:

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}\left\|v_\theta(q_u, u) - b_u(q_u \mid q_1)\right\|^2.$$

This loss has the same gradient with respect to θ as the loss on the marginal field, under the regularity conditions of the source [55]; the proof is in Supplementary Section S1. Training therefore solves no ODE. It draws a data state, a base state and a progress value, builds the intermediate state and regresses a velocity.

A Gaussian conditional path mixes the data state with noise $\zeta \sim \mathcal{N}(0, I)$ as $q_u = \alpha(u)\, q_1 + \sigma(u)\, \zeta$, with $\alpha(0) = 0$, $\sigma(0) = 1$, $\alpha(1) = 1$ and $\sigma(1) = 0$. Differentiating at fixed q_1 and ζ gives $b_u =$

$\alpha'(u)\, q_1 + \sigma'(u)\, \zeta$. For $\alpha(u) = u$, $\sigma(u) = 1 - u$, $q_1 = 2$, $\zeta = -1$ and $u = 0.5$ the state is 0.5 and the velocity is 3.

The base model uses the simplest such path. It pairs a base state q_0 with a data state q_1 and draws the line between them:

$$q_u = (1 - u)\, q_0 + u\, q_1, \qquad b_u(q_u \mid q_0, q_1) = q_1 - q_0.$$

The same interpolation applies to the correlated Gaussian base law defined below. The target is constant along each line, and the learned field is the average of this target over all pairs whose lines cross at a point. For $q_0 = (-1, 2)$, $q_1 = (3, 0)$ and $u = 0.25$ the state is $(0, 1.5)$ and the target is $(4, -2)$. A network output of $(3.5, -1)$ would give a squared error of 1.25. Panel a of Figure 4 shows the straight path on one deposited chain. Each data state is paired with an independent noise draw. Minibatch optimal-transport pairing would choose shorter pairs [60], and the design does not use it.

Flow state and base law. For one complex of n residues the state is $q = (x, z)$. The base model transports every present residue, so $n_e = n_m = n$ in the notation of Section 8. The array $x \in \mathbb{R}^{n\times 3}$ holds C-alpha coordinates in nanometres, centred on the centroid of all present atoms of the complex. The shards store angstroms, so the code centres in angstroms and then multiplies by 0.1 (Section 8). The array $z \in \mathbb{R}^{n\times 8}$ holds the latent that the frozen encoder returns for each residue, and the latent is used as the encoder returns it, with no standardisation. The state has $D = 11n$ numbers, so a complex of 130 residues has $D = 1{,}430$. The chart that removes the three global translations has $D = 11n - 3$ and is not used.

The base law is Gaussian, with independent blocks. The coordinate noise is $x_0 = \varepsilon + \mathbf{1} \otimes d$, where ε has independent standard normal entries, so its standard deviation is 1 nm per coordinate, and d is one draw per complex from $\mathcal{N}(0, (0.2\,\mathrm{nm})^2 I_3)$, added to every residue. The coordinate block therefore has covariance $(I_n + 0.04\, \mathbf{1}\mathbf{1}^\top) \otimes I_3$ in nm^2. The shared translation of 0.2 nm, which is 2 Å, is the value of the Proteina-Complexa binder model, which adds it so that the network reasons about global position and gives it in nanometres [71]. The latent noise z_0 has independent standard normal entries. The velocity targets are $x_1 - x_0$ and $z_1 - z_0$, in nanometres for the coordinate block.

The denoiser is a transformer with no built-in rotational symmetry. The base model has no antigen frame and uses no rotation augmentation, and the field is learned in the centred frame of the input.

11.2. Block loss and independent block clocks

Block-normalised loss. The two blocks differ in size, with $3n$ coordinate numbers and $8n$ latent numbers. A plain sum of squares would let the wider block dominate. The loss divides each block's squared error by the block size, so each block contributes its mean squared error:

$$\mathcal{L}_{\mathrm{block}} = \frac{\lambda_x}{d_x} \|r_x\|^2 + \frac{\lambda_z}{d_z} \|r_z\|^2.$$

Here r_x and r_z are the differences between predicted and target velocity in the coordinate and latent blocks, $d_x = 3n$ and $d_z = 8n$ are the block sizes, and $\lambda_x = \lambda_z = 1$. With $d_x = 60$, $\|r_x\|^2 = 30$, $d_z = 160$ and $\|r_z\|^2 = 80$ the loss is $30/60 + 80/160 = 0.5 + 0.5 = 1$. In the code, the coordinate term of a complex is the masked sum of squared residuals (in nm^2) divided by the number of

present residues and by the block width 3. The latent term is the same sum over the latent block, divided by the number of present residues and by the block width 8. Padding is excluded. The update loss is the sum over complexes divided by the global batch of 48.

Independent block times. Coordinates and latent can move at different speeds, so each block has its own time. Training draws the coordinate time t_x and the latent time t_z independently and uniformly on $[0, 1]$, once per complex. The published models draw them from mixtures of a uniform and a beta law that favour $t_x > t_z$: $0.02\,\mathrm{U}(0,1) + 0.98\,\mathrm{Beta}(1.9, 1)$ for t_x and $0.02\,\mathrm{U}(0,1) + 0.98\,\mathrm{Beta}(1, 1.5)$ for t_z [68, 71]. The base model does not use the mixture.

The interpolated state is $q_t = (x_{t_x}, z_{t_z})$ with $x_{t_x} = (1 - t_x)\, x_0 + t_x\, x_1$ and $z_{t_z} = (1 - t_z)\, z_0 + t_z\, z_1$. The network receives both times and returns a coordinate velocity v_θ^x and a latent velocity v_θ^z. Their targets are the end differences. The network therefore learns a field over the whole unit square of times.

Master clock. A sampler needs one curve through the square and one clock. Let $t_x(u)$ and $t_z(u)$ be non-decreasing, absolutely continuous schedules that equal 0 at $u = 0$ and 1 at $u = 1$. The heads give velocity per unit of their own time. The chain rule converts it to velocity per unit of the master clock u, and it is applied once:

$$\frac{\mathrm{d}q}{\mathrm{d}u} = f_\theta(q, u) = \begin{bmatrix} t_x'(u)\, v_\theta^x\big(q, t_x(u), t_z(u)\big) \\ t_z'(u)\, v_\theta^z\big(q, t_x(u), t_z(u)\big) \end{bmatrix}.$$

Training never applies these factors. For $t_x = 2u - u^2$ and $t_z = u^2$ at $u = 0.25$ the times are 0.4375 and 0.0625, and the slopes are 1.5 and 0.5. Head outputs of 4 and 2 then give master-clock velocities of 6 and 1.

The sampler of the full system compares three curves. The reference curve follows the exponential and quadratic schedules of the published sampler [68]: $t_x(u) = (1 - 10^{-2u})/(1 - 10^{-2})$ with $t_z(u) = u^2$. The equal clock sets $t_x = t_z = u$. The geometry-leading candidate is the pair in the worked example. Training supplies both endpoints of the straight path illustrated in Figure 4a, while sampling advances from noise under the frozen velocity heads.

11.3. Autoencoder and decoder factorisation

The flow transports only the C-alpha coordinates and the latent, so a decoder must supply the rest. Let s be the residue sequence, $M_{\mathrm{atom}}(s)$ the atom37 mask that marks which of the 37 atom slots exist for those residues, and y the coordinates of the non-C-alpha atoms. Let s_d be the designable residues and s^c the clamped ones copied from the context. The joint density follows the order in which the pieces are needed:

$$p(q, s, M_{\mathrm{atom}}, y) = p_\theta(q)\; p_\phi(s_d \mid x, z)\; \delta_{s^c}\; \delta_{M_{\mathrm{atom}}(s)}\; p_\phi(y \mid s, M_{\mathrm{atom}}, x, z).$$

Read from left to right, the flow gives the state, and the decoder gives a probability for each designable residue. Clamped residues are copied. The atom mask is a fixed function of the sequence, so it is a point mass. The other atoms follow, given the sequence, the mask and the state. A serine and a tryptophan have different atoms, which fixes the order. The sequence factor uses counting measure and the coordinate factor uses Lebesgue measure on the present atoms. The base model has no context, so every residue is designable and the clamped factor is empty.

The base model implements the encoder and the decoder as two deterministic networks that train together (Figure 5a). The encoder reads the 169 numbers of each residue packet (111 centred coordinates in nanometres, 37 mask entries and 21 category entries; Section 8) and returns the latent. The decoder reads the centred C-alpha position and the latent, 11 numbers in all. It returns 37 by 3 coordinates in nanometres, with the C-alpha slot included, and 21 residue logits. Neither network sees the chain index. The loss of one complex with n residues and N_a present atoms is

$$\mathcal{L}_{\mathrm{AE}} = \frac{1}{3N_a} \sum_{(i,a)} \left\| \hat{y}_{ia} - y_{ia} \right\|^2 + \frac{1}{n} \sum_{i} \mathrm{CE}\big(s_i, \ell_i\big) + 0.01 \, \frac{1}{8n} \sum_{i} \|z_i\|^2 .$$

The first term is the mean squared error over present atoms, which corresponds to a Gaussian density of fixed variance. It is in nm^2 per coordinate, and the reconstruction report converts its mean to a per-coordinate root-mean-square error in angstroms (Technical Report III (Section "Reporting for the base model")). The second is the cross-entropy of the residue category, with logits ℓ_i. The third replaces the Kullback-Leibler term of a variational autoencoder. It keeps the latent from growing without bound, and its weight is 0.01. La-Proteina uses a Gaussian posterior with a Kullback-Leibler weight of 10^{-4} [68], which the base model does not.

Training uses the atoms that are present in the data record. The mask of a decoded sequence is needed only for generation. The encoder is frozen before the denoiser trains, and the denoiser stage reads the latent that the frozen encoder gives for each structure (Section 12).

11.4. Architecture and parameter counts

Block. The encoder, the decoder and the denoiser are stacks of one block (Figure 5a). The encoder and decoder have 12 blocks and the denoiser has 14. Every block has width 768 and 12 attention heads of width 64, and has a modulation layer, a self-attention layer and a gated feed-forward layer. A condition vector of width 256 passes through one linear layer to six vectors of width 768: a shift, a scale and a gate for the attention half, and the same three for the feed-forward half. The layout is the adaptive layer norm of diffusion transformers [131], with a sigmoid gate. In the denoiser the condition is the time embedding. The encoder and decoder have neither time nor antigen, so their condition is a learned constant vector of width 256, initialised at zero.

Feed-forward layer and masking. The feed-forward layer widens 768 to 6144, splits into a value and a gate of 3072 and returns to 768. The value half is multiplied by the sigmoid-linear unit (SiLU) of the gate half, in the pattern of gated linear units [132], and the hidden width is not reduced to equalise parameters. No network has dropout or triangle layers. Keys of padded residues are masked, and padded positions are zeroed after each residual update.

Inputs and outputs. Each network projects its per-residue input to width 768 and adds a sinusoidal encoding of the residue index (256 to 768). The encoder reads 169 numbers and returns the latent through a map from 768 to 8. The decoder reads 11 numbers, the centred C-alpha position and the latent, and returns 37 by 3 coordinates through a map from 768 to 111 and 21 logits. The denoiser reads the 11 numbers of the noisy state and returns the velocities through two linear heads, 768 to 3 and 768 to 8. Each time is multiplied by 1,000 and passes through a sinusoidal encoding of width 256. The two encodings are joined to width 512 and mapped by linear layers 512 to 256 to 256 to give the condition.

