DESIGN AND IMPLEMENTATION

Protein software architecture

Extractor, prediction models and Reader

How the programs store atomic data, calculate predictions and display results.

Working draft

24 September 2026

Reading this book

The extractor calculates molecular properties from structures. The prediction models use those properties to estimate shielding tensors and chemical shifts. Reader displays the structures and their measurements. This book describes the main objects and calculations in each program.

The extractor's core classes form a domain-specific language within C++. Calculators use these classes to identify atoms by chemical role, follow bonds and ring membership, and read or write values for a particular pose. Chapter 1 starts with a ring-current example to show how this vocabulary is used. The prediction software and Reader read the extracted data and build their own representations for learning and display.

1. **Extractor.** Calculators read chemical types, connectivity and coordinates from the protein and its conformation. Some calculate values for one pose; others accumulate results across trajectory frames.
2. **Prediction models.** An equivariant encoder represents each atom's environment. Separate models use that representation to predict shielding tensors and chemical shifts.
3. **Reader.** The application loads the extractor's files into its own molecular objects. Qt signals notify the displays when the current frame or selection changes.

The appendices trace a prediction call through the code, report parameter counts for the fitted networks, and list the Reader source files behind the diagrams. Class and function names connect the explanations to the code. The diagrams show selected relationships rather than every class member.

Appendix D identifies the versions described and the files used to build this book.

Contents

Reading this book i

1 Extractor 1

1.1 A language for molecular calculations . . . 1
1.2 Reading the class diagrams . . . 2
1.3 Identity and a pose . . . 2
1.4 Names, roles and grouping . . . 4
1.5 The same atom across buffers . . . 6
1.6 Pose calculators and their results . . . 8
1.7 Calculations across frames . . . 10
1.8 What lives for the whole trajectory . . . 12
1.9 One frame through the system . . . 14
1.10 Main objects . . . 16

2 Prediction models 17

2.1 The three predictions . . . 17
2.2 Preparation and units . . . 19
2.3 Encoder architecture . . . 19
2.4 Shielding readout . . . 21
2.5 Shift readout and chemical roles . . . 23
2.6 Transferring weights from shielding to shifts . . . 25

3 Reader 26

3.1 The same atom throughout . . . 26
3.2 Molecular identity and frame data . . . 28
3.3 The main window owns the loaded data . . . 30
3.4 Display coordinates . . . 32
3.5 Qt connects the views . . . 32
3.6 The correspondence with the extractor . . . 34

A From code to prediction 35

A.1 Input arrays and atom indices . . . 35
A.2 Inside one message layer . . . 37

A.3 The encoder builds on that layer . . . 39
A.4 Constructing and calling the models . . . 39
A.5 Follow one prediction call . . . 40
A.6 Where to read the code . . . 42

B Inside the fitted networks **43**
B.1 Parameter counts from the fitted models . . . 43
B.2 Reading the e3nn diagrams . . . 46
B.3 Dependencies . . . 49
B.4 Source and fitted files . . . 49

C Reader source map **50**

D Sources and working files **52**
D.1 The implementations described . . . 52
D.2 The documentation project . . . 52
D.3 Diagrams and evidence . . . 52
D.4 Rebuilding . . . 52

Chapter 1

Extractor

The extractor reads molecular structures and calculates properties for their atoms, residues, bonds and rings. Its core classes provide the language in which those calculations are written.

The diagrams show the main classes and representative calculators. Method signatures include only the arguments needed to explain each relationship.

1.1 A language for molecular calculations

The library is an embedded domain-specific language: its C++ types and operations let a calculator work with molecular concepts directly. `Protein` holds atoms, residues and topology; `ProteinConformation` holds coordinates and results for one pose. Chemical roles distinguish, for example, a carbonyl carbon from an alpha carbon. A bond or ring records which atoms belong to it.

For the ring-current calculation, `GeometryResult::Compute` first calls `Ring::ComputeGeometry` for each ring using the pose's atomic positions. It stores each ring's centre, normal and vertices in the conformation. `BiotSavartResult::Compute` uses this geometry to calculate the field from nearby rings at each atom. It stores the accumulated field and shielding contribution in that atom's conformation record. Ring membership stays fixed across poses; the geometry and calculated values can change.

The input readers and topology code establish chemical roles and connectivity before the calculators run. Calculators can then select atoms, bonds and rings from that description and concentrate on their numerical work. They use the result interfaces to declare prerequisites and write output. Adding a calculation need not mean inventing another naming scheme or atom order.

An atom's chemical record, pose data and accumulated trajectory measurements all use the same index. These separate stores let coordinates and results change with the frame without redefining the atom. The following sections explain the class and buffer relationships that make this possible.

Source: `GeometryResult.cpp::Compute`; `BiotSavartResult.cpp::Compute`; `LegacyAmberTopology.h::SemanticAt`; `ConformationResult.cpp::WriteAllFeatures`.

1.2 Reading the class diagrams

- A filled diamond sits beside the owner; the other end of the association is the owned object or collection.
- A hollow triangle points to a base class; italic class names are abstract.
- A plain arrow labelled **borrows** is a reference, not ownership.
- A dashed arrow indicates use or a call, not inheritance.
- Underlined operations are static. Italic operations are abstract.
- Notes explain the design; they are not extra C++ objects.

1.3 Identity and a pose

`Protein` records the atoms, their residues and their topology, independently of their coordinates. `ProteinConformation` holds the positions and calculated results for one pose. It refers to the protein without owning it, and keeps its own per-atom fields, geometry and result objects.

`Protein` also owns a list of retained conformations, omitted from this diagram. Temporary conformations used to process trajectory frames need not be added to that list.

Source: `Protein.h:37`, its owned storage near line 221; `ProteinConformation.h:28`; `ProteinConformation.cpp:7`; `Atom.h:14`.

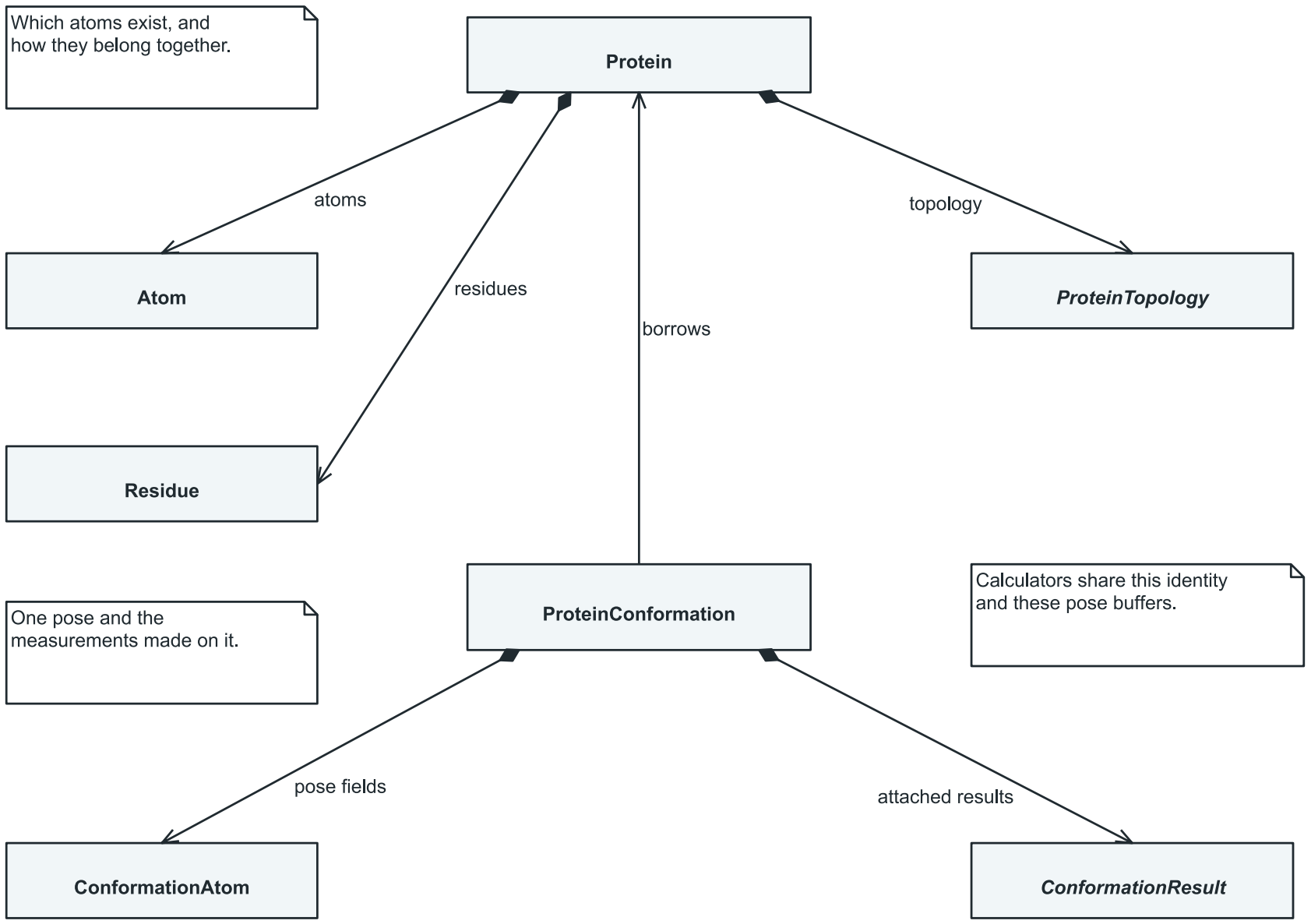


Figure 1.1: Protein identity and a conformation

1.4 Names, roles and grouping

The input code interprets atom names when it constructs the protein. `NamingRegistry` records the naming conventions used by different source tools for each residue type. `NamingApplicator` interprets a supplied atom name using the source convention and the atom's chemical context. `FullSystemReader` is one of the readers that uses these classes.

This interpretation lets calculators use chemical roles and connectivity without decoding each source tool's atom names. `LegacyAmberTopology`, the concrete implementation of `ProteinTopology`, provides access to bonds, rings and chemical categories. Residues, bonds and rings store indices into the protein's atom collection. Names remain available for identification and output.

Source: `NamingRegistry.h:274` and `:378`; `FullSystemReader.cpp:883` and `:959`; `Protein.cpp:440` onward; `ProteinTopology.h:19`; `LegacyAmberTopology.h:93`; `SemanticEnums.h:837`.

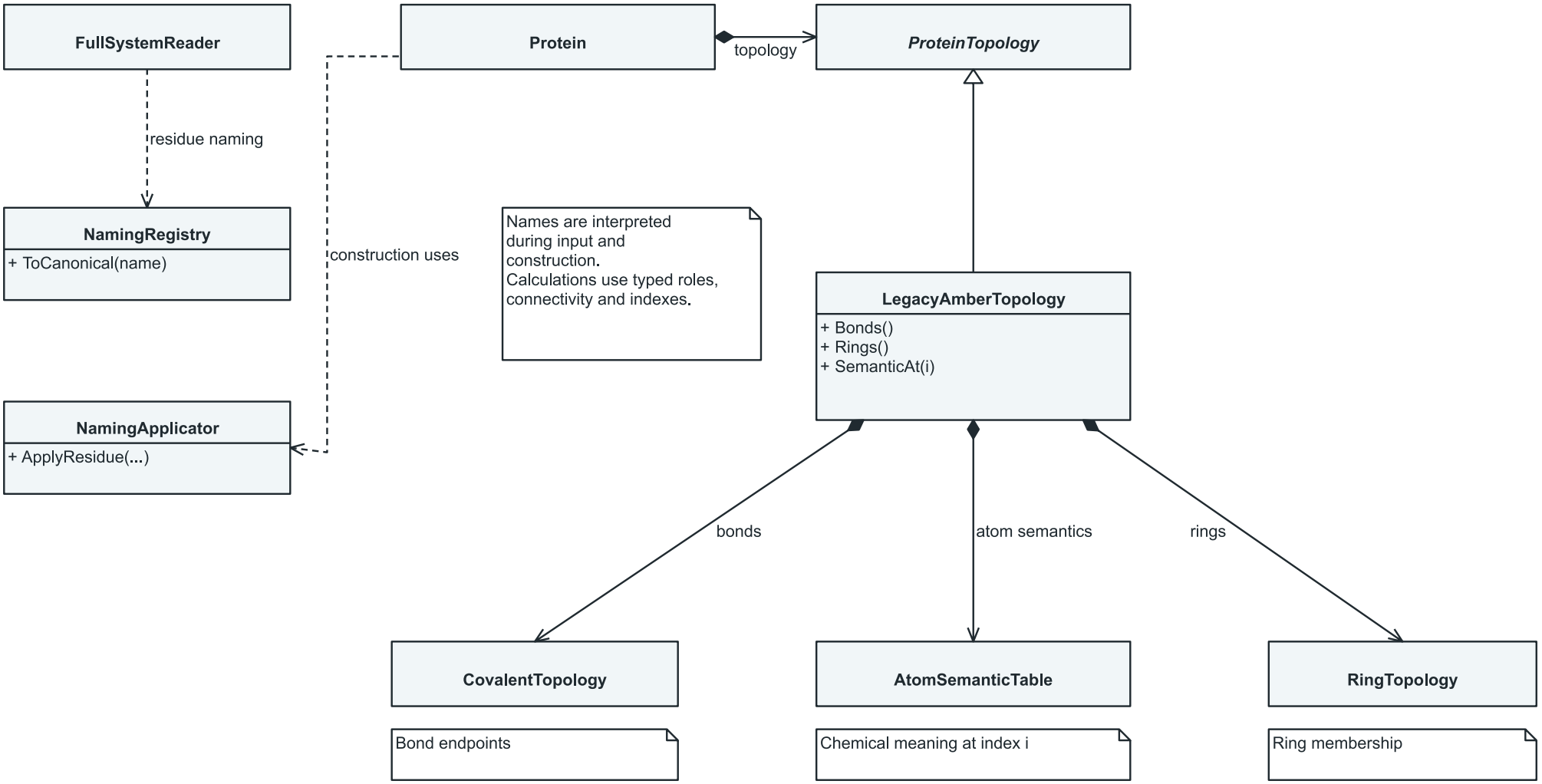


Figure 1.2: Naming and typed topology

1.5 The same atom across buffers

For the atom at index `i`, `Protein::AtomAt(i)` retrieves its chemical record, `ProteinConformation::AtomAt(i)` retrieves its pose data, and `TrajectoryProtein::AtomAt(i)` retrieves its accumulated measurements. These are separate records for the same atom. A pose calculator writes the current result; a trajectory calculator can use it to update statistics or append a time-series sample.

The records occupy separate arrays. In a dense time-series buffer, one index selects the atom and a second selects the stored sample. The sample's position in the buffer need not equal its original frame number. Original frame numbers and times are recorded separately.

Source: `Protein::AddAtom` and `AtomAt`; `ProteinConformation` construction and `AtomAt`; `TrajectoryProtein::InitTrajectoryAtoms` and `AtomAt`; `DenseBuffer.h:36`; `BsShieldingTimeSeriesTrajectoryResult.cpp:40` onward.