Pair representation. Only the denoiser has one. Its attention logits receive a pair bias, one

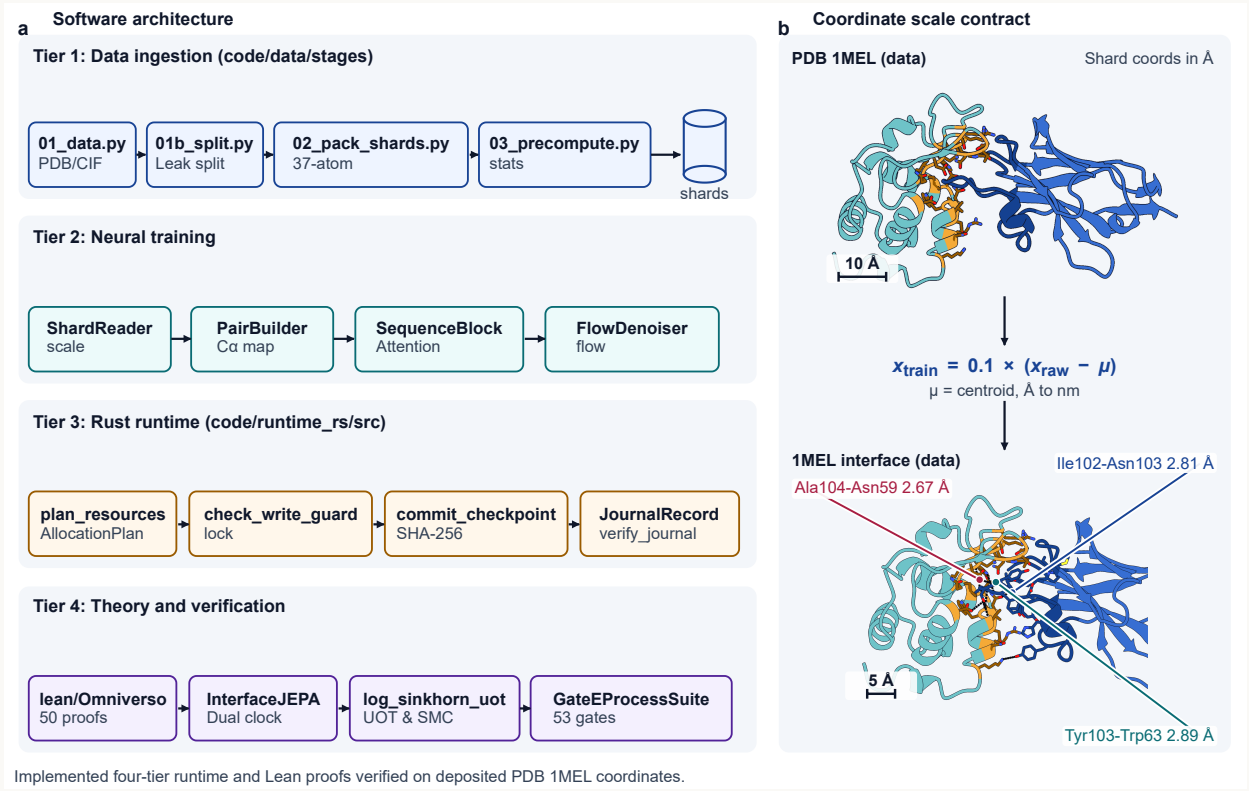

Figure 5: Four implemented software and formal verification tiers surround the Angstrom-to-nanometre centring contract on PDB 1MEL. a, Live software architecture tying Python shard ingestion (`code/data/stages/01_data.py` to `03_precompute.py`), the PyTorch training trunk (`ShardReader`, `PairBuilder`, `SequenceBlock` ×14, `FlowDenoiser`, and autoencoder modules), the Rust hardware orchestrator (`checkpoint.rs`, `distributor.rs`, `journal.rs`, and `sha256.rs`), and the formal verification and conditioning layer (50 theorems across six files with zero `sorry` placeholders in `lean/Omniverso`, alongside `InterfaceJEPA`, `log_sinkhorn_uot`, and `GateEProcessSuite`). **b,** Coordinate centring transform on deposited PDB 1MEL chain A and chain L removing atom centroids before multiplying by 0.1 to convert ångströms to nanometres (4.5 Å cutoff, 55 interface pairs). Architecture tiers trace live repository modules; PDB 1MEL coordinates illustrate data ingestion geometry, and no trained neural checkpoint was involved. PDB, Protein Data Bank.

linear map of the pair representation to 12 heads. In the encoder and decoder the bias is zero. The pair representation is the sum of three terms.

Pair features. Each C-alpha distance of the noisy coordinates is computed in nanometres and assigned to the nearest of 30 bins whose centres are defined at 1, 2, up to 30 Å, and mapped from 30 to 256 numbers. The code converts the 29 midpoints between the centres to nanometres once, so the bins keep their definition in angstroms, and a distance equal to a midpoint falls in the lower bin. The signed index difference, clipped to the range from -128 to 128, selects one of 258 embedding rows of width 128, and a linear map takes it to 256. Pairs in different chains take the separate category 257. A two-row embedding of width 256 marks whether two residues share a chain. The published configuration also bins the distances of a predicted clean structure [71]. The base model leaves them out because they need a second network pass.

Pair update. A residual MLP with a layer norm, 256 to 512 to 256, updates the pair representation before the attention of its layer. The default updates it in all 14 layers. Variants update it in every second layer or never (Table 6), and with no update the pair features still bias attention in every layer. The use of pair-biased attention follows La-Proteina [68].

Precision. The trunk runs in BF16, and coordinates, losses, the optimiser and the averaged weights stay in FP32 (Section 12).

Parameter counts. Table 6 gives parameter counts obtained by building each module on the meta device, which allocates shapes and no storage, and summing the tensors. A block without a pair path holds 10,634,496 parameters. The pair bias adds 3,072 and each pair update adds 263,424. The feed-forward layers hold about two thirds of each network, attention about a fifth and modulation about a tenth. Table 6 also names the layers that carry a pair update in each variant. The per-tensor table is in Supplementary Section S5.

Table 6: Number of trainable parameters of the base-model networks. The last row is the pair that a sampler loads, the decoder plus the denoiser. Updating the pair representation in every second layer removes 1,843,968 parameters from the default denoiser, and omitting the update removes 3,687,936.

Module	Pair updates	Parameters
Encoder	none	127,949,832
Decoder	none	127,923,844
Autoencoder (encoder plus decoder)	none	255,873,676
Denoiser, every layer (default)	all 14 layers	153,102,091
Denoiser, every second layer	layers 2, 4, 6, 8, 10, 12, 14	151,258,123
Denoiser, no pair update	none	149,414,155
Decoder plus denoiser (default)	all 14 layers	281,025,935

The Proteina-Complexa denoiser has 159 M parameters and its autoencoder 256 M [71]. The denoiser here has 5,897,909 parameters, or 3.7 percent, fewer than 159 M, and the autoencoder is within 0.1 percent of 256 M. The remaining gap comes from details that the published tables do not give, such as the width of the pair MLP and the module list. La-Proteina reports about 160 M for its denoiser in its text [68]. A pair update adds few parameters but acts on a tensor of size L by L by 256, so its cost in time and memory cannot be read from the count (Section 12).

Flow density. For a continuously differentiable velocity field whose flow is a diffeomorphism, the log density of a state follows from the change of variables. Along a trajectory the log density

changes at a rate equal to minus the divergence of the field, and integrating from the base state to the endpoint gives

$$\log p_1(q_1) = \log p_0(q_0) - \int_0^1 \operatorname{div} f_\theta(q_u, u)\,\mathrm{d}u.$$

The divergence is the trace of the Jacobian of the field. With the master clock it equals $t'_x(u)\,\mathrm{tr}(\partial v^x_\theta/\partial x) + t'_z(u)\,\mathrm{tr}(\partial v^z_\theta/\partial z)$, because the coupling terms between coordinates and latent lie off the diagonal. A Hutchinson estimator replaces the trace by the mean of $\xi^\top J \xi$ over random probe vectors ξ whose covariance is the identity [133]. A density in angstroms differs from one in standardised units by the determinant of the change of units, so a report states which one it gives. The base-model denoiser assigns distances to hard bins, so its field is not continuously differentiable in the coordinates, and the base model computes no likelihood. The derivations of the density formula, the clock rule and the change of units are in Supplementary Section S1.

12. Training of the base model

The base model is specified for training from random initialisation in two stages, one after the other. No released weight is loaded. In stage `ae`, the encoder and the decoder train together to turn each residue of a complex into eight latent numbers and to decode them into atoms. The encoder is then frozen. In stage `denoiser`, the velocity network learns the field of the flow over C-alpha coordinates and latents (Section 11) from the latents that the frozen encoder returns in evaluation mode. The decoder takes no part in the second stage, and no gradient reaches the encoder. Both stages read the same shards (Section 10.4). The model is unconditional: it has no antigen, hotspot or format input. Table 8 gives the length buckets with their microbatch, accumulation steps and ranks, and Table 9 gives one distributed update with the skip rule and the moving average.

Units and losses. Unit conversion follows Section 8. The autoencoder uses the objective of Section 11.3 and receives the true centred C-alpha positions. The denoiser uses the noise and interpolation of Section 11.1 and the block-normalised loss of Section 11.2, with independent uniform coordinate and latent times. Padding is excluded from both objectives. Reconstruction reporting follows Technical Report III (Section “Reporting for the base model”).

Optimiser and schedule. Both stages use AdamW [134] with the settings of Table 7. With zero weight decay AdamW reduces to Adam. The learning rate is 10^{-4} after a linear warm-up: the update that follows k successful updates uses $10^{-4}\min\{1, (k+1)/500\}$, so the first update uses 2×10^{-7} and the 500th reaches 10^{-4}. No decay follows. Skipped updates do not advance the schedule. Gradients are clipped to a global 2-norm of 1.

Moving average. A moving average of the parameters is kept for later use. Its decay is 0.999, so it averages over about 1,000 updates. The copy is taken at the first successful update whose step count exceeds 1,000, so that the average starts from weights that have left the warm-up. After the copy, each successful update sets the average to 0.999 times itself plus 0.001 times the current parameters. Only parameters are averaged, and the average is stored in every checkpoint.

Length buckets and batch composition. Complexes fall into three disjoint length buckets, up to 300, 301 to 400 and 401 to 500 residues. Each bucket is padded to its upper limit and masked (Section 10.4). Every update uses 48 complexes from one bucket. Two ranks share the work, each

Table 7: Training settings shared by both stages. The learning rate applies to each optimiser update. Loss weights and block times are dimensionless.

Setting	Value
Optimiser	AdamW, betas 0.9 and 0.999, epsilon 10^{-8}, weight decay 0
Learning rate	10^{-4} after 500 warm-up updates, then constant
Gradient clipping	global 2-norm 1
Moving average	decay 0.999, first copy after step 1,000
Global batch	48 complexes per update, on two ranks
Loss weights	coordinate 1, latent 1, latent penalty 0.01
Block times	t_x, t_z independent, uniform on $[0, 1]$
Checkpoint interval	1,000 updates

with a microbatch m and a accumulation steps, so that $2ma = 48$ (Table 8). The code derives a as $48/(\text{ranks} \times m)$, so a single process doubles it and still uses 48 complexes. A bucket is drawn uniformly at random for each update. All ranks obtain the same bucket and the same 48 indices, drawn with replacement inside the bucket, and each rank takes its own slices.

Table 8: Batch composition by length bucket. Lengths are in residues; the microbatch is counted in complexes per rank and per accumulation step.

Bucket (residues)	Padded length	Microbatch per rank	Accumulation steps	Ranks	Complexes per update
up to 300	300	8	3	2	48
301 to 400	400	6	4	2	48
401 to 500	500	4	6	2	48

Distributed update. The loss of a microstep is the sum of the per-complex losses divided by 48. Summed over microsteps and ranks, it is the mean over the 48 complexes of the update, whatever the bucket. The data-parallel wrapper of PyTorch averages gradients over ranks by default [135], which would halve them under this scale. A communication hook therefore replaces the average by an FP32 sum and refuses a buffer of any other type. Only the last microstep exchanges gradients. The earlier ones run without synchronisation and accumulate locally.

Non-finite gradients. After the last microstep the gradient is clipped, and every rank checks that its norm and every entry are finite. The ranks share one flag, the minimum over ranks. If it is false, gradients are cleared, and the weights, optimiser state, moving average, schedule and data cursor stay as they were. The random state saved at the start of the update is restored, so the retry uses the same batch and the same noise. Three consecutive skips stop the run. Table 9 states one update in order.

Mixed precision. Trunk matrix products and attention run in BF16 under autocast. BF16 has the exponent range of FP32 [136]. The specified configuration uses BF16 trunk arithmetic without loss scaling and retains the stated optimiser settings. Attention softmax is accumulated in FP32 inside the kernel, and the attention bias is computed in FP32. BF16 keeps 8 significant bits, so a coordinate of 3 nm would be rounded on a grid of 0.016 nm, or 0.16 Å, and for this reason no coordinate enters BF16. Parameters, accumulated gradients, optimiser state, the moving average and both residual streams stay in FP32. So do layer normalisation, centring, unit conversion, noise,

Table 9: One distributed update of either stage, in order. A skipped update advances neither the step count, the schedule, the moving average nor the data cursor. The rules are those of the training code and of the cycle-1 parameter lock.

Step	Rule
1	Draw one length bucket and 48 complex indices, the same on both ranks, and save the random state
2	For each microstep, the loss is the sum of the per-complex losses divided by 48; only the last microstep exchanges gradients, and the earlier ones accumulate without synchronisation
3	Clip the gradient to a global 2-norm of 1
4	Each rank sets a flag from the finiteness of the norm and of every gradient entry; the ranks share the minimum of the flags
5	If the flag is false, clear the gradients, restore the saved random state, advance no counter and add one to the skip count; three consecutive skips stop the run
6	Otherwise take the optimiser step, add one to the step count, advance the warm-up schedule and reset the skip count
7	If the step count exceeds 1,000, copy the parameters at the first such update, and afterwards set the average to 0.999 times itself plus 0.001 times the parameters
8	Write a checkpoint every 1,000 successful updates and at the last update

interpolation, distances, the output heads, the losses, the latent penalty, clipping and the gradient exchange between ranks. TF32 is off.

Activation checkpointing. Activation checkpointing recomputes the intermediate tensors of each layer in the backward pass, at the cost of one extra forward pass [137, 138]. It is applied to every layer of the trunks.

Latents and seeds. The frozen encoder supplies the latent of each complex from the canonical input, with no rotation and no crop before encoding, because the encoder reads the whole complex. A cache of these latents may replace the live encoder pass, and it is valid only under that condition. Every random draw derives from a SHA-256 hash of the run identifier, the rank and the update index (Section 13).

Run record, checkpoints and resume. Each base-model training run records its data manifest and configuration and writes checkpoints and a training log. A checkpoint is written after a completed optimiser, moving-average and scheduler step, every 1,000 updates and at the last update. It holds the model, optimiser state, moving average, scheduler, step, data cursor and the random state of every rank, and the model file is renamed last, as the commit marker. Resume restores all of these and requires the same stage, configuration, device type and number of ranks.

Number of updates. The number of successful updates of stage s follows from the measured update time. For an allocated stage time T_s it is

$$N_s = \left\lfloor \frac{T_s}{\bar{t}_s} \right\rfloor, \qquad \bar{t}_s = \frac{1}{3} \sum_b t_{s,b} + \frac{c_s}{I}.$$

Here $t_{s,b}$ is the measured time of one complete update of stage s in length bucket b on the two cards, the sum runs over the three buckets, c_s is the time of one checkpoint write and $I = 1{,}000$ is the checkpoint interval. 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 to the training script through the option `--updates`. A skipped update does not advance the count. We estimate the matrix-product work of the checkpointed denoiser blocks as four times their forward work: a forward pass, a backward pass of twice that cost and one recomputed forward pass. Including the modulation layers, this is about 3.0, 5.3 and 8.1 TFLOP for one complex at lengths 300, 400 and 500, and it excludes the forward pass of the frozen encoder (derivation in Supplementary Section S5). The pair update takes 88 to 92 percent of the forward arithmetic and only 2.4 percent of the parameters, so the measured time sets the number of updates.

13. Computing, software and availability

Hardware and software. The training configuration uses one node with two NVIDIA A100 cards of 40 GB each, with one process per card and data-parallel gradient sums [135]. The code is written in Python 3.11 with PyTorch 2.11 built for CUDA 12.8 [139] and NumPy [140]. The number of updates follows from the measured update time (Section 12).

Code. The base-model code is supplied as an archive with the Supplementary Information; Supplementary Section S4 describes its contents and use. The archive holds the training code: the

autoencoder, the denoiser, the flow path and losses, the shard reader, the precision policy, the update loop, checkpoints and run records. It also holds three configurations, the scripts that launch a run and the unit tests that run on a CPU. Support code retrieves and cleans PDB entries and performs temporal, duplicate-identity and structural-fingerprint checks. The four-axis split protocol is specified in Section 10.3. Numerical checks cover the statements of Supplementary Section S1. The archive contains no trained weight, no structure data and no code of other systems. The code is licensed under AGPL-3.0 (Section 17).

Data. The base model reads public protein structures. The sources and their licence classes are given in Section 10.1. Each dataset stays with its provider, and this paper redistributes none. The unit tests use synthetic complexes that a seeded function generates, so that they need no download.

Weights. No trained weights accompany this paper. A weight set is released only if the licences of all its training sources allow it (Section 17).

Determinism. Every random draw starts from a SHA-256 hash of the run identifier together with a role label. The initial weights and the global generators use the label for the model, the noise generator of each rank uses the rank, and the batch of update u uses the update index. The run identifier and the update cursor alone therefore fix the length bucket and the draws of an update, and the data order is fixed. Both ranks draw the same indices and take separate slices. A resumed run restores the random state of each rank from the checkpoint, and a skipped update restores the state it started from, so the retry draws the same noise. The seed fixes sample selection and random-number streams. The code does not enable deterministic GPU kernels, and nondeterministic GPU operations can produce differences between repeated training runs. Integer outputs such as shard hashes, manifests, sample order and bucket choice are exact.

14. 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. Table 10 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.

Table 10: 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.

Dataset	Content and count	Licence class (terms reported)	Role
PDB [122]	Dated mmCIF snapshot, first biological assembly per entry	Open (CC0 1.0)	pretrain, reference
AlphaFold DB v6, 2025-09-15 [123]	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 [124]	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 [71]	3,556,223 clusters; 510,454 representatives after the interface filters	Non-commercial (CC BY-NC 4.0)	pretrain
PDB multimers, filtered [71]	45,856 structures	Open (CC0 1.0)	pretrain
SAbDab, 2026-09-30 [22, 23]	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 [103]	15,523 curated complexes; benchmark of 113 antibody-antigen and 101 VHH-antigen complexes	Reserved (training reserved by the authors)	benchmark
AbDb [141]; 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 [99]	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 [104]	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 [46]	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 [44]	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 [21]	2,428,152,691 sequences from 91 studies, 1.115 TB	Open (CC BY 4.0)	pretrain
OAS, paired [21]	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 [142]; PLAbDab, 2024-08-28 [143]; 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

Continued on next page

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, bench-mark
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 [144]	7,085 rows from 345 PDB entries; antibody-antigen 1,211 rows from 55 entries	Open (CC BY 4.0)	train, calibrate
AB-Bind [145]; AbBiBench [146]	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
Absolut! [125]	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 [15, 112]	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

Continued on next page

Dataset	Content and count	Licence class (terms reported)	Role
Harvey 2022 PSR [109]; 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 [18]	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

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 Table 11 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 for these antigens ”held out” means an unseen antibody interface.

Cut-off dates of published systems. A published result is comparable with another only if the training and test dates are known. Table 11 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.

Table 11: 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 $T^* = 2025\text{-}01\text{-}01$.

System	Benchmark targets	Cut-off or boundary
Chai-2 [84]	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 [78]	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 [80]	PD-L1, IL3, IL20, BHRF1	Not reported
BoltzGen [83]	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 [71]	19 protein targets; no antibody task	Last training deposit 2024-08-02, from the released training list
mBER [82]	145 screened of 436 targets; list unpublished	Not reported
SNAC-DB benchmark [103]	113 antibody-antigen and 101 VHH-antigen complexes	PDB deposits up to 2025-03-31
Hitawala-Gray benchmark [44]	118 antibody-antigen and 70 nanobody-antigen complexes	Boundary 2021-09-30

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. Its role is therefore to control overlap with the disclosed training sets of those four systems.

15. 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 Technical Report II (Section "Mathematical supplement"), and the parameter and cost arithmetic is in Section 16.

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 Technical Report II (Section "Mathematical supplement") on small synthetic problems. `code/tools` holds the test runner and the script that reproduces the cost arithmetic of Section 16.

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. The function `fp32(device)` returns a context with autocast off, and `trunk_autocast(device, enabled)` returns the BF16 context or an empty one. A call to `set_tf32(enabled)` sets the TF32 flags of matrix products and cuDNN and returns their values for the training log.

The layer-norm class `FloatLayerNorm` casts its input to FP32 and runs with autocast off. Attention goes through `attention(q, k, v, bias)`, which calls PyTorch's scaled dot-product attention in the query dtype, with the additive bias cast to that dtype. The helper `linear_float(layer, x)` applies a linear layer in FP32.

Layer normalisation, the output heads, the losses and the coordinate arithmetic always run in FP32, and the softmax inside attention accumulates in FP32. Stored parameters and accumulated

gradients stay in FP32. Trunk matrix products use BF16 autocast, so the forward and backward passes use mixed precision. TF32 is off unless a run switches it on.

File: `code/training/precision.py`.

```
"""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 [137] in training when the padded length reaches `checkpoint_min_length`. The class `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. Its counterpart `Decoder` maps the centred C-alpha position and the latent to all-atom coordinates in nanometres (37×3) and to 21 residue logits. The denoiser class `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 $m_j = 0.1\,(j + \frac{1}{2})$ 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 [71]. The coordinate loss uses nanometre coordinates; the latent is not standardised, and both block losses receive weight 1.

File: `code/training/models.py`.

```
"""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. Shards are written by `pack_shard`, which returns the bytes of an NPZ shard without pickle. The class `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` and offsets that do not start at zero and increase. Further checks reject a residue without a C-alpha atom at slot 1, a repeated (chain, residue index) pair and a manifest whose units are not ångströms. Batches are assembled by `collate(complexes, length, device)`, which 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.

File: `code/training/data.py`.

```
"""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 [135]. 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.

File: `code/training/train.py`

```
"""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. Table 12 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.

Table 12: Model, data and optimiser keys of the reference configuration `reference_2gpu.json` and their values. Learning rates are per update; lengths are in residues.

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

Continued on next page

Key	Meaning	Value
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

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.

File: `code/training/tests/test_units.py`.

```
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)
```

File: `code/training/tests/test_counts.py.`

```
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.

16. 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 15 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 $2w$. Table 13 gives each group and its formula.

Table 13: 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.

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**

Table 14 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.

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 [71]; 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

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

Table 14: 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.

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**

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 Table 15. 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
```

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

Table 15: 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).

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
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)	$2w\,(11+256+3+8)\,L+2\,(512\cdot256+256\cdot256)$	0.13	0.17	0.21
Modulation layers (GFLOP)	$K\cdot2\cdot256\cdot6w$	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)	$24\cdot32\,w^2L+24\cdot4\,w\,L^2$ 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

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. Table 16 gives the figures at the reference size. Activations are not included.

Table 16: Memory of the optimiser state at the reference size, in bytes.

Quantity	Formula	Autoencoder stage	Denoiser stage
Trainable parameters P_{train}	Table 14	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

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.

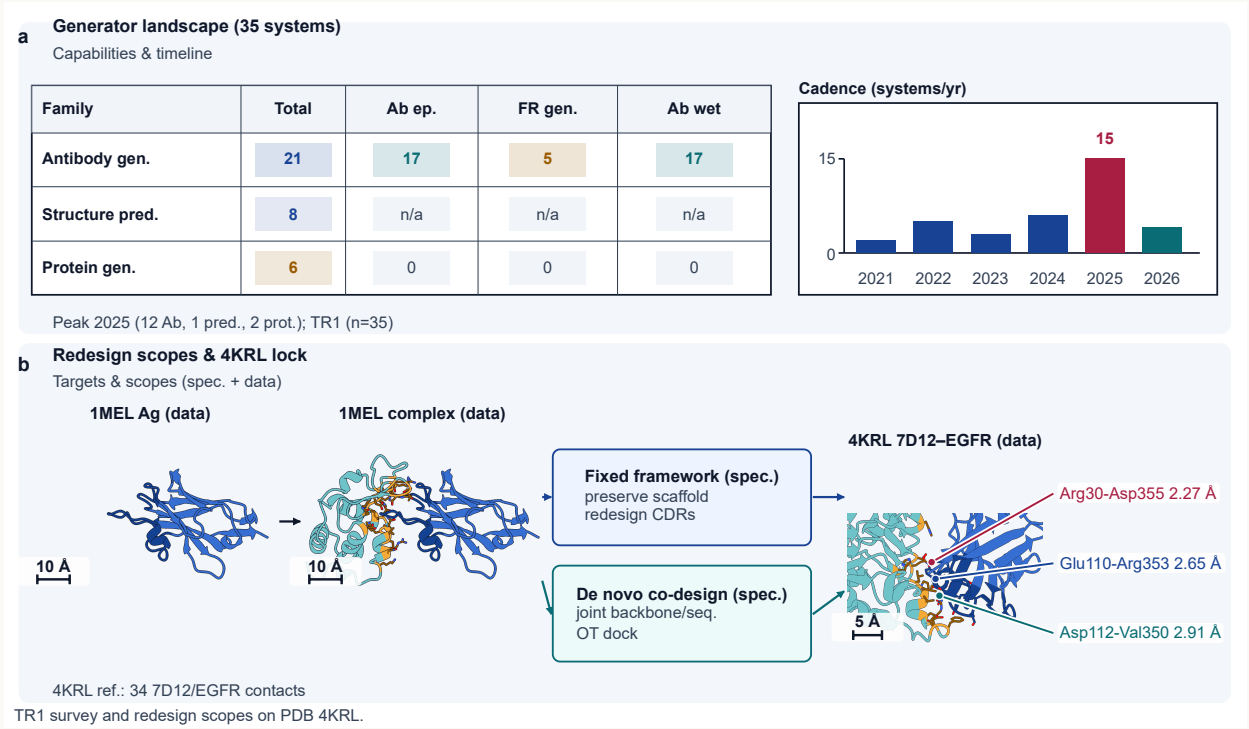

Figure 6: Thirty-five public structural design systems peak in 2025 alongside Omniverso fixed-framework and de novo redesign scopes tied to PDB 4KRL. a, Summary table of 35 surveyed tools across antibody generation (21 total, 17 epitope-aware, 5 framework-generating, 17 with reported wet-lab tests), structure prediction (8 total), and general protein generation (6 total), paired with yearly publication counts from 2021 to 2026 peaking at 15 systems in 2025. **b,** Target complex workflow moving from PDB 1MEL antigen and bound complex through fixed-framework CDR redesign and de novo co-design with optimal-transport docking toward atomic interface checks on PDB 4KRL 7D12–EGFR (Arg30–Asp355 at 2.27 Å, Glu110–Arg353 at 2.65 Å, and Asp112–Val350 at 2.91 Å; 34 contacts). System counts summarise unmatched external reports; redesign branches show full-system scopes without competitive ranking. CDR, complementarity-determining region; FR, framework region; OT, optimal transport; PDB, Protein Data Bank.