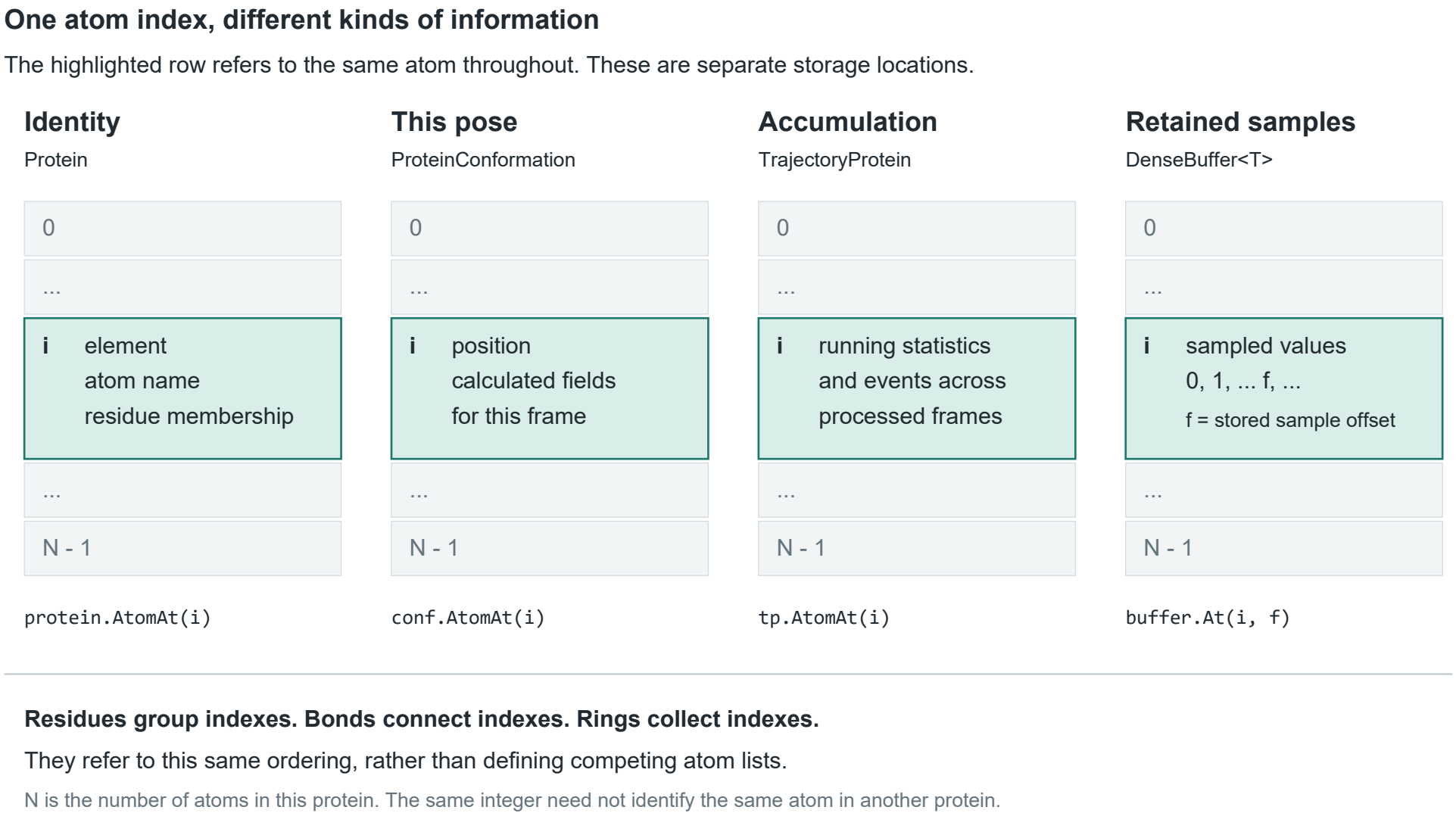


Figure 1.3: The same index identifies an atom in the identity records, pose data, accumulated measurements and stored samples.

1.6 Pose calculators and their results

The diagram shows result classes for geometry, ring-current and MOPAC calculations. Their static `Compute` functions receive a conformation, perform a calculation and return a result object. Each function can read the conformation's coordinates and refer to the associated `Protein` for chemical identity and topology.

`ConformationResult` provides the common interface for naming results, declaring their prerequisites and writing their output. It has no virtual `Compute` function: `OperationRunner` calls the concrete classes' static functions in an explicit order and attaches the results to the conformation. Before accepting a result, `ProteinConformation::AttachResult` checks that its prerequisite results are already attached.

Some result objects own their numerical arrays. Others read or write the conformation's named per-atom or geometry fields, providing access to those fields without keeping a duplicate buffer.

Source: `ConformationResult.h:40`; `GeometryResult.h:12`; `BiotSavartResult.h:70`; `MopacResult.h:33`; `OperationRunner.cpp:54` and `:68`; `ProteinConformation.cpp:21`.

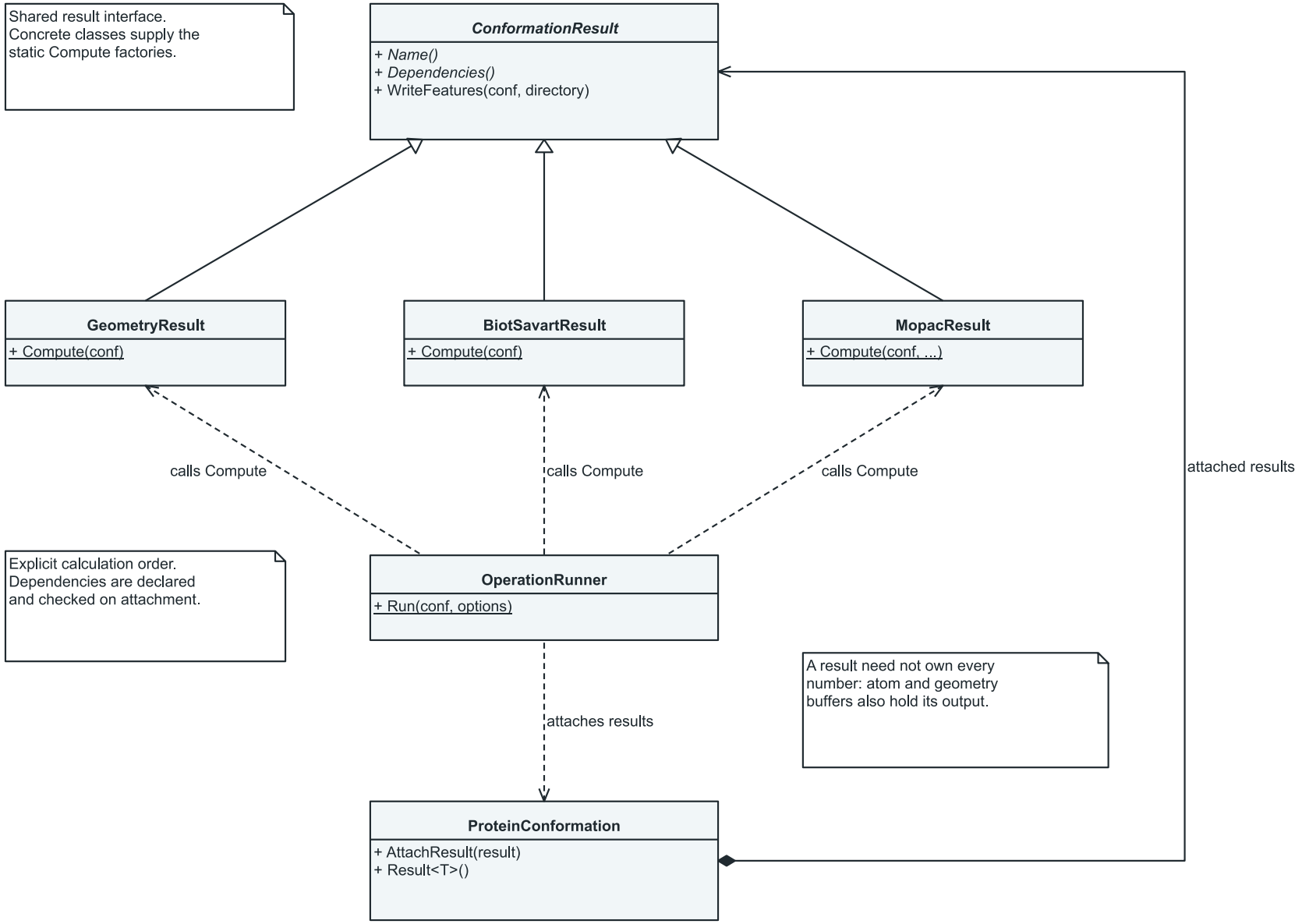


Figure 1.4: Pose calculator hierarchy and execution

1.7 Calculations across frames

`TrajectoryResult` is a separate abstract base for calculations that process successive frames. Unlike `ConformationResult`, it declares a virtual `Compute` operation. For each frame, `TrajectoryProtein` calls `Compute` on every attached trajectory result. After the last frame, it calls `Finalize` on those results.

`BsWelfordTrajectoryResult` updates statistical moments for each atom. `BsShieldingTimeSeriesTrajectoryResult` instead collects individual samples and packs them into a dense buffer. Both read the ring-current values calculated for the current conformation.

The diagram omits naming, dependency and HDF5-writing operations to show the calls made during frame processing and finalisation.

Source: `TrajectoryResult.h:27`; `TrajectoryProtein.cpp:238` and `:250`; `BsWelfordTrajectoryResult.cpp:55`; `BsShieldingTimeSeriesTrajectoryResult.cpp:40`.

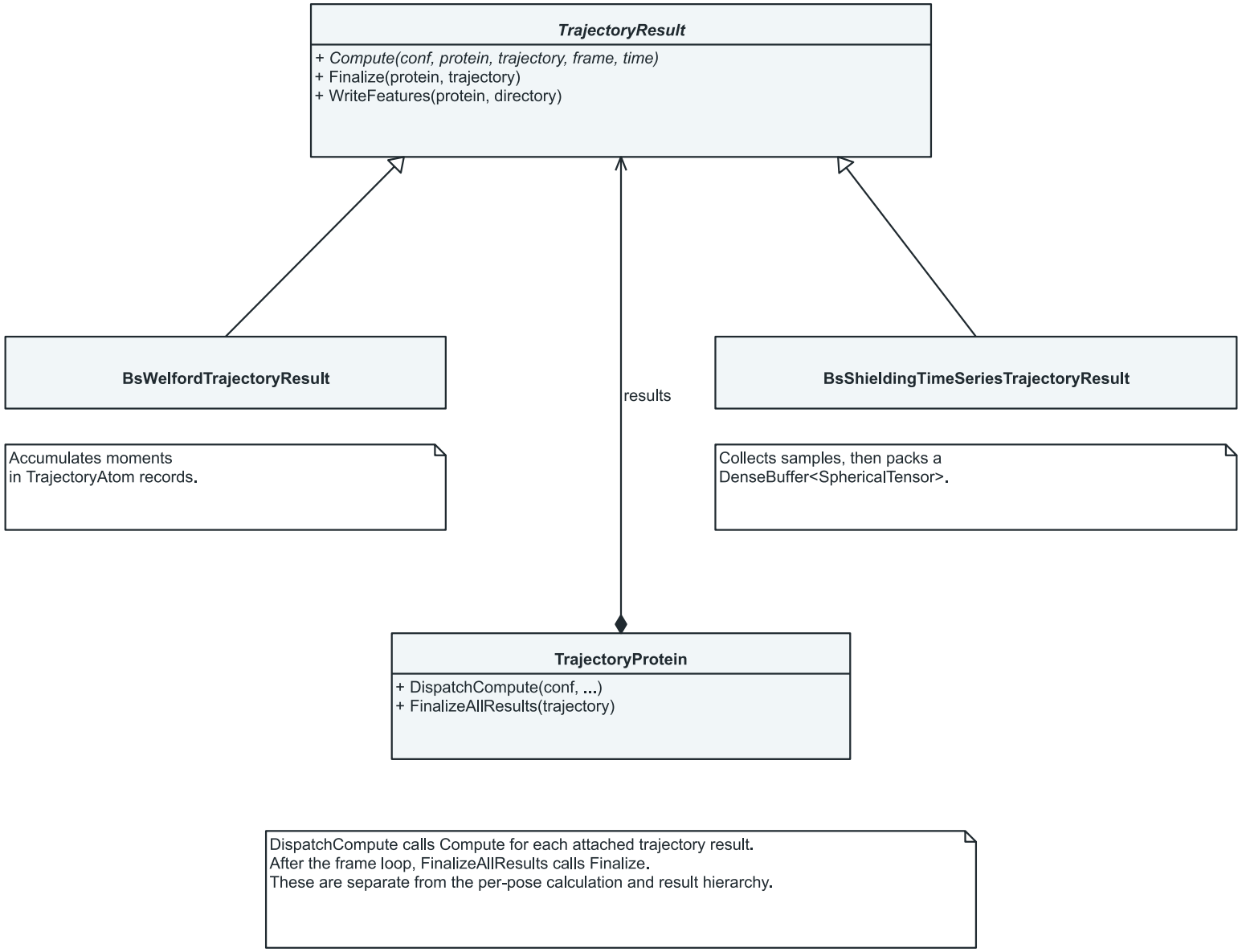


Figure 1.5: The separate trajectory calculator hierarchy

1.8 What lives for the whole trajectory

`Trajectory` controls frame processing. `TrajectoryProtein` owns the protein, the measurements accumulated for each atom, the attached trajectory results and the dense buffers retained after calculation.

The buffer classes inherit from `DenseBufferBase`, so `TrajectoryProtein` can own buffers of different numerical types in one collection. `DenseBuffer<T>` provides access to the values of type `T` in a particular buffer.

The ring-current time-series result collects samples in its own storage. During finalisation, it packs them into a dense buffer and transfers that buffer to `TrajectoryProtein`.

Source: `TrajectoryProtein.h:27`; `TrajectoryProtein.cpp:183`; `DenseBuffer.h:26`; `BsShieldingTimeSeriesTrajectoryResult::Finalize`.

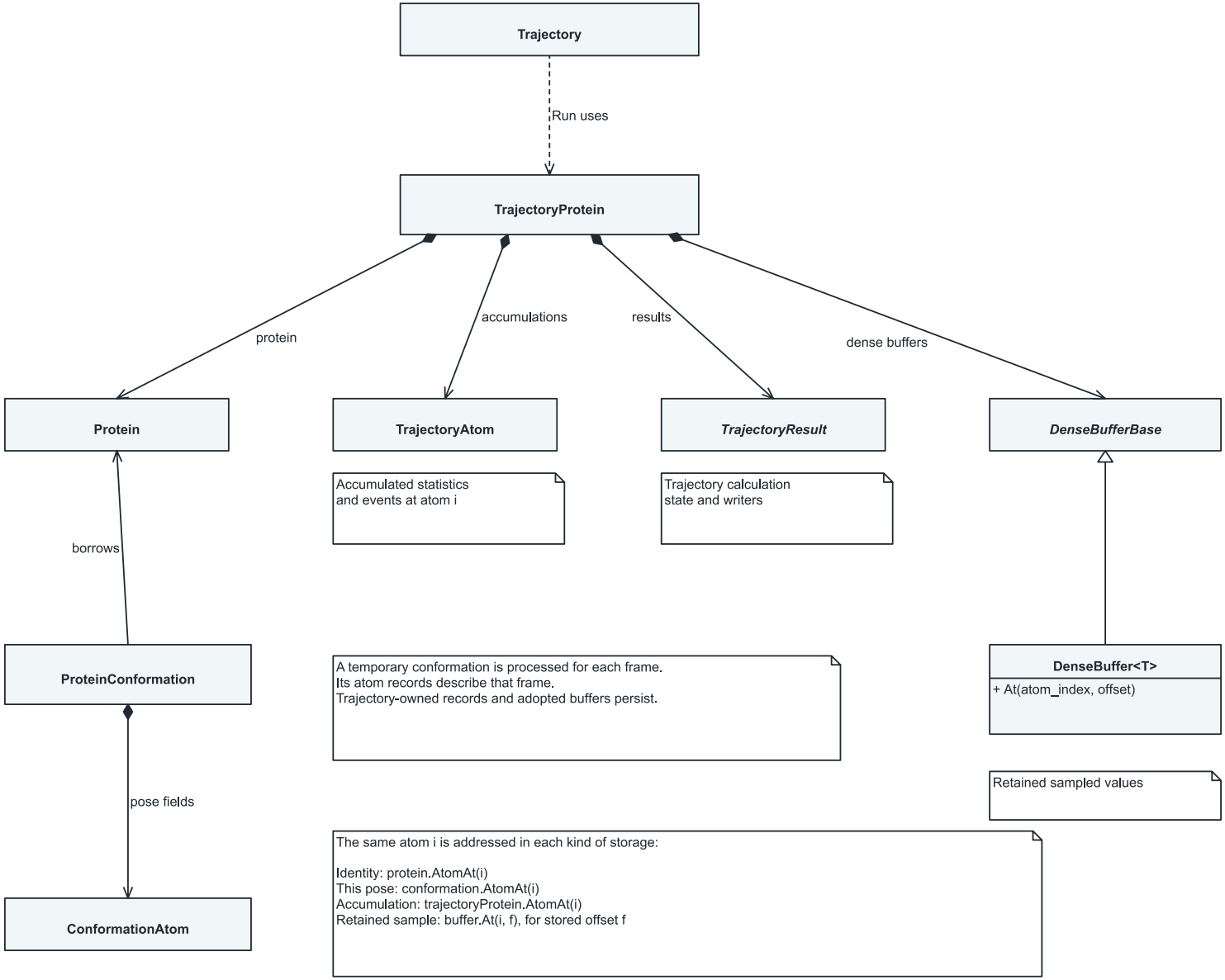


Figure 1.6: Trajectory ownership and storage

1.9 One frame through the system

The initial conformation is retained as a canonical pose. For each subsequent frame, the loop creates a temporary conformation from the new coordinates. `OperationRunner` runs the pose calculators and attaches their results to that conformation. Trajectory calculators read the pose results and update the measurements they retain across frames. The loop then writes the configured frame output and can release the temporary conformation.