17. Scope, Ethics, Licence, and Declarations

17.1. Scope, Dual Use, and Licensing

Scope. The work is computational and contains no wet-lab step. Predictions of the full system would remain predictions until an experiment tests them. A comparison with another group uses its published numbers, names the differences in protocol and states that the panels are not matched.

Dual use and biosafety. A tool that designs binders can be misused, as can any generative model for biochemistry. A drug-discovery model proposed toxic molecules once its objective was inverted [147]. AI-redesigned variants of proteins of concern evaded the screens of nucleic acid synthesis providers until those screens were patched [148]. Synthesis is therefore the control point, and one proposal keeps synthesis records in repositories that are queried only in emergencies [149]. The study takes three positions. The base model contains no component that conditions on a target or emits an antibody sequence. The manuscript prints no designed sequence, and the code package releases none: its only data are synthetic complexes. Any later release of an antigen-conditioned generator would first need a review of misuse and a screen of requested targets against curated lists of controlled pathogen accessions. These positions reduce risk and do not amount to a safety claim.

Licences and data. The code package is licensed under the GNU Affero General Public License, version 3 (AGPL-3.0). Its section 13 requires that a modified version offered to users over a network make its source available. The licence covers code and grants nothing on weights or data. Third-party libraries keep their own licences, and the main ones are named in Section 13. Third-party papers appear as citations only. The licence of trained weights follows the licences of the training data (Section 10.1). The base model reads public protein structures only and no personal, clinical or patient data, and each downloaded file is stored with its SHA-256 digest (Section 10.4).

17.2. Declarations

Author contributions. D.K. is the sole author and carried out the conception, design, code and writing of the work.

Data and code availability. The training code, its tests, the configuration files and the container definition are supplied as Supplementary Information and described in Supplementary Section S4. The code is licensed under the GNU Affero General Public License, version 3. Each dataset stays with its provider, and this manuscript redistributes none. The sources and their licences are listed in Section 10 and Supplementary Section S2.

Artwork. The protein pictures in Figures 1 to 5 and Figure 6 were rendered with UCSF ChimeraX from deposited coordinates of the Protein Data Bank (PDB entries 1MEL, 1VFB, 7Z0X, 1IGT and 4KRL).

References

[1] Richard A Norman, Francesco Ambrosetti, Alexandre M J J Bonvin, Lucy J Colwell, Sebastian Kelm, Sandeep Kumar, and Konrad Krawczyk. Computational approaches to therapeutic antibody design:

established methods and emerging trends. *Briefings in Bioinformatics*, 21:1549–1567, 2020. doi: 10.1093/bib/bbz095.

[2] Marie-Paule Lefranc and Gérard Lefranc. IMGT® and 30 Years of Immunoinformatics Insight in Antibody V and C Domain Structure and Function. *Antibodies*, 8:29, 2019. doi: 10.3390/antib8020029.

[3] Inbal Sela-Culang, Vered Kunik, and Yanay Ofran. The Structural Basis of Antibody-Antigen Recognition. *Frontiers in Immunology*, 4:302, 2013. doi: 10.3389/fimmu.2013.00302.

[4] C. Hamers-Casterman, T. Atarhouch, S. Muyldermans, G. Robinson, C. Hammers, E. Bajyana Songa, N. Bendahman, and R. Hammers. Naturally occurring antibodies devoid of light chains. *Nature*, 363: 446–448, 1993. doi: 10.1038/363446a0.

[5] Serge Muyldermans. Nanobodies: Natural Single-Domain Antibodies. *Annual Review of Biochemistry*, 82:775–797, 2013. doi: 10.1146/annurev-biochem-063011-092449.

[6] Ivana Jovčevska and Serge Muyldermans. The Therapeutic Potential of Nanobodies. *BioDrugs*, 34 (1):11–26, 2020. doi: 10.1007/s40259-019-00392-z. PMID:31686399.

[7] Laura S. Mitchell and Lucy J. Colwell. Comparative analysis of nanobody sequence and structure data. *Proteins: Structure, Function, and Bioinformatics*, 86:697–706, 2018. doi: 10.1002/prot.25497.

[8] Gemma L. Gordon, Henriette L. Capel, Bora Guloglu, Eve Richardson, Ryan L. Stafford, and Charlotte M. Deane. A comparison of the binding sites of antibodies and single-domain antibodies. *Frontiers in Immunology*, 14:1231623, 2023. doi: 10.3389/fimmu.2023.1231623. PMID:37533864.

[9] Marie-Paule Lefranc, Christelle Pommié, Manuel Ruiz, Véronique Giudicelli, Elodie Foulquier, Lisa Truong, Valérie Thouvenin-Contet, and Gérard Lefranc. IMGT unique numbering for immunoglobulin and T cell receptor variable domains and Ig superfamily V-like domains. *Developmental & Comparative Immunology*, 27(1):55–77, 2003. doi: 10.1016/s0145-305x(02)00039-3. PMID:12477501.

[10] James Dunbar and Charlotte M. Deane. ANARCI: antigen receptor numbering and receptor classification. *Bioinformatics*, 32(2):298–300, 2016. doi: 10.1093/bioinformatics/btv552. PMID:26424857.

[11] Zuhaida Asra Ahmad, Swee Keong Yeap, Abdul Manaf Ali, Wan Yong Ho, Noorjahan Banu Mohamed Alitheen, and Muhajir Hamid. scFv Antibody: Principles and Clinical Application. *Clinical and Developmental Immunology*, 2012:980250, 2012. doi: 10.1155/2012/980250.

[12] Yasmina Noubia Abdiche, Andy Yik Yeung, Irene Ni, Donna Stone, Adam Miles, Winse Morishige, Andrea Rossi, and Pavel Strop. Antibodies Targeting Closely Adjacent or Minimally Overlapping Epitopes Can Displace One Another. *PLOS ONE*, 12(1):e0169535, 2017. doi: 10.1371/journal.pone.0169535. PMID:28060885.

[13] Yu-Xi Tsai, Ning-En Chang, Klaus Reuter, Hao-Ting Chang, Tzu-Jing Yang, Sören von Bülow, Vidhi Sehrawat, Noémie Zerrouki, Matthieu Tuffery, Michael Gecht, Isabell Louise Grothaus, Lucio Colombi Ciacchi, Yong-Sheng Wang, Min-Feng Hsu, Kay-Hooi Khoo, Gerhard Hummer, Shang-Te Danny Hsu, Cyril Hanus, and Mateusz Sikora. Rapid simulation of glycoprotein structures by grafting and steric exclusion of glycan conformer libraries. *Cell*, 187(5):1296–1311.e26, 2024. doi: 10.1016/j.cell.2024.01.034. PMID:38428397.

[14] Yingda Xu, William Roach, Tingwan Sun, Tushar Jain, Bianka Prinz, Ta-Yi Yu, Joshua Torrey, Jerry Thomas, Piotr Bobrowicz, Maximiliano Vásquez, K. Dane Wittrup, and Eric Krauland. Addressing polyspecificity of antibodies selected from an in vitro yeast presentation system: a FACS-based, high-throughput selection and analytical tool. *Protein Engineering, Design and Selection*, 26:663–670, 2013. doi: 10.1093/protein/gzt047.

[15] Matthew I. J. Raybould, Claire Marks, Konrad Krawczyk, Bruck Taddese, Jaroslaw Nowak, Alan P. Lewis, Alexander Bujotzek, Jiye Shi, and Charlotte M. Deane. Five computational developability guidelines for therapeutic antibody profiling. *Proceedings of the National Academy of Sciences*, 116 (10):4025–4030, 2019. doi: 10.1073/pnas.1810576116. PMID:30765520.

[16] Jasmin F. Sydow, Florian Lipsmeier, Vincent Larraillet, Maximiliane Hilger, Bjoern Mautz, Michael Mølhøj, Jan Kuentzer, Stefan Klostermann, Juergen Schoch, Hans R. Voelger, Joerg T. Regula, Patrick

Cramer, Apollon Papadimitriou, and Hubert Kettenberger. Structure-Based Prediction of Asparagine and Aspartate Degradation Sites in Antibody Variable Regions. *PLoS ONE*, 9(6):e100736, 2014. doi: 10.1371/journal.pone.0100736. PMID:24959685.

[17] Fleur S. van de Bovenkamp, Ninotska I. L. Derksen, Pleuni Ooijevaar-de Heer, Karin A. van Schie, Simone Kruithof, Magdalena A. Berkowska, C. Ellen van der Schoot, Hanna IJspeert, Mirjam van der Burg, Ann Gils, Lise Hafkenscheid, René E. M. Toes, Yoann Rombouts, Rosina Plomp, Manfred Wuhrer, S. Marieke van Ham, Gestur Vidarsson, and Theo Rispens. Adaptive antibody diversification through N -linked glycosylation of the immunoglobulin variable region. *Proceedings of the National Academy of Sciences*, 115(8):1901–1906, 2018. doi: 10.1073/pnas.1711720115. PMID:29432186.

[18] David Prihoda, Jad Maamary, Andrew Waight, Veronica Juan, Laurence Fayadat-Dilman, Daniel Svozil, and Danny A. Bitton. BioPhi: A platform for antibody design, humanization, and humanness evaluation based on natural antibody repertoires and deep learning. *mAbs*, 14(1):2020203, 2022. doi: 10.1080/19420862.2021.2020203. PMID:35133949.

[19] Birkir Reynisson, Bruno Alvarez, Sinu Paul, Bjoern Peters, and Morten Nielsen. NetMHCpan-4.1 and NetMHCIIpan-4.0: improved predictions of MHC antigen presentation by concurrent motif deconvolution and integration of MS MHC eluted ligand data. *Nucleic Acids Research*, 48(W1):W449–W454, 2020. doi: 10.1093/nar/gkaa379. PMID:32406916.

[20] Andrew R M Bradbury, Sachdev Sidhu, Stefan Dübel, and John McCafferty. Beyond natural antibodies: the power of in vitro display technologies. *Nature Biotechnology*, 29:245–254, 2011. doi: 10.1038/nbt.1791.

[21] Tobias H. Olsen, Fergus Boyles, and Charlotte M. Deane. Observed Antibody Space: A diverse database of cleaned, annotated, and translated unpaired and paired antibody sequences. *Protein Science*, 31(1):141–146, 2022. doi: 10.1002/pro.4205. PMID:34655133.

[22] James Dunbar, Konrad Krawczyk, Jinwoo Leem, Terry Baker, Angelika Fuchs, Guy Georges, Jiye Shi, and Charlotte M. Deane. SAbDab: the structural antibody database. *Nucleic Acids Research*, 42(D1):D1140–D1146, 2014. doi: 10.1093/nar/gkt1043. PMID:24214988.

[23] Henriette L. Capel, Odysseas Vavourakis, Benjamin H. Williams, Christopher R. Taylor, and Charlotte M. Deane. SAbDab2: The structural antibody database in the age of machine learning. *bioRxiv*, 2026. doi: 10.64898/2026.06.16.732554.

[24] Ryan Brenke, David R. Hall, Gwo-Yu Chuang, Stephen R. Comeau, Tanggis Bohnuud, Dmitri Beglov, Ora Schueler-Furman, Sandor Vajda, and Dima Kozakov. Application of asymmetric statistical potentials to antibody–protein docking. *Bioinformatics*, 28:2608–2614, 2012. doi: 10.1093/bioinformatics/bts493.