The diagram starts after the initial conformation has been processed; PDB output and environment preparation are omitted. The pose-calculator box groups the concrete `Compute` functions. The NPY writer calls `ConformationResult::WriteAllFeatures` to write the common arrays and invoke each attached result's output function.

Source: `Trajectory.cpp:353` through the frame loop and `:405`; `OperationRunner.cpp:68`; `FrameNpyEmitter.cpp:68`; `ConformationResult.cpp:50` and its attached-result writer loop.

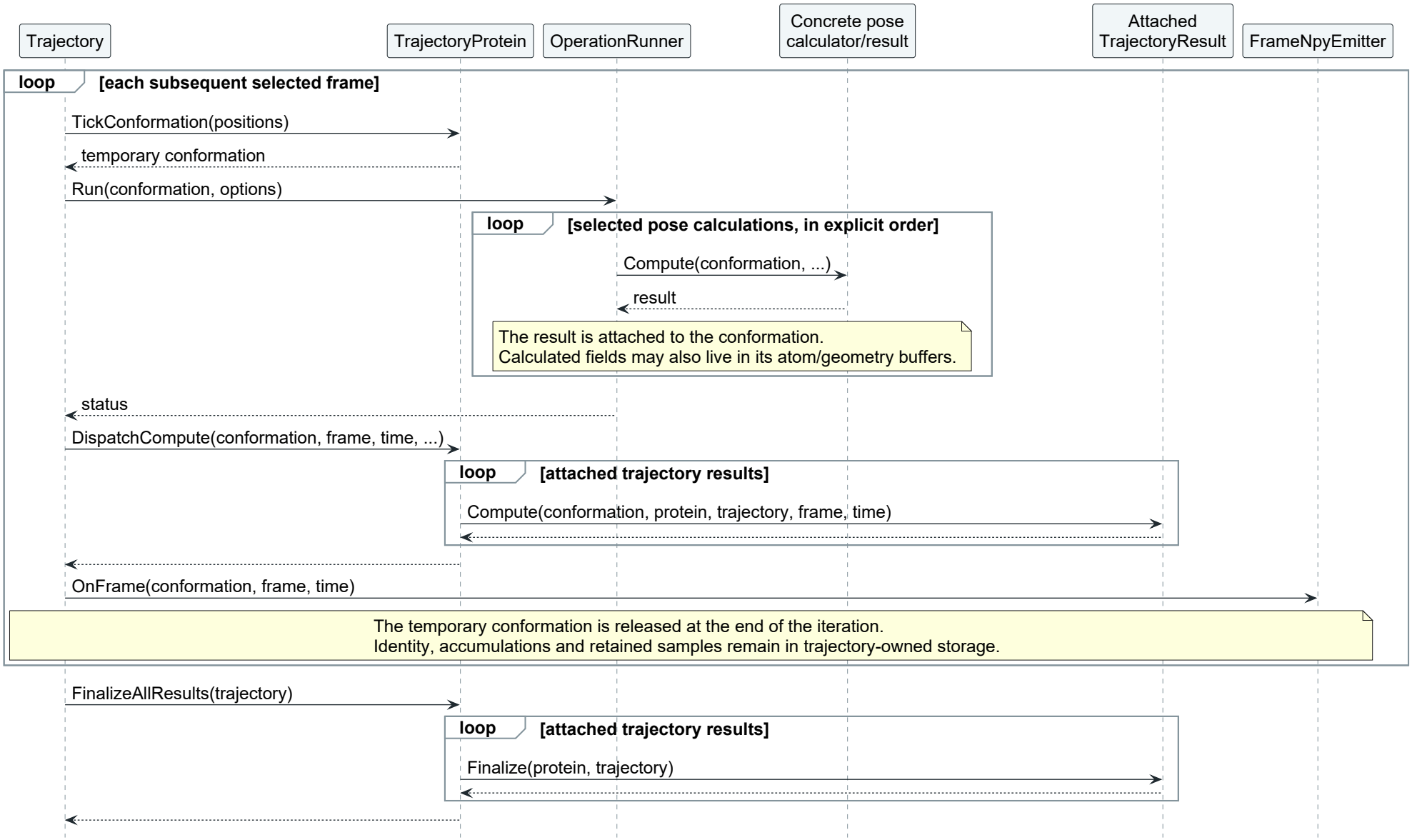


Figure 1.7: The main trajectory frame loop

1.10 Main objects

Object	What it does
`Protein`	Owns atom and residue records and the protein's topology.
`ProteinConformation`	Stores coordinates and calculated values for one pose and refers to its `Protein`.
`Atom`	Records an atom's chemical identity and its place in the protein.
`ConformationAtom`	Holds the position and named per-atom fields used for one pose.
`LegacyAmberTopology`	Supplies bonds, rings and typed chemical descriptions to the calculators.
`ConformationResult`	Declares the functions through which pose results report prerequisites and write output.
`OperationRunner`	Calls the selected pose calculations in an explicit order and attaches their results.
`Trajectory`	Reads and processes the selected frames in sequence.
`TrajectoryProtein`	Owns the protein and the calculation state retained across frames.
`TrajectoryAtom`	Holds accumulated measurements and events for the same atom over time.
`TrajectoryResult`	Supplies the per-frame and finalisation interface for a trajectory calculation.
`DenseBuffer<T>`	Retains typed values in flat storage addressed by atom and sample offset.

Chapter 2

Prediction models

Three models use the extracted measurements: one predicts shielding tensors, one predicts shifts from static structures, and one predicts shifts from trajectory conformations. When training a shift model, the fitting code can initialise its encoders with weights learned by the shielding model.

This chapter describes the calculations and the transfer of weights. Appendix A follows the implementation from its data records through a complete prediction call.

2.1 The three predictions

Shielding. An equivariant encoder learns a representation of each atom's environment from chemical categories, geometry and selected measurements. Two readouts convert these features into contributions to the atom's symmetric shielding tensor. Their sum gives six predicted components, which the prediction code converts from normalised values back to shielding units.

Static shifts. The shift model starts with copies of the shielding encoder and is fitted to experimental shifts. Its readouts use features from the encoders, local descriptors and OpenFold residue embeddings to predict shifts for particular atom roles. The model adds rotamer corrections to those predictions.

Trajectory shifts. A separately fitted shift model uses the static model's architecture without the OpenFold embedding. The evaluation code calls this model once for each of 100 conformations, then takes the equally weighted mean of each atom's 100 predicted shifts.

The published MD fitting configuration uses one selected conformation per protein. The 100-frame average is calculated during evaluation, not during training. Each prediction uses one frame's coordinates; the model does not receive averaged coordinates. See `md/config.toml` and `md/evaluate.py::predict_trajectory`.

Solid arrows in the overview indicate calculation flow. Dashed arrows indicate parameter copying before fitting, not a shielding-model call during prediction. "Decode to ppm" labels the conversion from normalised predictions to shifts (Section 2.2).

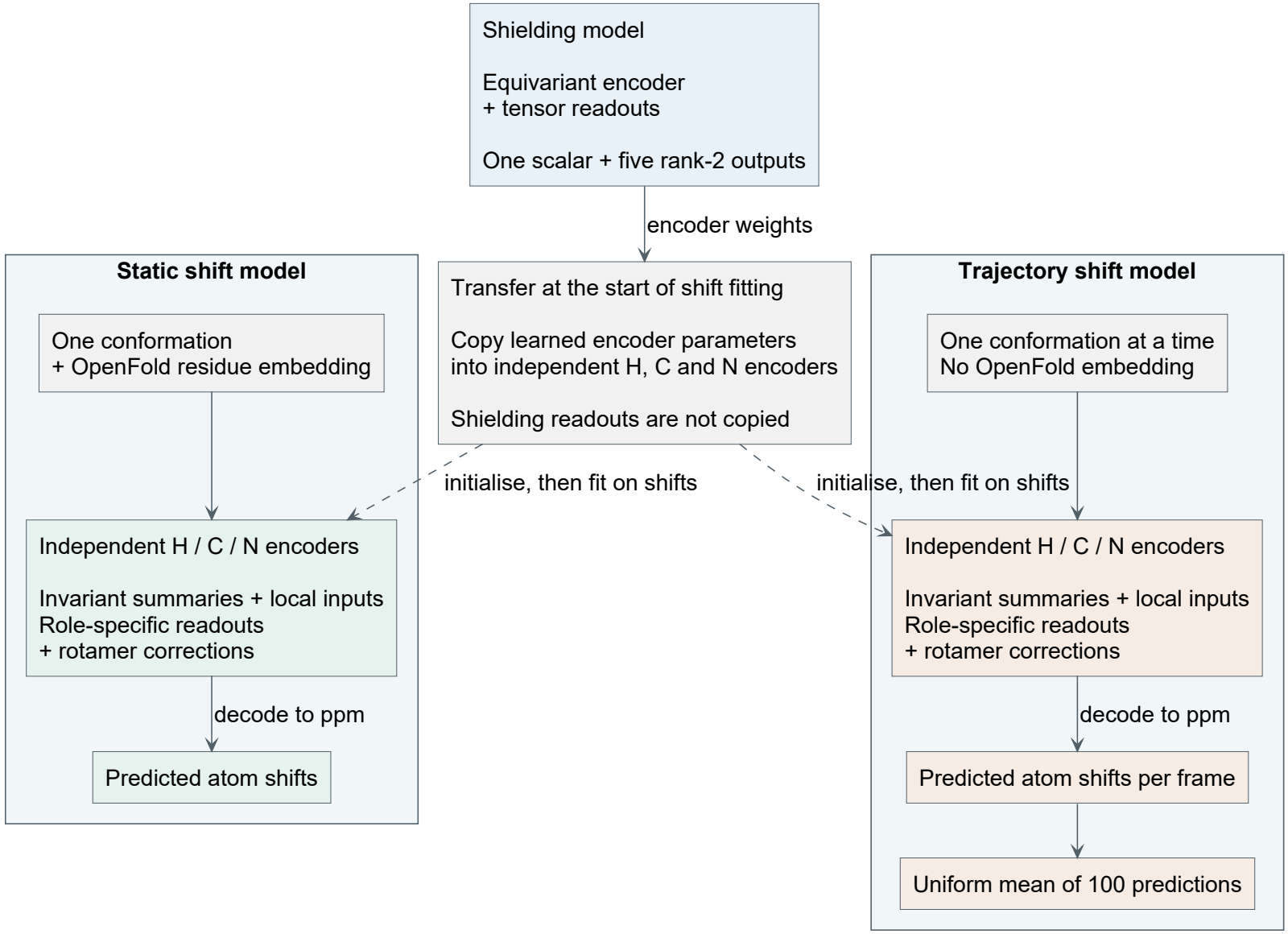


Figure 2.1: Transfer and prediction

2.2 Preparation and units

Before passing per-atom measurements to a network, the preparation code subtracts a fitted centre from each scalar channel and divides by a fitted scale. For a vector or rank-2 channel, it divides all components by one scale and does not subtract a centre. Scaling the components together preserves their rotation rules.

A saved vocabulary assigns an integer identifier to each chemical category. The encoder uses that identifier to select a row of its learned embedding table.

The shift models keep the shielding model's input normalisation and extend its vocabularies with categories found in the shift-training inputs. The fitting code calculates separate normalisation values for the additional rotamer descriptors and the shift targets. The fitted model includes these vocabularies and normalisation values for use during prediction.

The networks are trained to predict normalised targets. To recover shielding values, `ShieldingTargetNormalizer` selects the fitted centre and scales for the atom's category. It rescales the isotropic part and restores its centre, then rescales the five rank-2 components together without adding a centre. For shifts, `absolute_prediction` multiplies the prediction by the scale fitted for the atom's output role and adds the corresponding centre to obtain a shift in ppm. These conversions are separate from the neural readouts.

Source: `shielding/preparation.py::_normalize_input`, `ShieldingTargetNormalizer`; `shift_preparation.py::prepare_shift_input`; `shift_targets.py::absolute_prediction`.

2.3 Encoder architecture

`ModelBatch` contains the input arrays for one complete protein. Row `i` refers to the same atom in the position, label and measurement arrays. Bond endpoints use these row numbers; residue indices identify the atoms' residues.

`EquivariantEncoder` maps each categorical label to a learned list of numbers, called an embedding. Its 17 embedding tables cover element, atom name, residue type, atom role, hybridisation and prepared chemical state. The encoder combines these embeddings with the input measurements and applies three message-passing layers over a neighbour graph.

The graph includes up to 64 nearest neighbours within 8 angstroms of each atom, plus edges for covalent bonds and selected electronic pairs. An atom can therefore receive messages from more than 64 neighbours. Edge directions are represented by spherical harmonics of ranks 0, 1 and 2, and distances by 10 radial basis functions.

At each layer, `MessageConvolution` calculates a message from each neighbouring atom to the receiving atom. The tensor product combines the neighbour's features with spherical harmonics of the direction from neighbour to receiver. A small radial network calculates the mixing weights from distance and connectivity descriptors. The layer averages the incoming messages and adds that average to a learned transformation of the receiving atom's own features. An `e3nn.nn.Gate` then applies nonlinear functions to scalar features and uses scalar values to scale each vector or rank-2 feature as a whole.

After each layer the hidden representation has 12 scalar channels, 12 vector channels and 12 rank-2 channels: `12+12*3+12*5=108` numeric components per atom. This implementation is equivariant under the rotations in SO(3). The `e` labels used in the code do not establish equivariance under physical reflections of all the inputs.

Source: `equivariant.py::ModelBatch`, `EquivariantEncoder`, `MessageConvolution`; `atom_labels.py::LABEL_EMBEDDING_DIMS`.

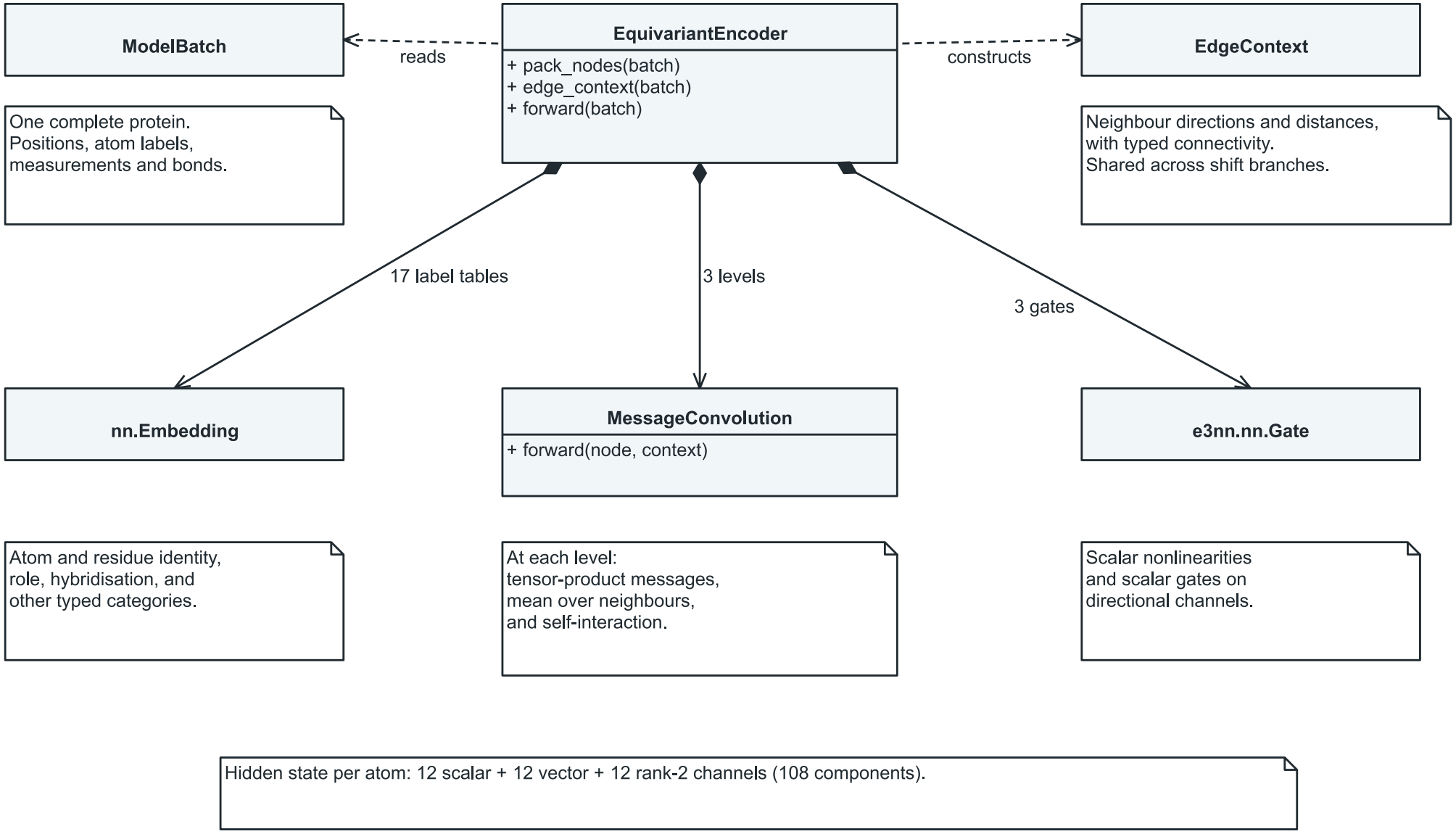


Figure 2.2: The encoder class used by the shielding and shift models. Each encoder instance has its own parameters.

2.4 Shielding readout

`ShieldingModel` passes the encoder's features to two readouts. The first, `e3nn.o3.Linear`, uses the same learned weights for every atom. The second, `ResidueAtomCorrection`, uses weights specific to the residue:atom category. The model sums their outputs: one scalar component for isotropic shielding and five components for the symmetric-traceless part. The model does not predict the three antisymmetric components of a full tensor.

The diagram shows `ShieldingTargetNormalizer` separately because it is not a child module of the neural model. Its conversion to shielding units is described in Section 2.2.

When the selected inputs include the MOPAC peptide rank-2 contribution, the model sets that input channel to zero before calling the encoder. It multiplies the original peptide contribution by a learned coefficient for the atom name and adds the result to the five rank-2 output components. This term leaves the scalar output unchanged. The structure-only model has no such term, and the transfer code does not copy these coefficients to the shift models.

Source: `shielding/model.py::ShieldingModel`, `ResidueAtomCorrection`; `shielding/preparation.py::ShieldingTargetNormalizer`.

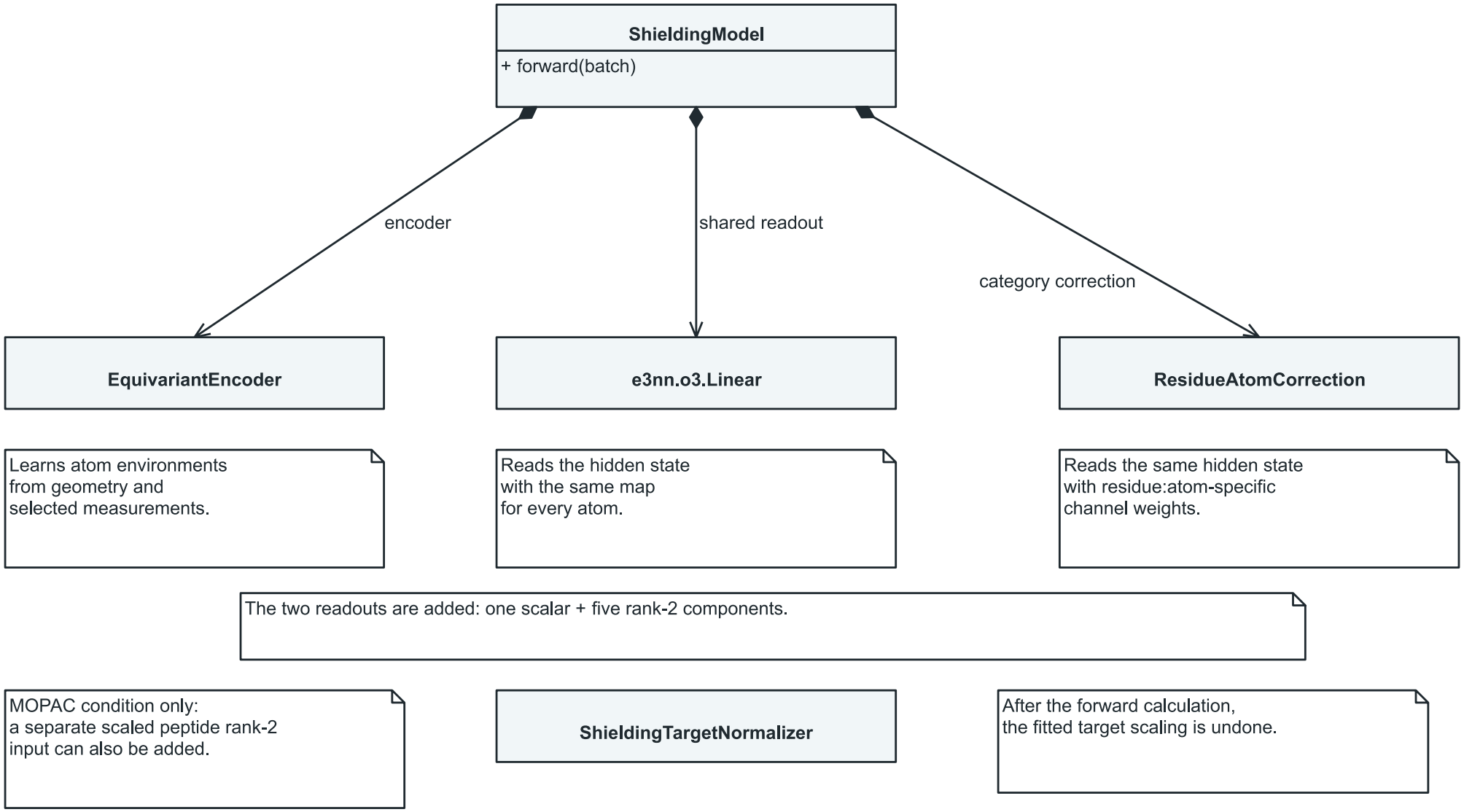


Figure 2.3: Shielding model

2.5 Shift readout and chemical roles

`ShiftPredictor` combines two calculations for each of H, C and N: a prediction from `IndependentEncoderBranches` and an added dihedral-based correction from `NucleusRotamerResidual`.

`IndependentEncoderBranches` contains a separate encoder and readouts for each of H, C and N. Each encoder calculates features for the whole molecular graph; its readouts predict shifts only for its own element. The three branches have separate learned parameters but use the same `EdgeContext`, which stores edge geometry and descriptors rather than learned values.

Within each branch, chemical labels determine additional scale factors for features after message layers two and three. These factors apply to atoms of the branch's element. To prepare the readout's inputs, the branch keeps the encoder's twelve final scalar features and takes the norm of each of its twelve vector and twelve rank-2 features. The resulting 36 values per atom do not change when the molecule and its directional measurements rotate together.

A small network supplies a further 24 values from the atom's original invariant inputs. These are concatenated with the 36 encoder-derived values for the readout. This "local bypass" lets the readout use local information that has not passed through the message layers; each output role has its own bypass. In the static model, each H/C/N branch also reduces the 384-value OpenFold residue embedding to 32 values and appends them to the readout's input, using the same values for all atoms in a residue.

The nine output roles are:

Nucleus	Readout roles
H	Amide, alpha, other
C	Carbonyl, alpha, beta, other
N	Backbone, sidechain

Each role has a network that predicts a shift from the concatenated features. A second readout uses chemical labels to weight the same network's hidden values, and its output is added to the first prediction.

The rotamer modules calculate three further corrections from dihedral descriptors. One uses the descriptors of individual sidechain angles; another uses products of descriptors for selected pairs of angles. The third also uses the encoder's invariant features as context. Support flags determine which corrections are added to an atom's encoder-branch prediction.

The prediction code then converts the summed, normalised output to ppm as described in Section 2.2.

Source: `shift_model.py::IndependentEncoderBranches`, `PreparedChemistryGate`, `NucleusRotamerResidual`, `ShiftPredictor`; `shift_roles.py::NUCLEUS_ROLES`; `shift_targets.py::absolute_prediction`.

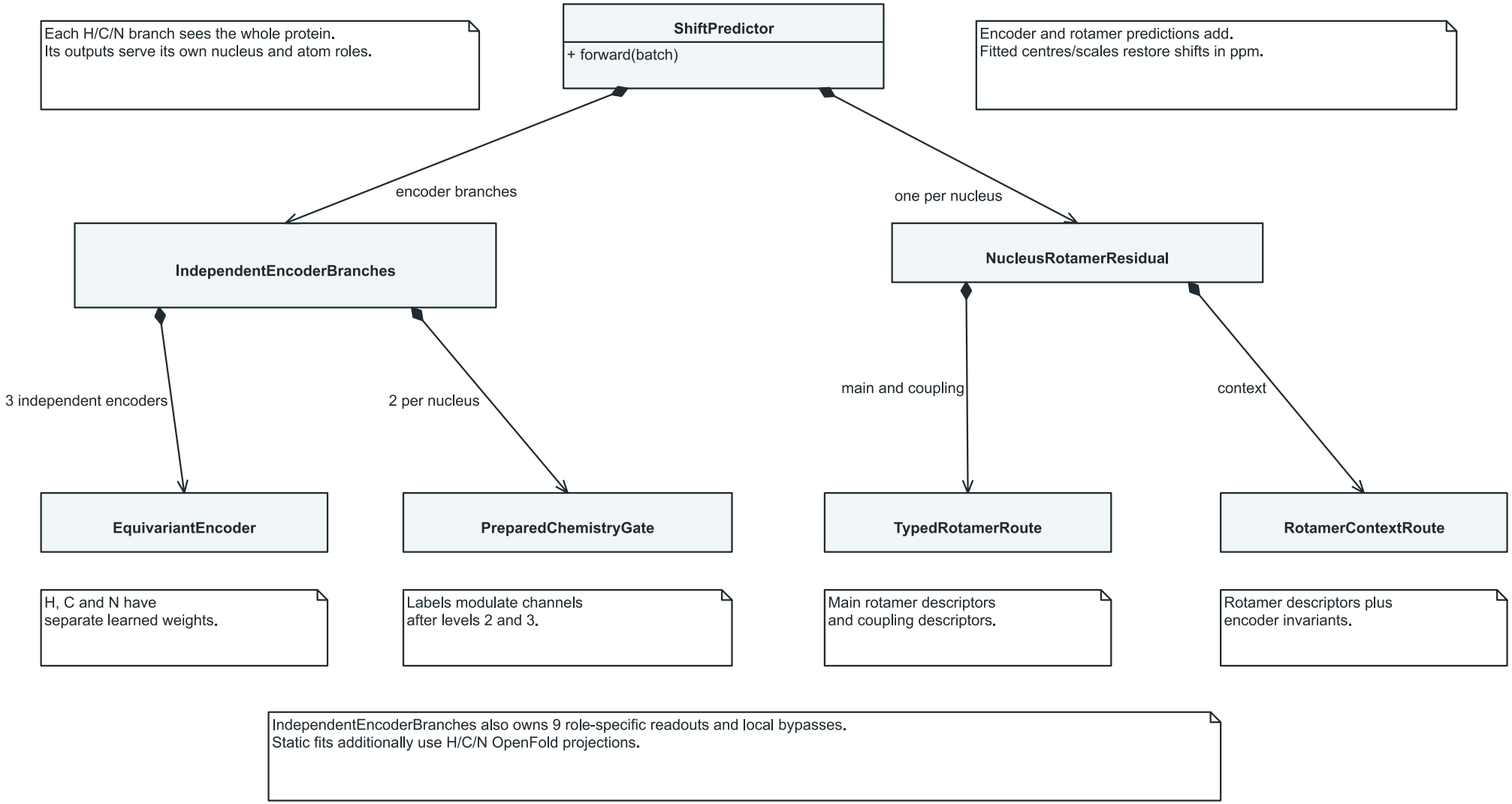


Figure 2.4: Shift model

2.6 Transferring weights from shielding to shifts

`shift_transfer.py::transfer_encoder_parameters_` copies the learned parameters from the shielding encoder's three message convolutions into each H/C/N encoder. The function also copies rows from the shielding encoder's 17 embedding tables for categories encountered during shielding training. Each shift encoder gets a separate copy. Rows for new categories keep their initial random values. The ordinary e3nn gates have no fitted parameters to copy.

The function does not copy parameters from the shielding readouts, residue:atom correction or optional peptide output term. Training on shift data fits the parameters of the shift model's chemistry gates, local bypasses, OpenFold projections, role readouts and rotamer modules.

The published comparison also trains an otherwise matched shift model without copying encoder weights. The diagrams show the version that uses transferred weights.

Source: `shift_transfer.py::transfer_encoder_parameters_`; `shift_fit.py::_fit_initialization_arms`; `shift_parent.py`.

Chapter 3

Reader

Reader builds a molecular model from the extractor's files and displays structures and measurements through Qt and VTK. It stores an atom's chemical identity separately from its coordinates and calculated values. The display and analysis tools use atom indices to connect these records.

This chapter describes the published Reader at revision `8714b8ca` (10 September 2026). Appendix C identifies the source files behind the diagrams.

3.1 The same atom throughout

`QtProtein` holds the atoms, residues, names and topology for one loaded protein. Each `QtAtom` records chemical categories including element, locant, branch, backbone role and stereochemistry. It also stores the atom's index and residue index. A parallel `QtAtomNames` array holds the names used for display labels.

`QtResidue` groups atom indices and retains direct indices for common backbone atoms. The loader builds these lists from the extractor's topology records. `QtTopology` stores bonds and rings, plus lists of those containing each atom. Bond endpoints, residue members and selected atoms all use indices into the same atom array.

For atom index `i`, these expressions refer to the same atom:

```
protein.atom(i);                          // chemical identity
protein.atomNames(i);                     // names in supported conventions
conformation.atomPosition(frame, i);      // coordinates in a chosen frame
biotSavart.shielding(i);                  // a result in a selected snapshot
```

These calls use the index directly rather than searching by atom name. Reader retains names for reading files, labelling the display and writing output.

Code: `QtProtein.h`, `QtAtom.h`, `QtResidue.h`, `QtTopology.h`, `QtTopologySidecar.cpp::PopulateBackboneCache`, and the membership construction in `QtTopologySidecar::Load`.