[25] Francesco Ambrosetti, Brian Jiménez-García, Jorge Roel-Touris, and Alexandre M.J.J. Bonvin. Modeling Antibody-Antigen Complexes by Information-Driven Docking. *Structure*, 28:119–129.e2, 2020. doi: 10.1016/j.str.2019.10.011.

[26] Aroop Sircar and Jeffrey J. Gray. SnugDock: Paratope Structural Optimization during Antibody-Antigen Docking Compensates for Errors in Antibody Homology Models. *PLoS Computational Biology*, 6:e1000644, 2010. doi: 10.1371/journal.pcbi.1000644.

[27] Shaun M Lippow, K Dane Wittrup, and Bruce Tidor. Computational design of antibody-affinity improvement beyond in vivo maturation. *Nature Biotechnology*, 25:1171–1176, 2007. doi: 10.1038/nbt1336.

[28] Jared Adolf-Bryfogle, Oleks Kalyuzhniy, Michael Kubitz, Brian D. Weitzner, Xiaozhen Hu, Yumiko Adachi, William R. Schief, and Roland L. Dunbrack. RosettaAntibodyDesign (RAbD): A general framework for computational antibody design. *PLOS Computational Biology*, 14:e1006112, 2018. doi: 10.1371/journal.pcbi.1006112.

[29] Jeffrey A. Ruffolo, Jeffrey J. Gray, and Jeremias Sulam. Deciphering antibody affinity maturation with language models and weakly supervised learning. *arXiv*, 2021. URL: https://arxiv.org/abs/2112.07782.

[30] Richard W. Shuai, Jeffrey A. Ruffolo, and Jeffrey J. Gray. IgLM: Infilling language modeling for antibody sequence design. *Cell Systems*, 14:979–989.e4, 2023. doi: 10.1016/j.cels.2023.10.001.

[31] Tobias H Olsen, Iain H Moal, and Charlotte M Deane. AbLang: an antibody language model for completing antibody sequences. *Bioinformatics Advances*, 2(1):vbac046, 2022. doi: 10.1093/bioadv/vbac046. PMID:36699403.

[32] Brian L. Hie, Varun R. Shanker, Duo Xu, Theodora U. J. Bruun, Payton A. Weidenbacher, Shaogeng Tang, Wesley Wu, John E. Pak, and Peter S. Kim. Efficient evolution of human antibodies from general protein language models. *Nature Biotechnology*, 42:275–283, 2024. doi: 10.1038/s41587-023-01763-2.

[33] Tobias H Olsen, Iain H Moal, and Charlotte M Deane. Addressing the antibody germline bias and its effect on language models for improved antibody design. *Bioinformatics*, 40:btae618, 2024. doi: 10.1093/bioinformatics/btae618.

[34] Jeffrey A. Ruffolo, Jeremias Sulam, and Jeffrey J. Gray. Antibody structure prediction using interpretable deep learning. *Patterns*, 3:100406, 2022. doi: 10.1016/j.patter.2021.100406.

[35] Jeffrey A. Ruffolo, Lee-Shin Chu, Sai Pooja Mahajan, and Jeffrey J. Gray. Fast, accurate antibody structure prediction from deep learning on massive set of natural antibodies. *Nature Communications*, 14:2389, 2023. doi: 10.1038/s41467-023-38063-x.

[36] Brennan Abanades, Wing Ki Wong, Fergus Boyles, Guy Georges, Alexander Bujotzek, and Charlotte M. Deane. ImmuneBuilder: Deep-Learning models for predicting the structures of immune proteins. *Communications Biology*, 6:575, 2023. doi: 10.1038/s42003-023-04927-7.

[37] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596:583–589, 2021. doi: 10.1038/s41586-021-03819-2.

[38] Richard Evans, Michael O'Neill, Alexander Pritzel, Natasha Antropova, Andrew Senior, Tim Green, Augustin Žídek, Russ Bates, Sam Blackwell, Jason Yim, Olaf Ronneberger, Sebastian Bodenstein, Michal Zielinski, Alex Bridgland, Anna Potapenko, Andrew Cowie, Kathryn Tunyasuvunakool, Rishub Jain, Ellen Clancy, Pushmeet Kohli, John Jumper, and Demis Hassabis. Protein complex prediction with AlphaFold-Multimer. *bioRxiv*, 2021. doi: 10.1101/2021.10.04.463034.

[39] Rui Yin, Brandon Y. Feng, Amitabh Varshney, and Brian G. Pierce. Benchmarking AlphaFold for protein complex modeling reveals accuracy determinants. *Protein Science*, 31:e4379, 2022. doi: 10.1002/pro.4379.

[40] Josh Abramson, Jonas Adler, Jack Dunger, Richard Evans, Tim Green, Alexander Pritzel, Olaf Ronneberger, Lindsay Willmore, Andrew J. Ballard, Joshua Bambrick, Sebastian W. Bodenstein, David A. Evans, Chia-Chun Hung, Michael O'Neill, David Reiman, Kathryn Tunyasuvunakool, Zachary Wu, Akvilė Žemgulytė, Eirini Arvaniti, Charles Beattie, Ottavia Bertolli, Alex Bridgland, Alexey Cherepanov, Miles Congreve, Alexander I. Cowen-Rivers, Andrew Cowie, Michael Figurnov, Fabian B. Fuchs, Hannah Gladman, Rishub Jain, Yousuf A. Khan, Caroline M. R. Low, Kuba Perlin, Anna Potapenko, Pascal Savy, Sukhdeep Singh, Adrian Stecula, Ashok Thillaisundaram, Catherine Tong, Sergei Yakneen, Ellen D. Zhong, Michal Zielinski, Augustin Žídek, Victor Bapst, Pushmeet Kohli, Max Jaderberg, Demis Hassabis, and John M. Jumper. Accurate structure prediction of biomolecular interactions with AlphaFold 3. *Nature*, 630(8016):493–500, 2024. doi: 10.1038/s41586-024-07487-w. PMID:38718835.

[41] Jeremy Wohlwend, Gabriele Corso, Saro Passaro, Noah Getz, Mateo Reveiz, Ken Leidal, Wojtek Swiderski, Liam Atkinson, Tally Portnoi, Itamar Chinn, Jacob Silterra, Tommi Jaakkola, and Regina Barzilay. Boltz-1 Democratizing Biomolecular Interaction Modeling. *bioRxiv*, 2024. doi: 10.1101/

2024.11.19.624167.

[42] Chai Discovery, Jacques Boitreaud, Jack Dent, Matthew McPartlon, Joshua Meier, Vinicius Reis, Alex Rogozhnikov, and Kevin Wu. Chai-1: Decoding the molecular interactions of life. *bioRxiv*, 2024. doi: 10.1101/2024.10.10.615955.

[43] Saro Passaro, Gabriele Corso, Jeremy Wohlwend, Mateo Reveiz, Stephan Thaler, Vignesh Ram Somnath, Noah Getz, Tally Portnoi, Julien Roy, Hannes Stark, David Kwabi-Addo, Dominique Beaini, Tommi Jaakkola, and Regina Barzilay. Boltz-2: Towards Accurate and Efficient Binding Affinity Prediction. *bioRxiv*, 2025. doi: 10.1101/2025.06.14.659707.

[44] Fatima N. Hitawala and Jeffrey J. Gray. What does AlphaFold3 learn about antibody and nanobody docking, and what remains unsolved? *mAbs*, 17:2545601, 2025. doi: 10.1080/19420862.2025.2545601.

[45] Samuel Fromm, Marko Ludaic, and Arne Elofsson. Evaluating deep learning based structure prediction methods on antibody–antigen complexes. *Bioinformatics*, 42(4):btag136, 2026. doi: 10.1093/bioinformatics/btag136. PMID:41863324.

[46] Eva Smorodina, Montader Ali, Klara Kropivšek Brumat, Leonardo Salicari, Samo Miklavc, Aibek Kappassov, Chengcheng Fu, Pietro Sormanni, Ario de Marco, and Victor Greiff. Structural Plausibility Without Binding Specificity: Limits of AI-Based Antibody-Antigen Structure Prediction Confidence Scores. *bioRxiv*, 2026. doi: 10.64898/2026.03.02.709004.

[47] J. Dauparas, I. Anishchenko, N. Bennett, H. Bai, R. J. Ragotte, L. F. Milles, B. I. M. Wicky, A. Courbet, R. J. de Haas, N. Bethel, P. J. Y. Leung, T. F. Huddy, S. Pellock, D. Tischer, F. Chan, B. Koepnick, H. Nguyen, A. Kang, B. Sankaran, A. K. Bera, N. P. King, and D. Baker. Robust deep learning–based protein sequence design using ProteinMPNN. *Science*, 378:49–56, 2022. doi: 10.1126/science.add2187.

[48] Chloe Hsu, Robert Verkuil, Jason Liu, Zeming Lin, Brian Hie, Tom Sercu, Adam Lerer, and Alexander Rives. Learning inverse folding from millions of predicted structures. *bioRxiv*, 2022. doi: 10.1101/2022.04.10.487779.

[49] Magnus Haraldson Høie, Alissa M Hummer, Tobias H Olsen, Broncio Aguilar-Sanjuan, Morten Nielsen, and Charlotte M Deane. AntiFold: improved structure-based antibody design using inverse folding. *Bioinformatics Advances*, 5:vbae202, 2025. doi: 10.1093/bioadv/vbae202.

[50] Danil D. Kotelnikov, Katerina S. Tatarinova, and Dmitry D. Zhdanov. NanoDeNovo: De Novo Design of Anti-Poliovirus I Sabin Strain Nanobodies by Semi-Automated Computational Pipeline. *International Journal of Molecular Sciences*, 26(19):9262, 2025. doi: 10.3390/ijms26199262. PMID:41096532.

[51] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising Diffusion Probabilistic Models. *arXiv*, 2020. URL: https://arxiv.org/abs/2006.11239.

[52] Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-Based Generative Modeling through Stochastic Differential Equations. *arXiv*, 2020. URL: https://arxiv.org/abs/2011.13456.

[53] Joseph L. Watson, David Juergens, Nathaniel R. Bennett, Brian L. Trippe, Jason Yim, Helen E. Eisenach, Woody Ahern, Andrew J. Borst, Robert J. Ragotte, Lukas F. Milles, Basile I. M. Wicky, Nikita Hanikel, Samuel J. Pellock, Alexis Courbet, William Sheffler, Jue Wang, Preetham Venkatesh, Isaac Sappington, Susana Vázquez Torres, Anna Lauko, Valentin De Bortoli, Emile Mathieu, Sergey Ovchinnikov, Regina Barzilay, Tommi S. Jaakkola, Frank DiMaio, Minkyung Baek, and David Baker. De novo design of protein structure and function with RFdiffusion. *Nature*, 620(7976):1089–1100, 2023. doi: 10.1038/s41586-023-06415-8. PMID:37433327.

[54] John B. Ingraham, Max Baranov, Zak Costello, Karl W. Barber, Wujie Wang, Ahmed Ismail, Vincent Frappier, Dana M. Lord, Christopher Ng-Thow-Hing, Erik R. Van Vlack, Shan Tie, Vincent Xue, Sarah C. Cowles, Alan Leung, João V. Rodrigues, Claudio L. Morales-Perez, Alex M. Ayoub, Robin Green, Katherine Puentes, Frank Oplinger, Nishant V. Panwar, Fritz Obermeyer, Adam R. Root, Andrew L. Beam, Frank J. Poelwijk, and Gevorg Grigoryan. Illuminating protein space with a programmable generative model. Nature, 2023.

[55] Yaron Lipman, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. Flow Matching for Generative Modeling. *arXiv*, 2022. URL: https://arxiv.org/abs/2210.02747.

[56] Michael S. Albergo and Eric Vanden-Eijnden. Building Normalizing Flows with Stochastic Interpolants. *arXiv*, 2022. URL: https://arxiv.org/abs/2209.15571.

[57] Xingchao Liu, Chengyue Gong, and Qiang Liu. Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow. *arXiv*, 2022. URL: https://arxiv.org/abs/2209.03003.

[58] Tomas Geffner, Kieran Didi, Zuobai Zhang, Danny Reidenbach, Zhonglin Cao, Jason Yim, Mario Geiger, Christian Dallago, Emine Kucukbenli, Arash Vahdat, and Karsten Kreis. Proteina: Scaling Flow-based Protein Structure Generative Models. *arXiv*, 2025. URL: https://arxiv.org/abs/2503.00710.

[59] Michael S. Albergo, Nicholas M. Boffi, and Eric Vanden-Eijnden. Stochastic Interpolants: A Unifying Framework for Flows and Diffusions. *arXiv*, 2023. URL: https://arxiv.org/abs/2303.08797.