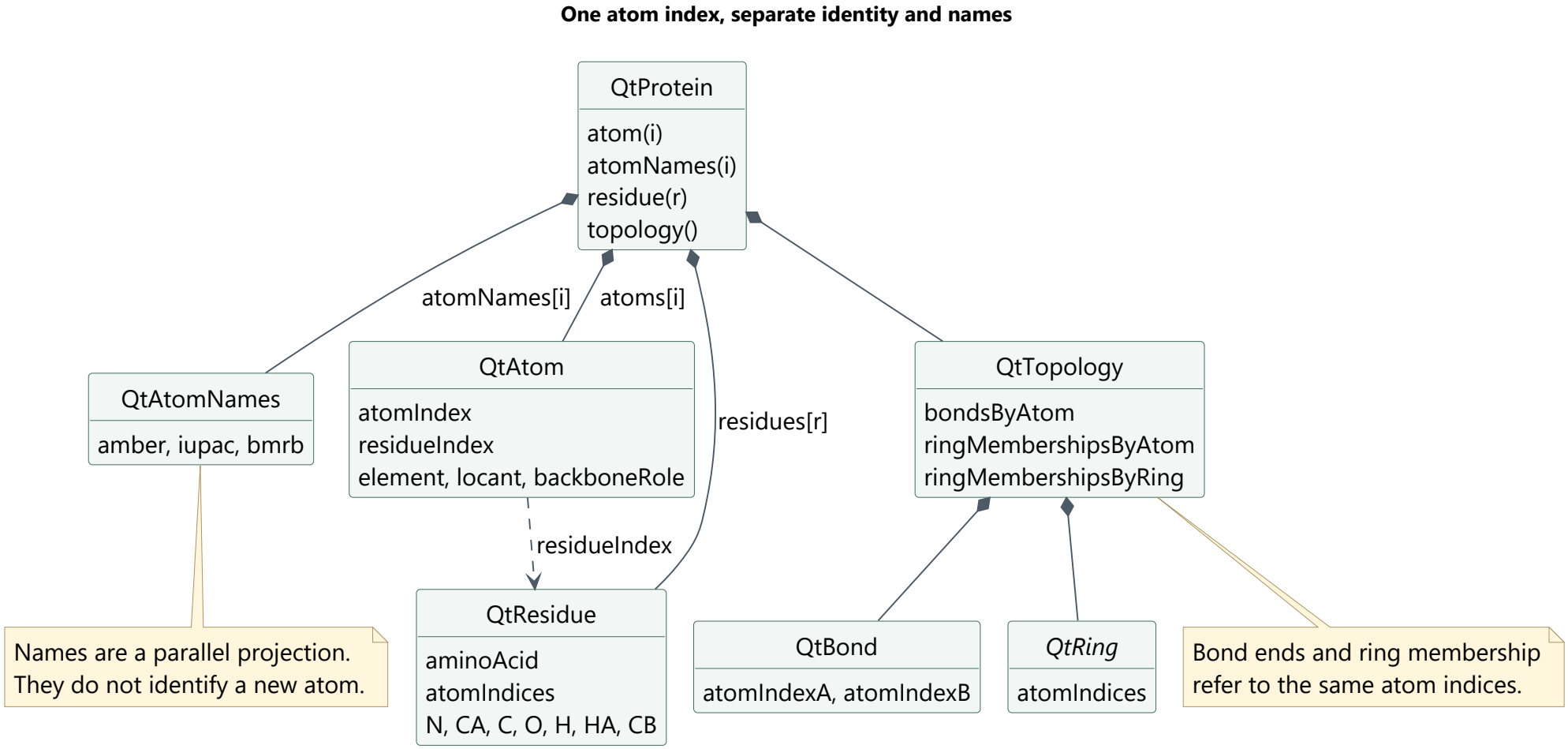


Figure 3.1: Typed identity, names and topology

3.2 Molecular identity and frame data

`Conformation` provides a common interface to frame counts, times, positions and detailed results. `SingleConformation` holds one pose snapshot. `TrajectoryConformation` owns `QtTrajectoryH5`, which keeps arrays of positions and available time-series measurements in memory. During playback, the scene reads those arrays through the conformation without reloading the per-frame NPY files at every animation step.

`QtConformationSnapshot` stores the detailed results for one frame. It holds an array of `NpyColumn` objects indexed by `FieldKind`. `FrameNpyLoader` maps filenames to field identifiers, loads the fields selected by Reader's policy and fills the columns. Each column records its presence, shape and numerical values; the catalogue specifies whether its rows refer to atoms, residues or rings. Loaded columns are accessed by `FieldKind`, without searching filenames.

Typed accessors such as `QtBiotSavartGroup` read vectors, tensors and small result records from the snapshot's columns. They refer to the snapshot without copying its arrays or rerunning the extractor's calculations. For example, `QtRingKernelGroup` reads an atom's nine stored values as one scalar, three rank-1 components and five rank-2 components. Its accessor returns an empty optional value if the required column is absent.

The base `Conformation` retains one current snapshot. `requestSnapshot` loads a requested snapshot synchronously on the object's thread, then emits `snapshotReady`. The dashboard keeps its own samples for history plots rather than relying on that single snapshot.

Positions and frame numbers use the following indexing conventions:

- The three coordinates of an atom in a stored frame begin at offset `(atom*frameCount+frame)*3` in the position array. A `QtFrame` holds a pointer to the conformation and the stored frame's row number. It retrieves values through the conformation instead of owning another set of arrays.
- A trajectory row and an original frame number are different identifiers. `TrajectoryFrameMap` maps between them. The loader uses the original number to find the corresponding `frame_NNNNNN` NPY directory and records that number in the snapshot.

Code: `Conformation.cpp`, `TrajectoryConformation.cpp`, `SingleConformation.cpp`, `QtTimeSeriesBuffers.h`, `FrameNpyLoader.cpp`, `QtConformationSnapshot.h`, and `QtRingKernelGroup.h`.

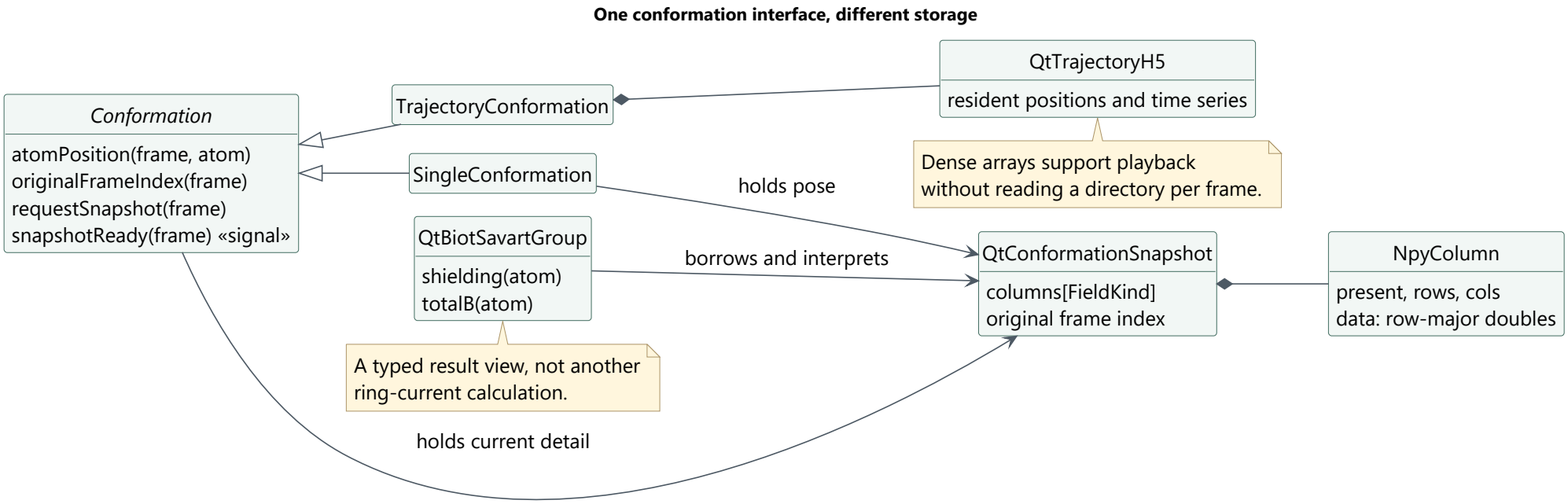


Figure 3.2: The conformation interface and its storage

3.3 The main window owns the loaded data

The extractor's `Protein` owns its retained conformations. Reader's loader instead returns a `QtLoadResult` that owns the `QtProtein` and `Conformation` separately. `ReaderMainWindow` owns that result and creates the display and control objects that use the loaded data. The conformation holds a non-owning pointer to `QtProtein`, through which it accesses chemical identity and topology.

`QtProtein` does not inherit from `QObject`. `Conformation` and the playback, selection and display controllers do. The diagrams distinguish ownership through `unique_ptr` or QObject parenting from borrowed pointers. Snapshots use `shared_ptr<constQtConformationSnapshot>`, which lets several readers retain the same snapshot without modifying it through those pointers.

Reader keeps its dock widgets when it replaces a loaded run. `clearLoadedRun` first disconnects the views from the old data. It then deletes the run's scene and controllers before releasing the transformed conformation and the loaded data. This order prevents views from referring to objects that have already been destroyed.

Code: `QtProteinLoader.h::QtLoadResult`, `ReaderMainWindow.cpp::installLoadedRun`, and `clearLoadedRun`.

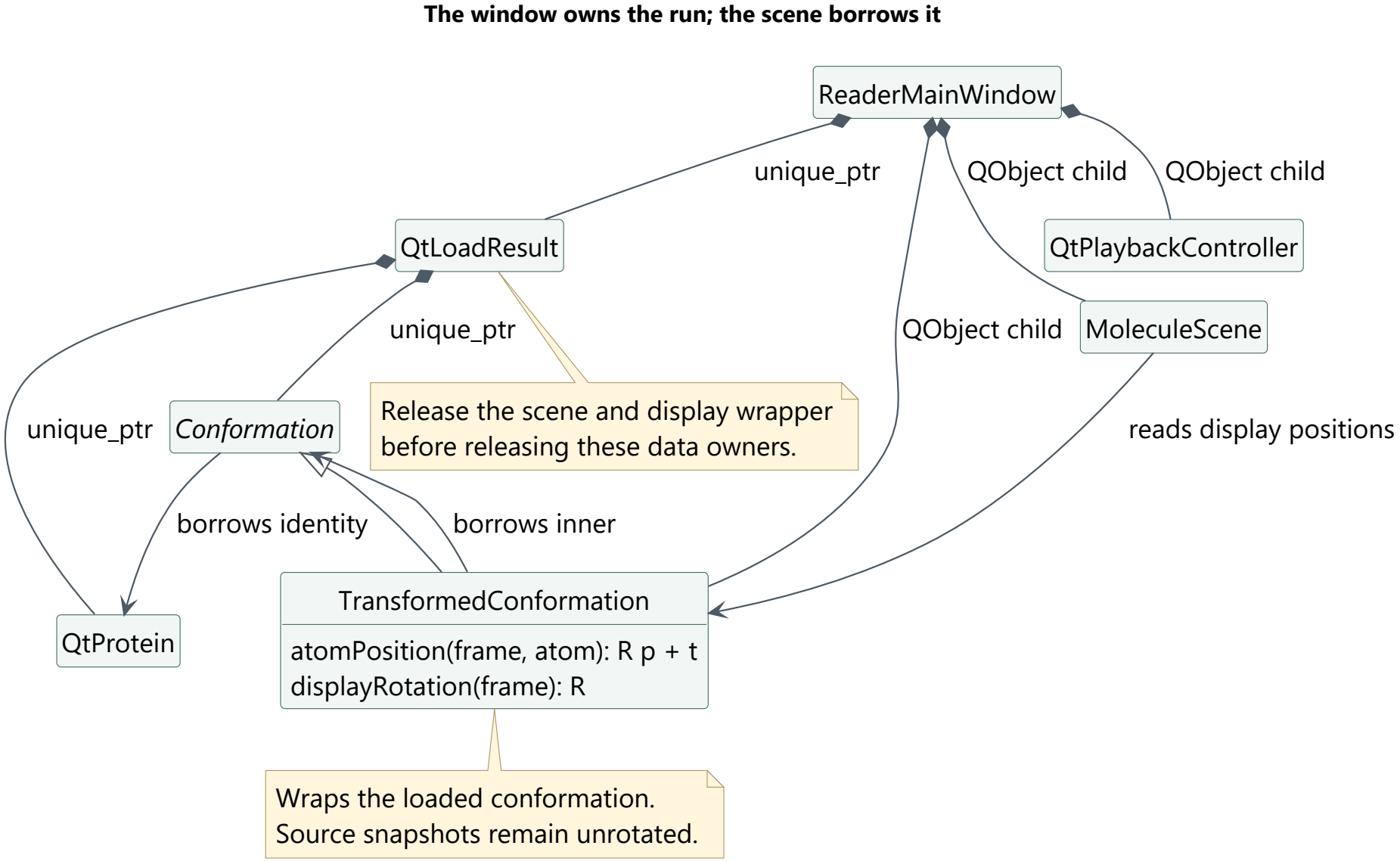


Figure 3.3: Loaded-run ownership and the display wrapper

3.4 Display coordinates

For display, `TransformedConformation` returns rotated and translated positions through the `Conformation` interface. It applies `R*position+translation` to positions while preserving atom indices and frame numbers. The wrapper returns the underlying snapshot with its stored vectors and tensors unchanged. Tensor display code obtains the rotation separately from the wrapper and applies it where required.

Code: `TransformedConformation.cpp::atomPosition`, `displayRotation`, and `loadSnapshot`.

3.5 Qt connects the views

`QtPlaybackController` stores the current frame number and emits `frameChanged` when it changes. The main window connects that signal to the scene, inspector, dashboard and time controls. The scene updates its geometry by reading positions through the conformation interface. If several render requests arrive before the next render, the scene queues only one Qt callback. The callback asks VTK to render after the current update has finished.

The scene and inspector share an `AtomSelection` object. This `QAbstractListModel` records the selected atom indices and the focused atom. To display a selected atom's measurements, the inspector requests a snapshot from the conformation. When the conformation emits `snapshotReady`, the inspector reads the atom's values through the typed result accessors.

The dashboard uses `TrajectorySignalCatalog` to list available measurements. Qt item models record the user's choice of measurements and panels. `DashboardDisplayController` and the panel widgets retain sampled measurements for the history plots.

The REST interface is another way to operate the application. The `/api/run/load` route calls the main window's `loadRunPath` function, and `/frame/set` requests a frame change through the playback controller. Both routes use the same functions as the interactive controls. Longer model-input exports use the loaded protein and conformation and proceed in steps queued through Qt's event loop.

Code: `ReaderMainWindow.cpp::installLoadedRun`, `QtPlaybackController.cpp`, `MoleculeScene.cpp::requestRender`, `QtAtomInspectorDock.cpp::setContext`, `AtomSelection.h`, and `RestServer.cpp`.

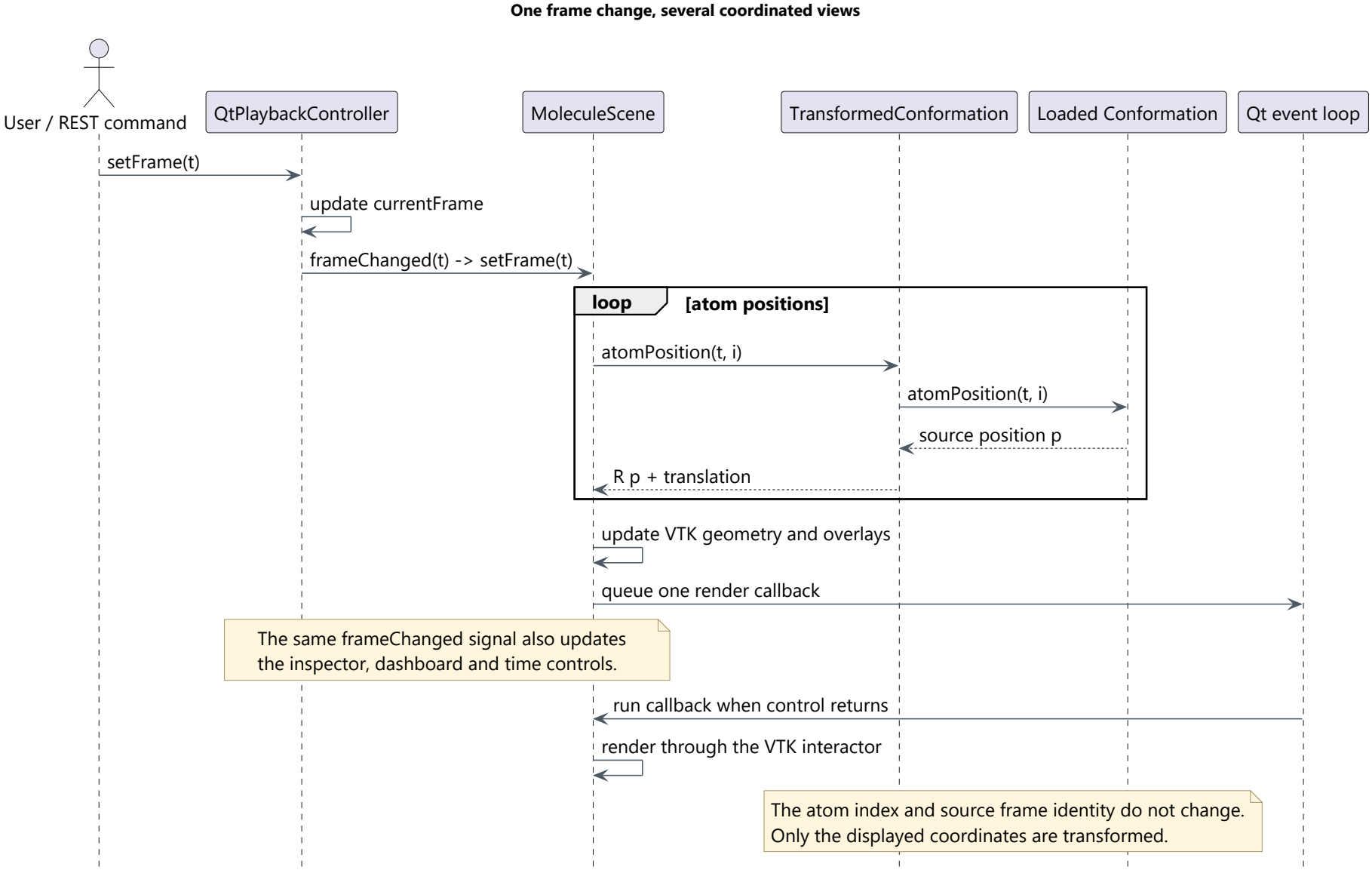


Figure 3.4: One frame change through the application

3.6 The correspondence with the extractor

Design idea	Extractor	Reader
Stable molecular identity	`Protein`, `Atom`, `Residue`	`QtProtein`, `QtAtom`, `QtResidue`
Find bonds and rings from an atom	Protein/covalent/ring topology objects	`QtTopology` with bonds, rings and reverse-index lists
Names are separate from chemical categories	Chemical categories and names in supported conventions	Typed atom fields and separate name arrays
Coordinates change; atom indices stay the same	`ProteinConformation`, `ConformationAtom`	`Conformation::atomPosition(frame, atom)`
Access calculated measurements	Calculator results and result buffers	Catalogued columns and typed group views
Trajectory storage is separate from identity	Trajectory protein/atom buffers	`QtTrajectoryH5` time series and `QtFrame` views

Reader shares these design principles with the extractor, but builds its own molecular objects from the extracted files. Its display, export functions, optional inference helper and visualisation calculations all use the loaded data. It does not run the extractor's calculator sequence or the Python training code.

Appendix A

From code to prediction

Chapter 2 describes what the three models do. This appendix follows their code: the input arrays, the calculation in one message layer, the construction of a model and a complete prediction call.

A.1 Input arrays and atom indices

The models store inputs in arrays rather than creating a Python object for every atom. `ModelBatch.positions[i]`, `scalars[i]`, `l1[i]`, `l2[i]` and each `labels[field][i]` refer to the same atom. Bond endpoints and graph endpoints use these row numbers too. One `ModelBatch` describes one conformation of a complete protein.

Three dataclasses organise the inputs: `ModelWidths` records the channel counts, `ModelBatch` holds the input tensors, and `EdgeContext` holds the graph and edge descriptors calculated from them. These records contain no learned modules.

For each directed edge `e`, `EdgeContext.source[e]` gives the sender's atom row and `destination[e]` gives the receiver's row. The other edge arrays contain the matching spherical harmonics, radial descriptors and optional electronic pair features.

Shift inputs use a larger record containing the `ModelBatch`, per-atom `shift_role` values and a `RotamerTensors` record of dihedral descriptors. `StaticShiftBatch` and `MdShiftBatch` inherit these fields from `_ShiftBatchBase`. The static subclass also holds the OpenFold residue embeddings.

Hidden features are calculated for every input atom. Shift predictions cover only selected atoms and are returned with their `atom_indices`. Oxygen and sulphur can therefore contribute to the molecular environment without receiving a shift prediction.

Read: ModelBatch and EdgeContext, the shift batch classes.

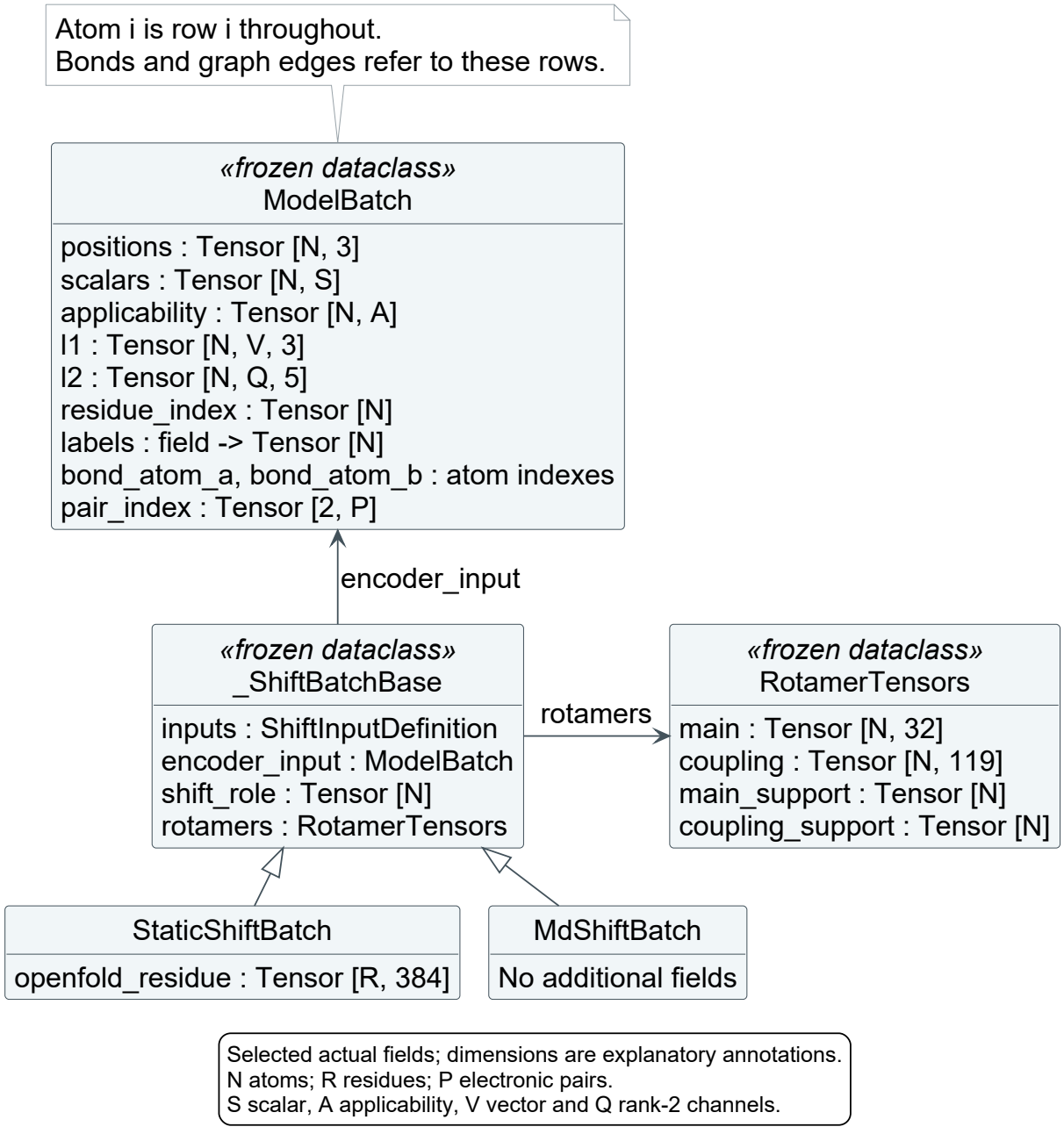


Figure A.1: Data records and atom indexing

A.2 Inside one message layer

`MessageConvolution` inherits from `torch.nn.Module`. Its constructor creates three principal child modules: `radial_weights`, `tensor_product`, and `self_interaction`. Together, they calculate what an atom receives from its neighbours and how its own features contribute to the next layer.

Two different linear modules appear here. The radial network uses `nn.Linear` to take weighted sums of rotation-invariant descriptors and add biases. The self-interaction uses `o3.Linear`, which transforms an atom's features while preserving their rotation rules. Between neighbouring atoms, `FullyConnectedTensorProduct` combines the sender's features with spherical harmonics of the edge direction, using the weights calculated by the radial network.

The geometric message is calculated by these lines:

```
source = context.source[edge_slice]
radial = self.radial_weights(context.radial_context[edge_slice])
result = self.tensor_product(
    node[source],
    context.spherical_harmonics[edge_slice],
    radial[:, : self.tensor_product.weight_numel],
)
```

`node[source]` gathers the sending atoms' current features for the selected edges. The second argument represents each sender-to-receiver direction with spherical harmonics. The third supplies mixing weights calculated from distance and the other edge descriptors.

Messages are added into the receiving atom's row using `index_add`. After counting incoming edges, the layer returns:

```
return self.self_interaction(node) + aggregate / degree.clamp_min(1.0)
```

The layer transforms the atom's own features and adds the mean of its incoming messages. With no incoming edges, only the self-interaction remains. The implementation processes edges in chunks to limit intermediate storage; the sum and normalisation follow the rule above.

When the inputs include electronic-pair measurements, a second tensor product calculates messages on the corresponding edges. It has its own learned weights, and a scalar gate scales each message. These messages are added to the geometric messages before averaging at the receiver.

Read: MessageConvolution.

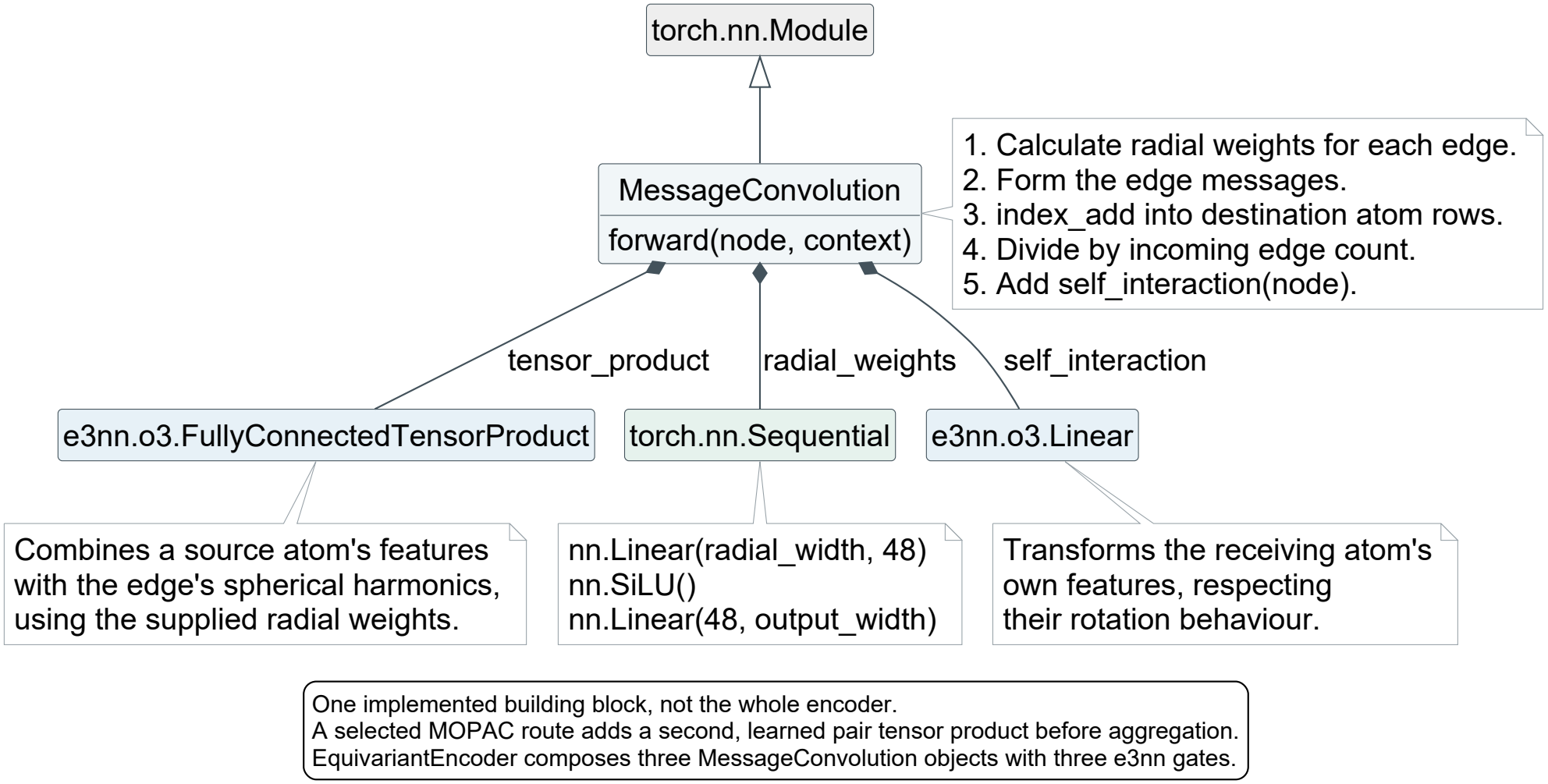


Figure A.2: MessageConvolution implementation

A.3 The encoder builds on that layer

`EquivariantEncoder.__init__` puts 17 embedding tables in `label_embeddings`, three message layers in `convolutions`, and three gates in `gates`. PyTorch's `ModuleDict` and `ModuleList` register them as child modules, so their parameters are included when the encoder is trained or saved.

`pack_nodes()` joins scalar measurements, applicability indicators, category embeddings and norms of the directional inputs into one scalar block for each atom. It also includes the full vector and rank-2 components in separate blocks with the corresponding rotation rules.

`edge_context()` constructs the graph once for the conformation. For each edge, it evaluates radial basis functions at the edge's length and spherical harmonics along its direction. The three message layers then use those same descriptors:

```
context = self.edge_context(batch)
node = self.pack_nodes(batch)
for gate, convolution in zip(self.gates, self.convolutions, strict=True):
    node = gate(convolution(node, context))
return node
```

The encoder returns 12 scalar, 12 vector and 12 rank-2 channels per atom: 108 components for the model's readouts.

Read: EquivariantEncoder, its forward method.

A.4 Constructing and calling the models

`ShieldingModel` contains one encoder, a shared equivariant readout and the residue:atom correction. Its forward method calls the encoder and sums the readouts' outputs. When the inputs include the peptide term, the model also adds its scaled contribution to the rank-2 prediction. The final output has six normalised components per atom.