[60] Alexander Tong, Kilian Fatras, Nikolay Malkin, Guillaume Huguet, Yanlei Zhang, Jarrid Rector-Brooks, Guy Wolf, and Yoshua Bengio. Improving and generalizing flow-based generative models with minibatch optimal transport. *arXiv*, 2023. URL: https://arxiv.org/abs/2302.00482.

[61] Jason Yim, Andrew Campbell, Andrew Y. K. Foong, Michael Gastegger, José Jiménez-Luna, Sarah Lewis, Victor Garcia Satorras, Bastiaan S. Veeling, Regina Barzilay, Tommi Jaakkola, and Frank Noé. Fast protein backbone generation with SE(3) flow matching. *arXiv*, 2023. URL: https://arxiv.org/abs/2310.05297.

[62] Jason Yim, Brian L. Trippe, Valentin De Bortoli, Emile Mathieu, Arnaud Doucet, Regina Barzilay, and Tommi Jaakkola. SE(3) diffusion model with application to protein backbone generation. *arXiv*, 2023. URL: https://arxiv.org/abs/2302.02277.

[63] Ricky T. Q. Chen and Yaron Lipman. Flow Matching on General Geometries. *arXiv*, 2023. URL: https://arxiv.org/abs/2302.03660.

[64] Avishek Joey Bose, Tara Akhound-Sadegh, Guillaume Huguet, Kilian Fatras, Jarrid Rector-Brooks, Cheng-Hao Liu, Andrei Cristian Nica, Maksym Korablyov, Michael Bronstein, and Alexander Tong. SE(3)-Stochastic Flow Matching for Protein Backbone Generation. *arXiv*, 2023. URL: https://arxiv.org/abs/2310.02391.

[65] Andrew Campbell, Jason Yim, Regina Barzilay, Tom Rainforth, and Tommi Jaakkola. Generative Flows on Discrete State-Spaces: Enabling Multimodal Flows with Applications to Protein Co-Design. *arXiv*, 2024. URL: https://arxiv.org/abs/2402.04997.

[66] Guillaume Huguet, James Vuckovic, Kilian Fatras, Eric Thibodeau-Laufer, Pablo Lemos, Riashat Islam, Cheng-Hao Liu, Jarrid Rector-Brooks, Tara Akhound-Sadegh, Michael Bronstein, Alexander Tong, and Avishek Joey Bose. Sequence-Augmented SE(3)-Flow Matching For Conditional Protein Backbone Generation. *arXiv*, 2024. URL: https://arxiv.org/abs/2405.20313.

[67] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-Resolution Image Synthesis with Latent Diffusion Models. *arXiv*, 2021. URL: https://arxiv.org/abs/2112.10752.

[68] Tomas Geffner, Kieran Didi, Zhonglin Cao, Danny Reidenbach, Zuobai Zhang, Christian Dallago, Emine Kucukbenli, Karsten Kreis, and Arash Vahdat. La-Proteina: Atomistic Protein Generation via Partially Latent Flow Matching. *arXiv*, 2025. URL: https://arxiv.org/abs/2507.09466.

[69] Amy X. Lu, Jason Yang, Yueming Long, Theophile Lambert, Wilson Yan, Sarah A. Robinson, Kevin K. Yang, Vladimir Gligorijevic, Kyunghyun Cho, Richard Bonneau, Nathan C. Frey, Frances Arnold, and Pieter Abbeel. Controllable All-Atom Protein Generation with Latent Diffusion. *bioRxiv*, 2024. doi: 10.1101/2024.12.02.626353.

[70] Jason Yim, Marouane Jaakik, Ge Liu, Jacob Gershon, Karsten Kreis, David Baker, Regina Barzilay, and Tommi Jaakkola. Hierarchical protein backbone generation with latent and structure diffusion. *arXiv*, 2025. URL: https://arxiv.org/abs/2504.09374.

[71] Kieran Didi, Zuobai Zhang, Guoqing Zhou, Danny Reidenbach, Zhonglin Cao, Sooyoung Cha, Tomas Geffner, Christian Dallago, Jian Tang, Michael M. Bronstein, Martin Steinegger, Emine Kucukbenli, Arash Vahdat, and Karsten Kreis. Scaling Atomistic Protein Binder Design with Generative Pretraining and Test-Time Compute. *arXiv*, 2026. URL: https://arxiv.org/abs/2603.27950.

[72] Shitong Luo, Yufeng Su, Xingang Peng, Sheng Wang, Jian Peng, and Jianzhu Ma. Antigen-Specific Antibody Design and Optimization with Diffusion-Based Generative Models for Protein Structures. *bioRxiv*, 2022. doi: 10.1101/2022.07.10.499510.

[73] Karolis Martinkus, Jan Ludwiczak, Kyunghyun Cho, Wei-Ching Liang, Julien Lafrance-Vanasse, Isidro Hotzel, Arvind Rajpal, Yan Wu, Richard Bonneau, Vladimir Gligorijevic, and Andreas Loukas. AbDiffuser: Full-Atom Generation of in vitro Functioning Antibodies. *arXiv*, 2023. URL: https://arxiv.org/abs/2308.05027.

[74] Daniel Cutting, Frédéric A. Dreyer, David Errington, Constantin Schneider, and Charlotte M. Deane. De novo antibody design with SE(3) diffusion. *arXiv*, 2024. URL: https://arxiv.org/abs/2405.07622.

[75] Amir Shanehsazzadeh, Sharrol Bachas, Matt McPartlon, George Kasun, John M. Sutton, Andrea K. Steiger, Richard Shuai, Christa Kohnert, Goran Rakocevic, Jahir M. Gutierrez, Chelsea Chung, Breanna K. Luton, Nicolas Diaz, Simon Levine, Julian Alverio, Bailey Knight, Macey Radach, Alex Morehead, Katherine Bateman, David A. Spencer, Zachary McDargh, Jovan Cejovic, Gaelin Kopec-Belliveau, Robel Haile, Edriss Yassine, Cailen McCloskey, Monica Natividad, Dalton Chapman, Joshua Bennett, Jubair Hossain, Abigail B. Ventura, Gustavo M. Canales, Muttappa Gowda, Kerianne A. Jackson, Jennifer T. Stanton, Marcin Ura, Luka Stojanovic, Engin Yapici, Katherine Moran, Rodante Caguiat, Amber Brown, Shaheed Abdulhaqq, Zheyuan Guo, Lillian R. Klug, Miles Gander, and Joshua Meier. Unlocking de novo antibody design with generative artificial intelligence. *bioRxiv*, 2023. doi: 10.1101/2023.01.08.523187.

[76] Rubo Wang, Fandi Wu, Xingyu Gao, Jiaxiang Wu, Peilin Zhao, and Jianhua Yao. IgGM: A Generative Model for Functional Antibody and Nanobody Design. *bioRxiv*, 2024. doi: 10.1101/2024.09.19.613838.

[77] Rubo Wang, Fandi Wu, Jiale Shi, Yidong Song, Yu Kong, Jian Ma, Bing He, Qihong Yan, Tianlei Ying, Peilin Zhao, Xingyu Gao, and Jianhua Yao. A Generative Foundation Model for Antibody Design. *bioRxiv*, 2025. doi: 10.1101/2025.09.12.675771.

[78] Nathaniel R. Bennett, Joseph L. Watson, Robert J. Ragotte, Andrew J. Borst, DéJenaé L. See, Connor Weidle, Riti Biswas, Yutong Yu, Ellen L. Shrock, Russell Ault, Philip J. Y. Leung, Buwei Huang, Inna Goreshnik, John Tam, Kenneth D. Carr, Benedikt Singer, Cameron Criswell, Basile I. M. Wicky, Dionne Vafeados, Mariana Garcia Sanchez, Ho Min Kim, Susana Vázquez Torres, Sidney Chan, Shirley M. Sun, Timothy T. Spear, Yi Sun, Keelan O'Reilly, John M. Maris, Nikolaos G. Sgourakis, Roman A. Melnyk, Chang C. Liu, and David Baker. Atomically accurate de novo design of antibodies with RFdiffusion. *Nature*, 649(8095):183–193, 2026. doi: 10.1038/s41586-025-09721-5. PMID:41193805.

[79] Nathaniel R. Bennett, Joseph L. Watson, Robert J. Ragotte, Andrew J. Borst, DéJenaé L. See, Connor Weidle, Riti Biswas, Yutong Yu, Ellen L. Shrock, Russell Ault, Philip J. Y. Leung, Buwei Huang, Inna Goreshnik, John Tam, Kenneth D. Carr, Benedikt Singer, Cameron Criswell, Basile I. M. Wicky, Dionne Vafeados, Mariana Garcia Sanchez, Ho Min Kim, Susana Vázquez Torres, Sidney Chan, Shirley M. Sun, Timothy Spear, Yi Sun, Keelan O'Reilly, John M. Maris, Nikolaos G. Sgourakis, Roman A. Melnyk, Chang C. Liu, and David Baker. Atomically accurate de novo design of antibodies with RFdiffusion. *bioRxiv*, 2024. doi: 10.1101/2024.03.14.585103.

[80] Luis S. Mille-Fragoso, Claudia L. Driscoll, John N. Wang, Haoyu Dai, Talal Widatalla, Jim L. Zhang, Xiaowei Zhang, Bing Rao, Liang Feng, Brian L. Hie, and Xiaojing J. Gao. Efficient generation of epitope-targeted antibodies with Germinal. *Nature Biotechnology*, 2026. doi: 10.1038/s41587-026-03187-0. PMID:42337361.

[81] Luis S. Mille-Fragoso, Claudia L. Driscoll, John N. Wang, Haoyu Dai, Talal Widatalla, Jim L. Zhang, Xiaowei Zhang, Bing Rao, Liang Feng, Brian L. Hie, and Xiaojing J. Gao. Efficient generation of

epitope-targeted de novo antibodies with Germinal. *bioRxiv*, 2025. doi: 10.1101/2025.09.19.677421.

[82] Erik Swanson, Michael Nichols, Supriya Ravichandran, and Pierce Ogden. mBER: Controllable de novo antibody design with million-scale experimental screening. *bioRxiv*, 2025. doi: 10.1101/2025.09.26.678877.

[83] Hannes Stark, Felix Faltings, MinGyu Choi, Yuxin Xie, Eunsu Hur, Timothy O'Donnell, Anton Bushuiev, Talip Uçar, Saro Passaro, Weian Mao, Mateo Reveiz, Roman Bushuiev, Tally Portnoi, Tomáš Pluskal, Josef Sivic, Karsten Kreis, Arash Vahdat, Shamayeeta Ray, Jonathan T. Goldstein, Andrew Savinov, Jacob A. Hambalek, Anshika Gupta, Diego A. Taquiri-Diaz, Yaotian Zhang, Samuel J. Snyder, A. Katherine Hatstat, Angelika Arada, Nam Hyeong Kim, Haoyu Fan, Ethel Tackie-Yarboi, Dylan Boselli, Lee Schnaider, Chang C. Liu, Gene-Wei Li, Denes Hnisz, David M. Sabatini, William F. DeGrado, Jeremy Wohlwend, Gabriele Corso, Regina Barzilay, and Tommi Jaakkola. BoltzGen: Toward Universal Binder Design. *bioRxiv*, 2025. doi: 10.1101/2025.11.20.689494.

[84] Chai Discovery Team, Jacques Boitreaud, Jack Dent, Danny Geisz, Matthew McPartlon, Joshua Meier, Zhuoran Qiao, Alex Rogozhnikov, Nathan Rollins, Paul Wollenhaupt, and Kevin Wu. Zero-shot antibody design in a 24-well plate. *bioRxiv*, 2025. doi: 10.1101/2025.07.05.663018.

[85] Chai Discovery Team, Jacques Boitreaud, Robert Chen, Jack Dent, Lucy Fairweather, Danny Geisz, Matthew Greenig, Nate Boyd, Jinay Jain, Brady Johnston, Matthew McPartlon, Joshua Meier, Neil Patil, Zhuoran Qiao, Alex Rogozhnikov, Nathan Rollins, Nikitha Vicas, Paul Wollenhaupt, Kevin Wu, and Andy Yeung. Drug-like antibody design against challenging targets with atomic precision. *bioRxiv*, 2025. doi: 10.1101/2025.11.29.691346.

[86] Nabla Bio and Surojit Biswas. De novo design of epitope-specific antibodies against soluble and multipass membrane proteins with high specificity, developability, and function. *bioRxiv*, 2025. doi: 10.1101/2025.01.21.633066.

[87] Latent Labs Team, Henry Kenlay, Daniella Pretorius, Jonathan Crabbé, Alex Bridgland, Sebastian M. Schmon, Agrin Hilmkil, James Vuckovic, Simon Mathis, Tomas Matteson, Rebecca Bartke-Croughan, Amir Motmaen, Robin Rombach, Mária Vlachynská, Alexander W. R. Nelson, David Yuan, Annette Obika, and Simon A. A. Kohl. Drug-like antibodies with low immunogenicity in human panels designed with Latent-X2. *arXiv*, 2025. URL: https://arxiv.org/abs/2512.20263.