Two functions construct the static and MD shift models through the same helper. Their calls differ in whether they request OpenFold inputs:

```
make_static_shift_model(...)
    -> _make_shift_model(..., openfold_width=384)

make_single_conformation_md_model(...)
    -> _make_shift_model(..., openfold_width=None)

_make_shift_model(...)
    -> construct one EquivariantEncoder prototype
    -> construct IndependentEncoderBranches from the prototype
    -> return ShiftPredictor containing those branches
```

`IndependentEncoderBranches` deep-copies the prototype for H, C and N, giving each encoder separate parameter storage. For the weight-transfer comparison, the fitting code copies the shielding encoder's learned parameters into all three before training.

The shielding model calls `EquivariantEncoder.forward`. The shift model instead calls the encoder's `pack_nodes`, `convolutions` and `gates` from its own `forward_nucleus` loop. This loop inserts the chemistry gates after layers two and three and retains the original invariant inputs for the local bypass.

Read: ShieldingModel.forward, shift constructors, the shift branch's forward method.

A.5 Follow one prediction call

The public `predict_static_shifts` and `predict_md_frame_shifts` functions both call `_predict_shifts`. Both take a `FittedShiftModel` containing the neural model and its saved normalisation values.

`prepare_shift_input` applies the saved transformations described in Section 2.2. `tensorize_shift_input` creates PyTorch tensors on the selected device and puts them in a `StaticShiftBatch` or `MdShiftBatch`. Both steps use the saved fitting results without learning new parameters.

Calling `fitted.model(batch)` runs `ShiftPredictor.forward`, which builds one edge context for all three H/C/N branches. For each element, the model adds the rotamer correction to the encoder branch's shift prediction. Each element's result occupies its own rows in an otherwise zero-filled atom array. Summing the three arrays combines the predictions in the original atom order.

Finally, `absolute_prediction` converts the predictions to ppm. `ShiftPrediction` returns those shifts with their atom indices. The prediction runs under `torch.inference_mode()` without experimental targets. For a trajectory, `md/evaluate.py::predict_trajectory` calls this procedure for each frame and averages the predictions afterwards.

The context-based rotamer correction uses `detach()` on the branch's invariant features. This leaves their values unchanged but, during training, prevents gradients from the correction passing back through those features into the encoder.

Read: the prediction entry points, preparation, tensorization, ShiftPredictor.forward.

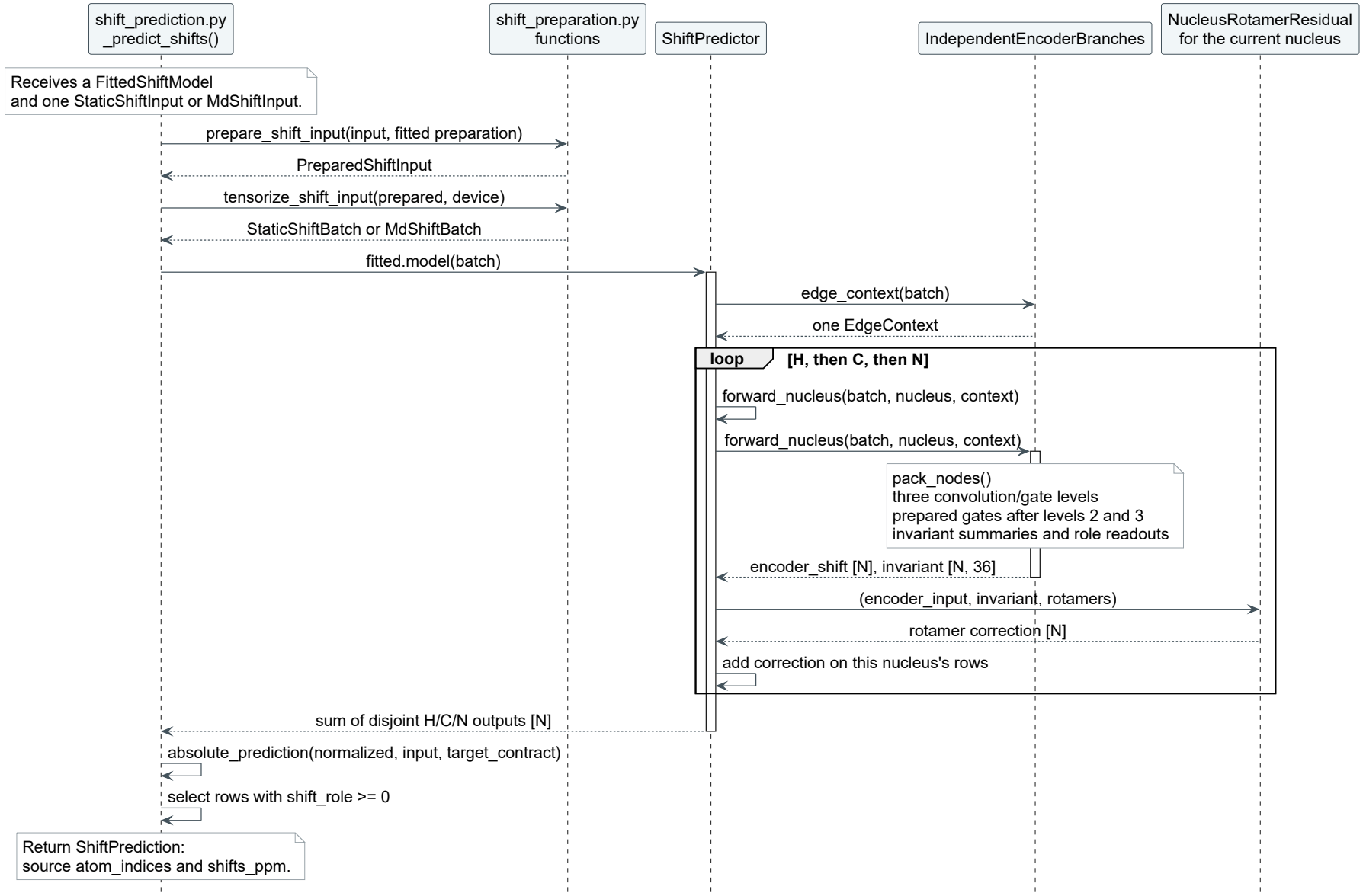


Figure A.3: Prediction call sequence

A.6 Where to read the code

File	What it contains
`equivariant.py`	Data records, one message layer, then the three-layer encoder.
`shielding/model.py`	Converts hidden features to predicted symmetric shielding tensors.
`shift_model.py`	Shift batches, branch construction, role readouts and residual additions.
`shift_transfer.py`	Which fitted encoder parameters are copied, and where.
`shift_preparation.py`	Transforms the inputs and creates tensors on the selected device.
`shift_prediction.py`	Predicts shifts for one conformation without experimental targets.
`shift_result.py`	Stores the fitted model with the input and target normalisation values.
`md/evaluate.py::predict_trajectory`	Per-frame prediction followed by uniform averaging.

The editable `code-*.puml` diagram sources are in `prediction/diagrams`. The working notes in `prediction/` give the rendering commands.

Appendix B

Inside the fitted networks

This appendix reports parameter counts for ten saved models and shows the connections in their tensor-product layers. The analysis loaded the fitted weights into the published Python implementation; it did not train new models or measure prediction accuracy.

B.1 Parameter counts from the fitted models

The table distinguishes models by their inputs. "AIMNet" means the structure inputs plus the 256-component AIMNet embedding, not all available AIMNet measurements. "MOPAC" means the structure inputs plus the MOPAC groups listed in `shift_parent.py::MOPAC_FAMILIES`.

Input condition	Shielding tensor	Static shift	Single-conformation MD shift
Structure	810,196	3,054,822	3,007,098
AIMNet	1,572,052	5,400,294	5,352,570
MOPAC	1,357,278	4,720,155	4,672,431
MOPAC + AIMNet	2,149,854	Not inspected	Not inspected

The inspected shift models used transferred shielding-encoder weights. The table includes the joint MOPAC–AIMNet shielding model, but no shift model with that combination was inspected. The fitted files are listed at the end of this appendix.

The analysis counts each registered parameter once, excluding buffers and the input and target normalisers. The counts agree with `torchinfo` for the same loaded models. They describe network size, not how much each component contributes to prediction accuracy.

B.1.1 Parameters in the encoder

The encoder contains categorical embedding tables, three message layers and three gates. Most of its parameters belong to `radial_weights`, particularly in the first layer:

Condition	Encoder total	First layer's radial network	First layer's direct pair weights
Structure	800,572	482,376	Absent
AIMNet	1,562,428	1,235,016	Absent
MOPAC	1,347,571	972,169	39,744
MOPAC + AIMNet	2,140,147	1,724,809	70,464

In each message, the geometric tensor product combines source-atom features with spherical harmonics of the edge direction. Its mixing coefficients come from the radial network, which reads edge descriptors that are unchanged by rotation. `tensor_product` therefore stores no learned coefficients of its own. The radial network learns how to calculate them for each edge.

By contrast, the electronic-pair tensor product stores its own mixing weights. The radial network supplies a scalar gate to scale the pair message on each edge. The layer's `self_interaction` also has learned weights. The ordinary e3nn `Gate` has none.

The radial network has two linear layers separated by a SiLU activation: `Linear(r, 48)->SiLU->Linear(48, w+q)`. Its parameter count is

$$N_{\text{parameters}} = 48(r + 1) + 49(w + q).$$

Here `r` is the number of input descriptors per edge and `w` is the number of coefficients required by the geometric tensor product. `q` is one when the electronic-pair gate is present and zero otherwise. The first linear layer has `48*r` weights and 48 biases; the second has `48*(w+q)` weights and `w+q` biases.

In the table's order, the first geometric tensor products require 9,816, 25,176, 19,800 and 35,160 coefficients. The AIMNet embedding supplies many scalar inputs, which increases the number of coefficients needed to mix them. The AIMNet encoder is therefore larger than the MOPAC encoder, even though MOPAC supplies more kinds of directional input.

Code: `equivariant.py::MessageConvolution.__init__`, `forward`, and `EquivariantEncoder.__init__`. The full breakdown is in `encoder-components.csv`.

B.1.2 Parameters outside the encoders

The shielding model's shared equivariant readout has 24 weights: twelve mix the scalar channels, and twelve mix the rank-2 channels. A rank-2 channel's five components share one mixing weight. The residue:atom correction accounts for another 9,600 weights. Models with MOPAC inputs also have 83 peptide tensor scale factors.

Each shift model contains three independent encoders, for hydrogen, carbon and nitrogen predictions. Further parameters belong to the chemistry gates, local bypasses, readouts and rotamer modules described in Chapter 2. Unlike the ordinary e3nn gates, these chemistry gates have learned parameters. The static model also has learned OpenFold projections.

OpenFold projections account for part of the difference between the static and MD parameter counts. The two training sets also contain different chemical categories, giving some embedding tables different numbers of rows:

Vocabulary	Static	MD
Amber residue	21	27
Residue variant	3	4
Polar hydrogen kind	10	12
IUPAC residue/atom pair	400	403

`vocabulary-sizes.csv` records every saved category count. `parameters.csv` records each parameter's name, shape, model component and count.

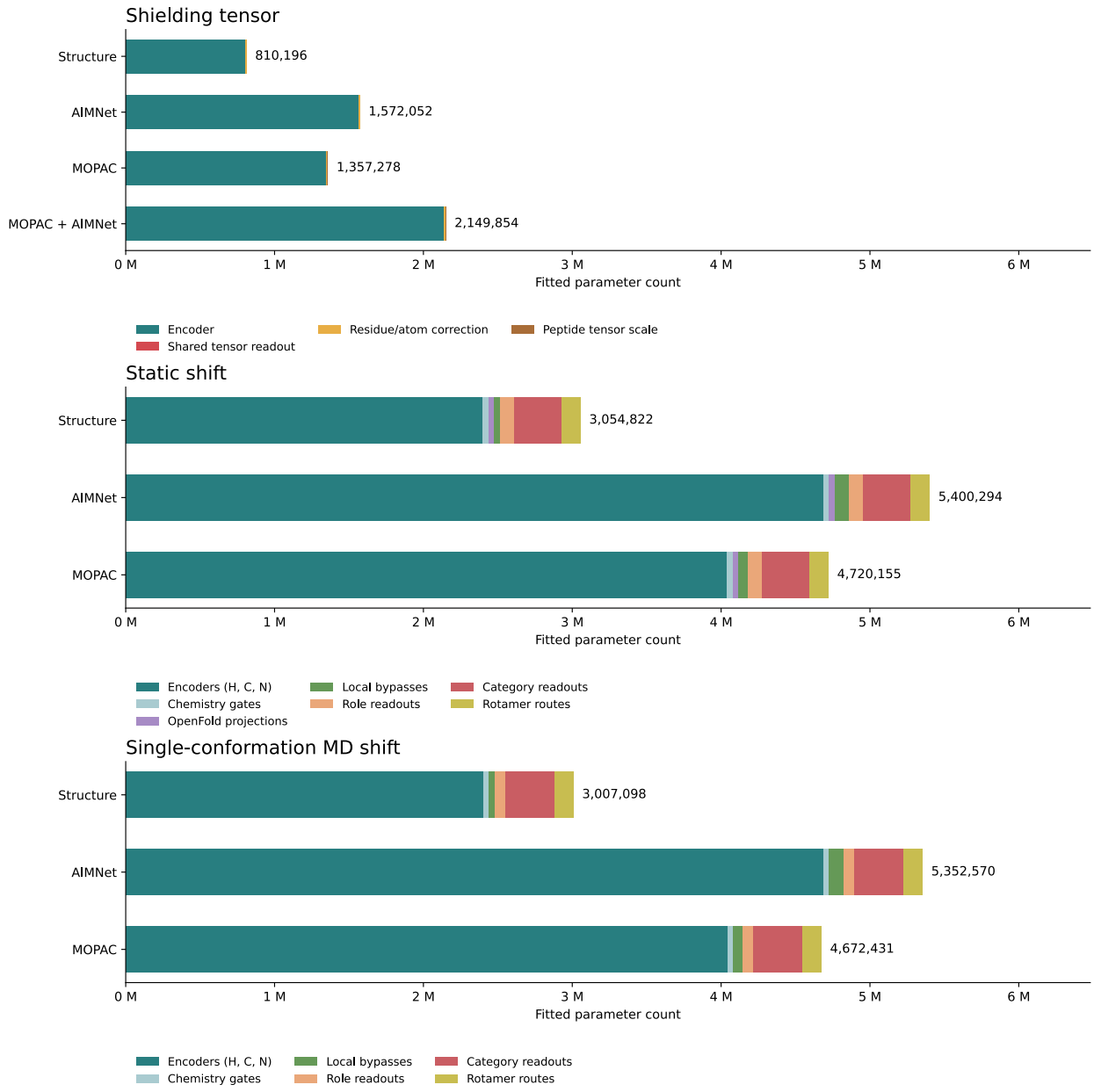


Figure B.1: Number of parameters in each component of the ten inspected models.

B.2 Reading the e3nn diagrams

In the tensor-product diagrams, the source atom's feature groups appear at the upper left. At the upper right are the edge-direction features, or electronic-pair features in the pair diagrams. Each three-way connection combines one input group from each side to produce an output group below. A connection includes all the channel combinations between those groups; it does not represent a single weight.

`0`, `1` and `2` label rotation types: scalars, vectors and five-component rank-2 features. For example, `12x2e` means twelve rank-2 features, each with five components, not twelve scalar numbers. The implementation uses SO(3) rotation equivariance and labels all representations with `e`. These labels do not specify how the physical inputs behave under reflection.

The encoder starts with these feature counts:

Condition	Scalars	Vectors	Rank-2 features
Structure	162	1	0
AIMNet	418	1	0
MOPAC	266	22	18
MOPAC + AIMNet	522	22	18

The scalar column includes category embeddings and norms of directional inputs as well as extracted scalar measurements. For example, the structure condition's 162 values comprise 35 scalar measurements, eight applicability values, 118 categorical embedding components, and one vector norm.

The structure-only model has no rank-2 input measurements. Its first message layer builds rank-2 features from geometry by combining scalar inputs with rank-2 spherical harmonics of the edge direction.

Before each e3nn gate there are 36 scalar values, twelve vectors and twelve rank-2 features. Of the scalar values, 24 control the vector and rank-2 gates; the other twelve remain scalar features. After gating there are `12+12*3+12*5=108` components per atom, arranged as twelve features of each rotation type.

Green lines show which connections are allowed. Darker lines arise where connections overlap, not where weights are larger or more important. The figures use `plot_weight=False` and do not show fitted weight magnitudes. For geometric messages, the radial network calculates coefficients for each edge, so there is no single set of fitted coefficients to colour. `tensor-paths.csv` records the connections, channel counts, coefficient counts and fixed normalisation factors.

Tool: e3nn TensorProduct documentation. The analysis script generates these figures with the `visualize` method of the installed e3nn 0.6.0 objects.

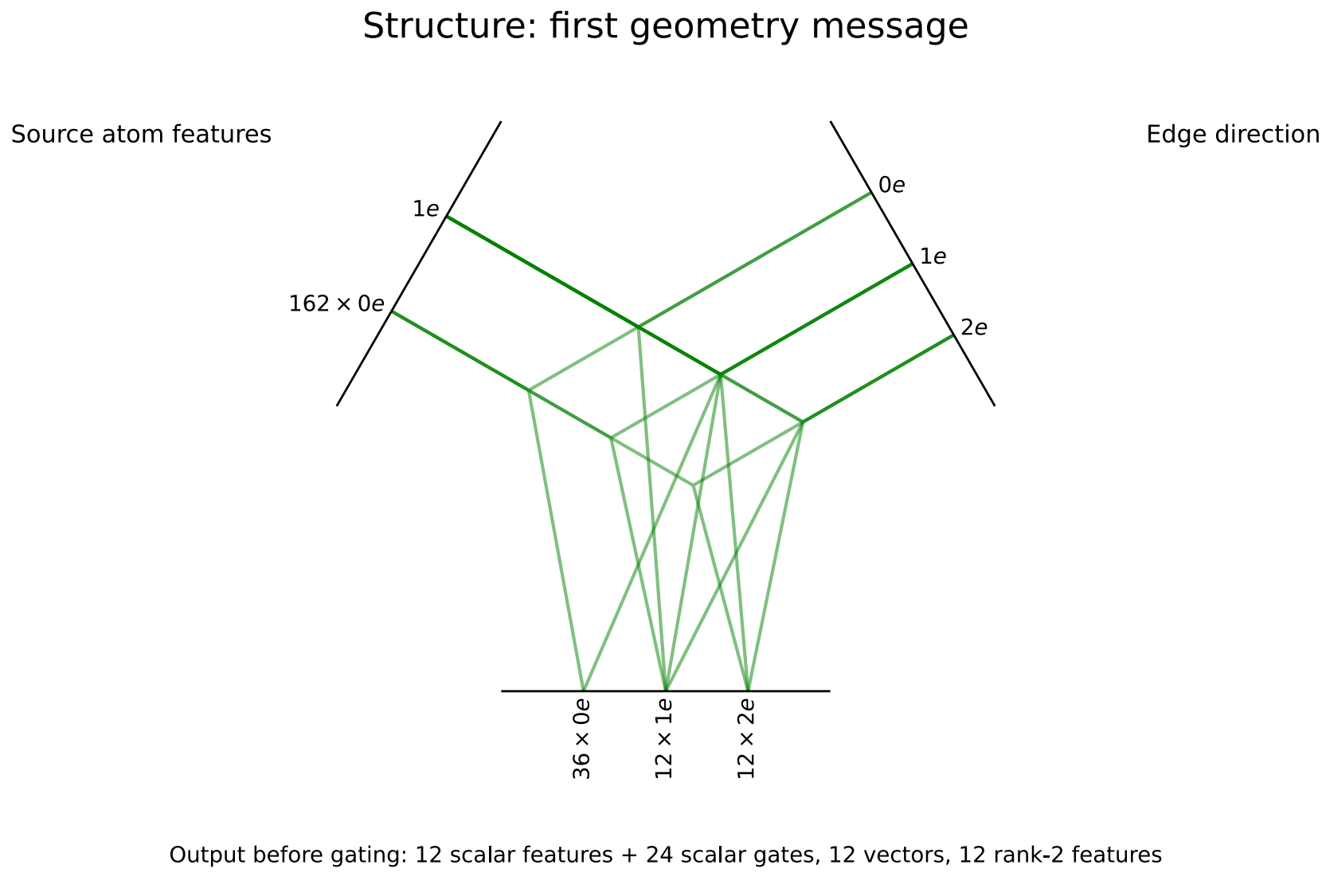


Figure B.2: The first structure-only tensor product, drawn by e3nn. Lines show the allowed combinations of input and output features.

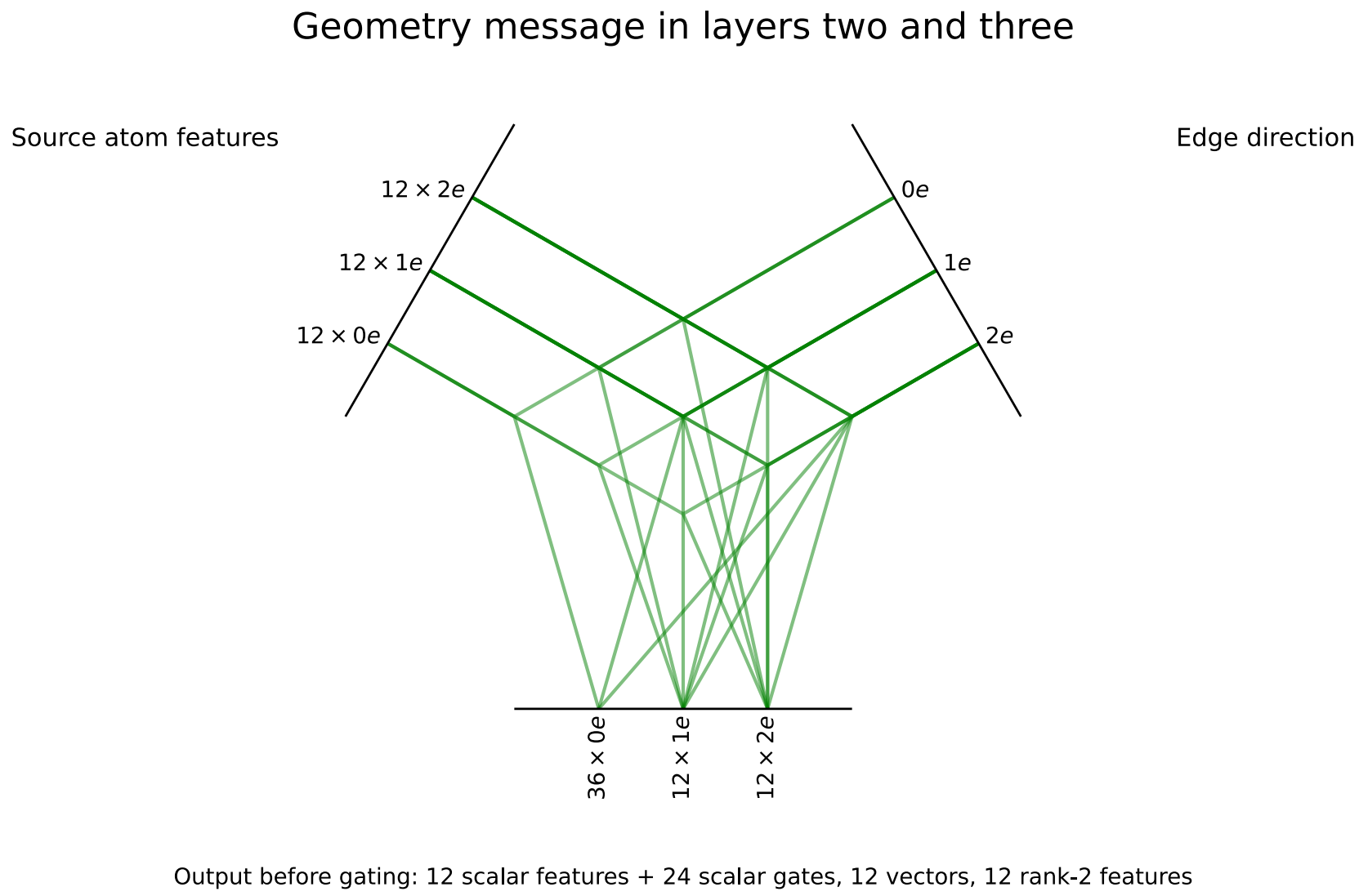


Figure B.3: Connections in the later geometric message layers. Layers two and three use this same pattern but have separate learned parameters.

B.3 Dependencies

Analysis with Python's syntax parser found 42 modules and 192 distinct import relationships between them. No module imports itself, and the dependencies contain no cycles. The count includes import statements inside functions and conditional blocks, whether or not they execute in a particular run. It does not include dynamic imports constructed from strings or function calls.

PyDeps found the same direct local imports for `equivariant`, `shift_model`, `shift_prediction` and `shift_transfer`. Its four raw reports are included. The complete `imports.csv` records filenames and line numbers for both local and external import statements.

B.4 Source and fitted files

Source: `T:/actualthesis/thesis-deliverables/nmr-models-submission`, the published nmr-models repository, revision `cb077c91faa0197d6fa86928981172f313e2bb6a`.

The fitted models are stored on Batcave under `/mnt/ugreen1/nmr-model-results/final/`:

```
shielding-fit/shielding_{structure,aimnet,mopac,mopac_aimnet}_foundation/model.pt
static-fit/{structure,aimnet,mopac}/weight_transfer/model.pt
md-fit/{structure,aimnet,mopac}/weight_transfer/model.pt
```

The analysis uses local copies in `local/fitted-models/` in this documentation project. It loads shift models through the published loading functions. For shielding, it constructs a `ShieldingModel` from the saved widths and vocabulary, then loads the complete saved state with strict key matching.

Both routes use `torch.load(..., weights_only=True, map_location="cpu")`. The saved parameters match the constructed modules in all ten models.

Appendix C

Reader source map

The paths below refer to the published protein-extract-and-view-tools repository, revision `8714b8ca6e859236f46d17e16c79dfe896780ff8`. They are relative to `h5-reader/src`. The local copy is at `T:/nmr-shielding-release/h5-reader/src`, also available on Batcave at `/shared/2026Thesis/nmr-shielding-release/h5-reader/src`.

Each entry identifies the code behind an explanation in the Reader chapter.

Topic	Start reading here	What the code shows
Protein ownership	`model/QtProtein.h`	Atom, residue and name vectors; `unique_ptr<QtTopology>`; explicit name accessors.
Typed atom identity	`model/QtAtom.h`	Atom, residue and parent indices; typed chemical categories rather than atom-name strings.
Residue grouping	`model/QtResidue.h`; `io/QtTopologySidecar.cpp:311, 542`	Builds residue membership from the sidecar's residue indices; `PopulateBackboneCache` records backbone atom indices.
Topology lookup	`model/QtTopology.h`, `QtTopology.cpp`	Stores bonds and rings; builds lookups from each atom to the bonds and rings it belongs to.
Atom and residue names	`model/QtAtomNames.h`, `QtResidueNames.h`, `QtProtein.cpp`	Stores names and retrieves them in a chosen naming convention.
Data ownership after load	`io/QtProteinLoader.h:22`; `.cpp:116, 168`	`QtLoadResult` owns the protein and conformation separately; the loader builds both from common sidecar data.
Common frame interface	`model/Conformation.h:25`; `.cpp:27`	Abstract accessors; current snapshot; synchronous `requestSnapshot`; readiness signal.
Single pose	`model/SingleConformation.cpp`	Position reads from the pose's `Pos` column; snapshot is shared.
Trajectory	`model/TrajectoryConformation.cpp`	Owns the H5 reader, maps frame numbers and finds each frame's NPY files.
Dense buffer layout	`model/QtTimeSeriesBuffers.h:210`	`QtPositionsTimeSeries::at`: `(atom*n_frames+frame)*3`.
Borrowed frame view	`model/QtFrame.h`, `QtFrame.cpp`	Holds a conformation pointer and row number; reads values through the conformation rather than owning arrays.
Frame identity	`io/TrajectoryFrameMap.h`, `TrajectoryFrameMap.cpp`	Maps stored rows to original frame numbers and back; finds sampled frames.
Loading NPY files	`io/FrameNpyLoader.cpp:31`; `QtFieldCatalog.gen.h`; `FrameFieldPolicy.h`	Maps filenames to `FieldKind`, checks row types and shapes, and loads selected columns.
Per-frame storage	`model/QtConformationSnapshot.h`	`array<NpyColumn, FieldKind::Count>`, original frame identity and borrowed protein pointer.

Topic	Start reading here	What the code shows
Typed result view	`model/QtRingKernelGroup.h`, `QtBiotSavartGroup.h`	Reads scalar, vector and tensor components from snapshot columns; returns an empty optional value if a column is absent.
Display transformation	`model/TransformedConformation.cpp:100, 117, 127`	`Rp+t`; explicit rotation accessor; unmodified underlying snapshot.
Loading a run	`app/ReaderMainWindow.cpp:613`	Owns the load result; creates the display wrapper, scene, playback and selection; connects the views.
Closing a run	`app/ReaderMainWindow.cpp:1337`	Detaches views and deletes the scene and controllers before releasing the data.
Shared selection	`model/AtomSelection.h`	`QAbstractListModel` of atom indices with focus and selection signals.
Changing frames	`app/QtPlaybackController.cpp`; `ReaderMainWindow.cpp:661, 707, 959`	Current-frame update and signal connections to scene, inspector and dashboard.
Render scheduling	`app/MoleculeScene.cpp:412, 535`	Updates geometry and combines pending render requests into one queued Qt callback.
Inspector detail	`app/QtAtomInspectorDock.cpp: 352, 373, 394`	Connects to the conformation, requests a snapshot and handles the signal that it is ready.
Dashboard roles	`model/TrajectorySignalCatalog.h`; `DashboardSignalModel.h`; `DashboardPanelModel.h`; `app/DashboardDisplayController.h`	Lists available measurements, records selections and stores samples for display.
REST uses the application	`app/RestServer.cpp:1956, 2192`	`/api/run/load` calls the window's loader; `/frame/set` requests a frame change through playback.
Model-input export	`io/ModelInputExporter.h`; `app/RestServer.cpp:1931`	Reads the loaded data and exports it in steps queued through Qt.

The compiler-generated JSON in `reader/analysis/generated/` records where classes and functions are declared. The event diagrams follow the signal connections made in the C++ source rather than listing every signal a class declares.

Appendix D

Sources and working files

D.1 The implementations described

The extractor and Reader descriptions refer to the published protein-extract-and-view-tools source release, revision `8714b8ca6e859236f46d17e16c79dfe896780ff8`. Reader is in its `h5-reader` subdirectory.

The prediction chapter describes the published nmr-models implementation at revision `cb077c91faa0197d6fa86928981172f313e2bb6a`, also used for the code walkthrough and model inventory.

Source paths are relative to the implementation described in each chapter. Line numbers refer to these versions.

D.2 The documentation project

This book belongs to the independent project `C:/projects/nmr-extract-architecture`. Edit the chapters in `latex/chapters/`; `latex/book.tex` assembles them, and `latex/layout.tex` controls the layout.

D.3 Diagrams and evidence

Editable Gaphor models for the extractor and prediction class diagrams are in `model/` and `prediction/model/`. Other explanatory diagrams use PlantUML or SVG sources under the corresponding `diagrams/` directories. The `latex/figures/` directory contains vector PDF copies for printing.

The numerical tables and class/function maps are in `prediction/analysis/`. Reader's compiler-derived class views are in `reader/analysis/generated/`.

D.4 Rebuilding

From the documentation project root, the local command is:

```
.\latex\build.ps1
```

The build uses a XeTeX-compatible engine and the included vector figures. It does not require the software repositories, datasets, fitted weights, Qt, Gaphor or Python. On another system, open `book.tex` with its folder as the working directory and compile with XeLaTeX; repeat until references settle. The supplied script uses Tectonic, which repeats those passes automatically.

The output is `output/pdf/book.pdf` under the project root.