[88] Simon Levine, Jonathan Edward King, Jacob Stern, David Grayson, Raymond Wang, Rui Yin, Umberto Lupo, Paulina Kulytė, Ryan Matthew Brand, Tristan Bertin, Robert Pfingsten, Jovan Cejovic, Chelsea Chung, Breanna K. Luton, Andrew Hagemann, Robel Haile, Elliot Medina, Pankaj Panwar, Oleksii Dubrovskyi, Chase LaCombe, Zahra Anderson, Derrik Mildh, Scott Benjamin, Joe Kaiser, Joseph Ferron, Marta Sarrico, Alexandria Kershner, Apurva Mishra, Kai R. Ejan, Emily K. Marsh, Paul Bringas, Phetsamay Vilaychack, Kyra Chapman, Jacob Ripley, Muttappa Gowda, Kathryn M. Collins, Cailen M. McCloskey, Jeremiah S. Joseph, Rylee Ripley, Shaheed A. Abdulhaqq, David Spencer, Tiffany DeVine, Audree Feltner, Michael Guerin, Jeffrey Goby, Jesse Hendricks, Danielle Castillo, Sean McClain, Douglas Ganini, Derek Shpiel, James Mategko, Eder Cruz Garcia, Masoud Zabet-Moghaddam, John M. Sutton, Zheyuan Guo, Sean M. West, Janani S. Iyer, and Amir Shanehsazzadeh. Origin-1: a generative AI platform for de novo antibody design against novel epitopes. *bioRxiv*, 2026. doi: 10.64898/2026.01.14.699389.

[89] Fandi Wu, Yu Zhao, JiaXiang Wu, Biaobin Jiang, Bing He, Longkai Huang, Chenchen Qin, Yang Xiao, Fan Yang, Rubo Wang, Ningqiao Huang, Huaxian Jia, Yuyi Liu, Houtim Lai, Tingyang Xu, Fang Wang, Zihan Wu, Yidong Song, Shaoning Li, Wei Liu, Yu Rong, Peilin Zhao, and Jianhua Yao. De novo design of epitope-specific antibodies via a structure-driven computational workflow. *Nature Communications*, 17(1):625, 2025. doi: 10.1038/s41467-025-67361-9. PMID:41365923.

[90] BioGeometry Team and Jian Tang. Rapid De Novo Antibody Design with GeoFlow-V3. *bioRxiv*, 2025. doi: 10.1101/2025.10.20.682964.

[91] Hyeonjin Cha, Kyesoo Cho, Jeonghyeon Gu, Daehyeon Gwak, Seung Woo Ham, Mirim Hong, Sehan Kim, Suyeon Kim, Sohee Kwon, Changsoo Lee, Da Kyoung Lee, Danyeong Lee, Dohoon Lee, Jungsub

Lim, Jinsung Noh, Soyeon Oh, Eunhwi Park, Seongchan Park, Taeyong Park, Eunwoo Ryu, Seongok Ryu, Deok Hyang Sa, Chaok Seok, Jiho Sim, Moo Young Song, Jonghun Won, Hyeonuk Woo, and Jinsol Yang. Precision at Every Scale: Efficiency in AI-Driven De Novo Antibody Design. *bioRxiv*, 2025. doi: 10.1101/2025.11.21.689414.

[92] Jian Ma, Fandi Wu, Tingyang Xu, Shaoyong Xu, Wei Liu, and Liang Yan. An adaptive autoregressive diffusion approach to design active humanized antibodies and nanobodies. *Nature Machine Intelligence*, 2025. doi: 10.1038/s42256-025-01120-9.

[93] Yu Kong, Jiale Shi, Fandi Wu, Ting Zhao, Rubo Wang, Xiaoyi Zhu, Qingyuan Xu, Yidong Song, Quanxiao Li, Yulu Wang, Xingyu Gao, Yuedong Yang, Yi Feng, Zifei Wang, Weifeng Ge, Yanling Wu, Zhenlin Yang, Jianhua Yao, and Tianlei Ying. A synergistic generative-ranking framework for tailored design of therapeutic single-domain antibodies. *Cell Discovery*, 11(1):85, 2025. doi: 10.1038/s41421-025-00843-8. PMID:41162386.

[94] Xiangzhe Kong, Zishen Zhang, Ziting Zhang, Rui Jiao, Jianzhu Ma, Wenbing Huang, Kai Liu, and Yang Liu. UniMoMo: Unified Generative Modeling of 3D Molecules for De Novo Binder Design. *arXiv*, 2025. URL: https://arxiv.org/abs/2503.19300.

[95] Mansoor Ahmed and Murray Patterson. AgForce Enables Antigen-conditioned Generative Antibody Design. *arXiv*, 2026. URL: https://arxiv.org/abs/2605.21610.

[96] Yiheng Zhu, Jian Ma, Mingze Yin, Jialu Wu, Lin Tang, Zhiyun Zhang, Qiuyi Li, Shikun Feng, Haiguang Liu, Tao Qin, Junchi Yan, Chang-Yu Hsieh, and Tingjun Hou. Ophiuchus-Ab: A Versatile Generative Foundation Model for Advanced Antibody-Based Immunotherapy. *bioRxiv*, 2026. doi: 10.64898/2026.02.02.703197.

[97] M. Frank Erasmus, Daniel Bedinger, Elizabeth Hopkins, Ginger Ferguson, Justine Strickler, Christilyn P. Graff, Samantha R. Summers, Stacy L. Capehart, Joshua D. Slocum, Crystal Richardson, Sumit Kumar, Zhifei Sun, Yujie Shang, Jixian Zhang, Ming Gu, Lixia Yi, Alon Wellner, Shuangjia Zheng, Wei Lu, Pietro Sormanni, Matthew Greenig, Haiping Zhang, Brendan T. Mann, Mahdi Baghbanzadeh, Ali Rahnavard, Gregory L. Moore, Huaiyu Sun, Ying Ding, Alex Nisthal, Jitendra Kanodia, Matthew J. Bernett, Aurélien Pélissier, Yanjun Shao, Maria Rodriguez Martinez, Karthik Ramesh, Horacio Nastri, Andreas Evers, Anhar Abdelatif, Andrew J. Bordner, Mykola Bordyuh, Lim Heo, Brian A. Kidd, H. Serhat Tetikol, Shuai Wei, Jung-Eun Shin, Ryan Peckner, Leigh Manley, Ajitesh Lunge, Yashas Devasurmutt, Bora Guloglu, Liviu Copoiu, Miles McGibbon, Monica L. Fernandez-Quintero, Nitesh Mishra, Sean M. Callaghan, Olivia M. Swanson, Daniel L. V. Bader, James A. Ferguson, Sai S. R. Raghavan, Benjamin Nemoz, Colleen A. Maillie, Charles Bowman, Bryan Briney, Andrew B. Ward, Paolo Marcatili, Rahmad Akbar, Bing He, Fandi Wu, Jianhua Yao, Bin Hu, Michal Kucer, Kaetlyn Rose Gibson, Rahul Somasundaram, Li-Wei Hung, Tomasz Kaszuba, Daved H. Fremont, Hyeongsun Jeong, Vinodh Babu Kurella, Shipra Malhotra, Satyendra Kumar, Yanyun Liu, Lingling Xu, Joshua Misa, Alexander Nicholas St. John, Jeff Vogt, Fátima A. Dávila-Hernández, Da Xu, Michael Chungyoun, Zyaja D. Huggan, Jeffrey J. Gray, Jonathan Parkinson, Young Su Ko, Wei Wang, Franziska Geiger, Jonathon D. Ziegler, Nikhil Haas, Chance Challacombe, Ahmad Qamar, Akshita Singh, Yi-Ching Tang, Zhiqiang An, Xiaoqian Jiang, Yejin Kim, Xinyan Zhao, Erik Swanson, Jürgen Klattig, Karsten Winkler, Tschimegma Bataa, Volker Sandig, Lilian Denzler, Chunan Liu, Randall J. Brezski, Laura Spector, Katheryn Perea-Schmittle, Sara D'Angelo, Fortunato Ferrara, and Andrew R. M. Bradbury. A blinded, prospective benchmark of in silico antibody discovery anchored to experimental affinity and developability. *Nature Biotechnology*, 2026. doi: 10.1038/s41587-026-03238-6. PMID:42618805.

[98] M. Frank Erasmus, Daniel Bedinger, Elizabeth Hopkins, Ginger Ferguson, Justine Strickler, and Christilyn P. Graff. A blinded, prospective benchmark of in silico antibody discovery anchored to experimental affinity and developability. *Nature Biotechnology*, 2026. doi: 10.1038/s41587-026-03238-6.

[99] Mansoor Ahmed, Nadeem Taj, Imdad Ullah Khan, Hemanth Venkateswara, and Murray Patterson. CHIMERA-Bench: A Benchmark Dataset for Epitope-Specific Antibody Design. *arXiv*, 2026. URL: https://arxiv.org/abs/2603.13431.

[100] Nathaniel R. Bennett, Brian Coventry, Inna Goreshnik, Buwei Huang, Aza Allen, Dionne Vafeados, Ying Po Peng, Justas Dauparas, Minkyung Baek, Lance Stewart, Frank DiMaio, Steven De Munck, Savvas N. Savvides, and David Baker. Improving de novo protein binder design with deep learning. *Nature Communications*, 14:2625, 2023. doi: 10.1038/s41467-023-38328-5.

[101] Anton Bushuiev, Roman Bushuiev, Jiri Sedlar, Tomas Pluskal, Jiri Damborsky, Stanislav Mazurenko, and Josef Sivic. Revealing data leakage in protein interaction benchmarks. *arXiv*, 2024. URL: https://arxiv.org/abs/2404.10457.

[102] Leon Hermann, Tobias Fiedler, Hoang An Nguyen, Melania Nowicka, and Jakub M. Bartoszewicz. Beware of Data Leakage from Protein LLM Pretraining. *bioRxiv*, 2024. doi: 10.1101/2024.07.23.604678.

[103] Abhinav Gupta, Bryan Munoz Rivero, Ruijiang Li, Jorge Roel-Touris, Yves Fomekong Nanfack, Maria Wendt, Yu Qiu, and Norbert Furtmann. SNAC - DB : An ML -ready database for antibody and NANOBODY ® VHH –antigen complexes with expanded structural diversity and real-world benchmarking. *Protein Science*, 35(7):e70655, 2026. doi: 10.1002/pro.70655. PMID:42252708.

[104] Sheng Xu, Qiantai Feng, Lifeng Qiao, Hao Wu, Tao Shen, Yu Cheng, Shuangjia Zheng, and Siqi Sun. Benchmarking all-atom biomolecular structure prediction with FoldBench. *Nature Communications*, 17(1):442, 2025. doi: 10.1038/s41467-025-67127-3. PMID:41345395.

[105] Roland L. Dunbrack. Res ipSAE loquuntur: What's wrong with AlphaFold's ipTM score and how to fix. *bioRxiv*, 2025. doi: 10.1101/2025.02.10.637595.

[106] M. D. Overath, A. Rygaard, C. P. Jacobsen, V. Brasas, O. Morell, P. Sormanni, and T. P. Jenkins. Predicting Experimental Success in De Novo Binder Design: A Meta-Analysis of 3,766 Experimentally Characterised Binders. *bioRxiv*, 2025. doi: 10.1101/2025.08.14.670059.

[107] Langzhang Liang, Juncheng Qian, Xiaojie Ma, Mingxi Zou, Yinghui Xu, and Zenglin Xu. Revisiting Structure-Based Filtering for AI-Designed Antibodies: Opportunities, Limits, and What Matters. *arXiv*, 2026. URL: https://arxiv.org/abs/2610.23136.

[108] Clara Fannjiang, Stephen Bates, Anastasios N. Angelopoulos, Jennifer Listgarten, and Michael I. Jordan. Conformal prediction under feedback covariate shift for biomolecular design. *Proceedings of the National Academy of Sciences*, 119(43):e2204569119, 2022. doi: 10.1073/pnas.2204569119. PMID:36256807.

[109] Edward P. Harvey, Jung-Eun Shin, Meredith A. Skiba, Genevieve R. Nemeth, Joseph D. Hurley, Alon Wellner, Ada Y. Shaw, Victor G. Miranda, Joseph K. Min, Chang C. Liu, Debora S. Marks, and Andrew C. Kruse. An in silico method to assess antibody fragment polyreactivity. *Nature Communications*, 13(1):7554, 2022. doi: 10.1038/s41467-022-35276-4. PMID:36477674.

[110] Benedikt Jedlitzke, Oliver Wissett, Tung H. Nguyen, Elodie Ghedin, and Pietro Sormanni. SurfDiff: protein surface profiling for selective or broadly reactive epitope prioritisation in binder and immunogen design. *bioRxiv*, 2025. doi: 10.64898/2025.12.23.696170.

[111] Valentas Brasas, Charlotte R. Christensen, Kasper H. Björnsson, Victor Emil Møller, Beatrice Scapolo, Melisa Benard-Valle, Mads Mørup Nygaard, Jens Christian Nielsen, Sine Reker Hadrup, Kristoffer H. Johansen, and Timothy P. Jenkins. Specificity-driven protein binder design with Odin-Multi. *bioRxiv*, 2026. doi: 10.64898/2026.09.08.749745.

[112] Tushar Jain, Tingwan Sun, Stéphanie Durand, Amy Hall, Nga Rewa Houston, Juergen H. Nett, Beth Sharkey, Beata Bobrowicz, Isabelle Caffry, Yao Yu, Yuan Cao, Heather Lynaugh, Michael Brown, Hemanta Baruah, Laura T. Gray, Eric M. Krauland, Yingda Xu, Maximiliano Vásquez, and K. Dane Wittrup. Biophysical properties of the clinical-stage antibody landscape. *Proceedings of the National Academy of Sciences*, 114(5):944–949, 2017. doi: 10.1073/pnas.1616408114. PMID:28096333.

[113] Michael Chungyoun and J. Robert Gray. FLAb2: Benchmarking reveals that protein AI models cannot yet consistently predict developability properties. *bioRxiv*, 2025. doi: 10.64898/2025.12.27.696706.

[114] Lood van Niekerk, Joshua Moller, Seth Ritter, Porfirio Quintero-Cadena, Rich Cohen, and Georgia

Channing. 2025 Ginkgo Datapoints Antibody Developability Competition outcomes: limited model performance and a call for data standardization. *mAbs*, 18(1):2634216, 2026. doi: 10.1080/19420862.2026.2634216.

[115] Jochem Gokemeijer, Vibha Jawa, and Shibani Mitra-Kaushik. How Close Are We to Profiling Immunogenicity Risk Using In Silico Algorithms and In Vitro Methods?: an Industry Perspective. *The AAPS Journal*, 19:1587–1592, 2017. doi: 10.1208/s12248-017-0143-z.

[116] A. Wright, M. H. Tao, E. A. Kabat, and S. L. Morrison. Antibody variable region glycosylation: position effects on antigen binding and carbohydrate structure. *The EMBO Journal*, 10:2717–2723, 1991. doi: 10.1002/j.1460-2075.1991.tb07819.x.

[117] Gwo-Yu Chuang, Mangaiarkarasi Asokan, Vera B. Ivleva, Amarendra Pegu, Eun Sung Yang, Baoshan Zhang, Rajoshi Chaudhuri, Hui Geng, Bob C. Lin, Mark K. Louder, Krisha McKee, Sijy O'Dell, Hairong Wang, Tongqing Zhou, Nicole A. Doria-Rose, Lisa A. Kueltzo, Q. Paula Lei, John R. Mascola, and Peter D. Kwong. Removal of variable domain-linked glycosylation as a means to improve the homogeneity of HIV-1 broadly neutralizing antibodies. *mAbs*, 12:1836719, 2020. doi: 10.1080/19420862.2020.1836719.

[118] Kieran Didi, Zuobai Zhang, Guoqing Zhou, Danny Reidenbach, Zhonglin Cao, Sooyoung Cha, Tomas Geffner, Christian Dallago, Jian Tang, Michael M. Bronstein, Martin Steinegger, Emine Kucukbenli, Arash Vahdat, and Karsten Kreis. Scaling Atomistic Protein Binder Design with Generative Pretraining and Test-Time Compute. *arXiv*, 2026. URL: https://arxiv.org/abs/2603.27950.

[119] Alexander Greenshields-Watson, Parth Agarwal, Sarah A. Robinson, Benjamin Heathcote Williams, Gemma L. Gordon, Henriette L. Capel, Yushi Li, Fabian C. Spoendlin, Broncio Aguilar-Sanjuan, Fergus Boyles, and Charlotte M. Deane. ANARCII enables alignment-free antigen receptor numbering using a generalised language model. *Communications Biology*, 9(1):1085, 2026. doi: 10.1038/s42003-026-10186-z. PMID:42162238.

[120] Jian Ye, Ning Ma, Thomas L. Madden, and James M. Ostell. IgBLAST: an immunoglobulin variable domain sequence analysis tool. *Nucleic Acids Research*, 41(W1):W34–W40, 2013. doi: 10.1093/nar/gkt382. PMID:23671333.

[121] Dongjun Guo, Deborah K. Dunn-Walters, Franca Fraternali, and Joseph C. F. Ng. ImmunoMatch learns and predicts cognate pairing of heavy and light immunoglobulin chains. *Nature Methods*, 23(1):106–117, 2026. doi: 10.1038/s41592-025-02913-x. PMID:41254366.

[122] H. M. Berman. The Protein Data Bank. *Nucleic Acids Research*, 28(1):235–242, 2000. doi: 10.1093/nar/28.1.235. PMID:10592235.

[123] Mihaly Varadi, Stephen Anyango, Mandar Deshpande, Sreenath Nair, Cindy Natassia, Galabina Yordanova, David Yuan, Oana Stroe, Gemma Wood, Agata Laydon, Augustin Žídek, Tim Green, Kathryn Tunyasuvunakool, Stig Petersen, John Jumper, Ellen Clancy, Richard Green, Ankur Vora, Mira Lutfi, Michael Figurnov, Andrew Cowie, Nicole Hobbs, Pushmeet Kohli, Gerard Kleywegt, Ewan Birney, Demis Hassabis, and Sameer Velankar. AlphaFold Protein Structure Database: massively expanding the structural coverage of protein-sequence space with high-accuracy models. *Nucleic Acids Research*, 50(D1):D439–D444, 2022. doi: 10.1093/nar/gkab1061. PMID:34791371.

[124] Andy M. Lau, Nicola Bordin, Shaun M. Kandathil, Ian Sillitoe, Vaishali P. Waman, Jude Wells, Christine A. Orengo, and David T. Jones. Exploring structural diversity across the protein universe with The Encyclopedia of Domains. *Science*, 386:eadq4946, 2024. doi: 10.1126/science.adq4946.

[125] Philippe A. Robert, Rahmad Akbar, Robert Frank, Milena Pavlović, Michael Widrich, Igor Snapkov, Andrei Slabodkin, Maria Chernigovskaya, Lonneke Scheffer, Eva Smorodina, Puneet Rawat, Brij Bhushan Mehta, Mai Ha Vu, Ingvild Frøberg Mathisen, Aurél Prósz, Krzysztof Abram, Alex Olar, Enkelejda Miho, Dag Trygve Tryslew Haug, Fridtjof Lund-Johansen, Sepp Hochreiter, Ingrid Hobæk Haff, Günter Klambauer, Geir Kjetil Sandve, and Victor Greiff. Unconstrained generation of synthetic antibody–antigen structures to guide machine learning methodology for antibody specificity prediction. *Nature Computational Science*, 2(12):845–865, 2022. doi: 10.1038/s43588-022-00372-4.

PMID:38177393.

[126] Yikai Wu, Xuejiao Liu, Karin Hrovatin, Dezhi Wu, Stephanie Maria Linker, and Mathias Winkel. A Unified Dataset for Antibody and Nanobody Design Including Sequence, Structure, and Binding Affinity Data. *Scientific Data*, 13, 2026. doi: 10.1038/s41597-026-06878-0.

[127] Martin Steinegger and Johannes Söding. Clustering huge protein sequence sets in linear time. *Nature Communications*, 9(1):2542, 2018. doi: 10.1038/s41467-018-04964-5. PMID:29959318.

[128] Michel van Kempen, Stephanie S. Kim, Charlotte Tumescheit, Milot Mirdita, Jeongjae Lee, Cameron L. M. Gilchrist, Johannes Söding, and Martin Steinegger. Fast and accurate protein structure search with Foldseek. *Nature Biotechnology*, 42(2):243–246, 2024. doi: 10.1038/s41587-023-01773-0. PMID:37156916.

[129] Roman Joeres, David B. Blumenthal, and Olga V. Kalinina. Data splitting to avoid information leakage with DataSAIL. *Nature Communications*, 16(1):3337, 2025. doi: 10.1038/s41467-025-58606-8. PMID:40199913.

[130] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural Ordinary Differential Equations. *arXiv*, 2018. URL: https://arxiv.org/abs/1806.07366.

[131] William Peebles and Saining Xie. Scalable Diffusion Models with Transformers. *arXiv*, 2022. URL: https://arxiv.org/abs/2212.09748.

[132] Noam Shazeer. GLU Variants Improve Transformer. *arXiv*, 2020. URL: https://arxiv.org/abs/2002.05202.

[133] Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models. *arXiv*, 2018. URL: https://arxiv.org/abs/1810.01367.

[134] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *arXiv*, 2017. URL: https://arxiv.org/abs/1711.05101.

[135] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv*, 2020. URL: https://arxiv.org/abs/2006.15704.

[136] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A Study of BFLOAT16 for Deep Learning Training. *arXiv*, 2019. URL: https://arxiv.org/abs/1905.12322.

[137] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv*, 2016. URL: https://arxiv.org/abs/1604.06174.

[138] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing Activation Recomputation in Large Transformer Models. *arXiv*, 2022. URL: https://arxiv.org/abs/2205.05198.

[139] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv*, 2019. URL: https://arxiv.org/abs/1912.01703.

[140] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi: 10.1038/s41586-020-2649-2.

[141] Saba Ferdous and Andrew C R Martin. AbDb: antibody structure database—a database of PDB-derived antibody structures. *Database*, 2018, 2018. doi: 10.1093/database/bay040. PMID:29718130.

[142] Matthew I J Raybould, Claire Marks, Alan P Lewis, Jiye Shi, Alexander Bujotzek, Bruck Taddese, and Charlotte M Deane. Thera-SAbDab: the Therapeutic Structural Antibody Database. *Nucleic Acids Research*, 48(D1):D383–D388, 2020. doi: 10.1093/nar/gkz827. PMID:31555805.

[143] Brennan Abanades, Tobias H Olsen, Matthew I J Raybould, Broncio Aguilar-Sanjuan, Wing Ki Wong, Guy Georges, Alexander Bujotzek, and Charlotte M Deane. The Patent and Literature Antibody Database (PLAbDab): an evolving reference set of functionally diverse, literature-annotated antibody sequences and structures. *Nucleic Acids Research*, 52(D1):D545–D551, 2024. doi: 10.1093/nar/gkad1056. PMID:37971316.

[144] Justina Jankauskaitė, Brian Jiménez-García, Justas Dapkūnas, Juan Fernández-Recio, and Iain H Moal. SKEMPI 2.0: an updated benchmark of changes in protein–protein binding energy, kinetics and thermodynamics upon mutation. *Bioinformatics*, 35(3):462–469, 2019. doi: 10.1093/bioinformatics/bty635. PMID:30020414.

[145] Sarah Sirin, James R. Apgar, Eric M. Bennett, and Amy E. Keating. AB -Bind: Antibody binding mutational database for computational affinity predictions. *Protein Science*, 25(2):393–409, 2016. doi: 10.1002/pro.2829. PMID:26473627.

[146] Xinyan Zhao, Yi-Ching Tang, Akshita Singh, Victor J Cantu, KwanHo An, Junseok Lee, Adam E Stogsdill, Ibraheem M Hamdi, Ashwin Kumar Ramesh, Zhiqiang An, Xiaoqian Jiang, and Yejin Kim. AbBiBench: A Benchmark for Antibody Binding Affinity Maturation and Design. *arXiv*, 2025. URL: https://arxiv.org/abs/2506.04235.

[147] Fabio Urbina, Filippa Lentzos, Cédric Invernizzi, and Sean Ekins. Dual Use of Artificial Intelligence-powered Drug Discovery. *Nature Machine Intelligence*, 4:189–191, 2022. doi: 10.1038/s42256-022-00465-9.

[148] Bruce J. Wittmann, Tessa Alexanian, Craig Bartling, Jacob Beal, Adam Clore, James Diggans, Kevin Flyangolts, Bryan T. Gemler, Tom Mitchell, Steven T. Murphy, Nicole E. Wheeler, and Eric Horvitz. Strengthening nucleic acid biosecurity screening against generative protein design tools. *Science*, 390:82–87, 2025. doi: 10.1126/science.adu8578.

[149] David Baker and George Church. Protein design meets biosecurity. *Science*, 383:349, 2024. doi: 10.1126/science.ado1671.