openmaterials.ai

Git for science: a versioned map of physics

The OpenMaterials project

`openmaterials.ai` · open source: map data CC BY 4.0, code Apache 2.0

July 22, 2026

Abstract

openmaterials stores the object a scientific result *is*, not the printout of it: a typed physical quantity, the executable formula that produces it, its lineage, its units and gauge, and the codes that represent it. The map doubles as a semantic layer for AI: fuzzy natural-language labels resolve to typed, gated, content-addressed identities, so agents ground their language in checkable physics instead of text similarity. This document is the project's single source of truth. Part I states the vision (why science needs a git-like object model). Part II describes the product (the free protocol and the hosted app, the map, the parsers, the index, the contracts). Part III records the architecture (the operator and representation layers, the two-worlds separation, the formal definitions, and the first thermal-transport demonstration). Part IV documents the implemented kernel (dimension algebra and the dimensional gate, content identity, the log-first store, the frozen genesis, and what remains). Part V reports live status. The appendices carry the working procedures for ingesting a code, extending the DAG, and encoding a catalog skill.

Contents

Part I
Vision

Git did not just store files. It stored the *repository*: a structured, content-addressed object with identity and lineage, of which the working files are one view. That object is why merging, provenance, and collaboration became mechanical, and why a planet's worth of code could accumulate in one shared form. Science is still pre-git. We ship the paper, the printout of a result, and throw the object away. This is an attempt to store the object: what a result *is* (a typed physical quantity, its formula, its lineage, and the code that produced it as one representation of it), so that results become reproducible by construction, knowledge compounds instead of evaporating, and an AI agent has a semantic place to act.

The creed

1. **The unit of scientific knowledge is meaning, not text.**
2. **Physics is the invariant. A code is one representation of it.**
3. **A scientific result is a structured object with identity and lineage, not a number in a table.**
4. **A result you cannot re-run is a claim, not yet knowledge.**
5. **Units, gauge, and lineage are part of a result, not footnotes to it.**
6. **Knowledge should compound, not evaporate into scripts.**
7. **Correctness should be structural, checked by the substrate, not re-derived by trust.**
8. **Verification must scale with compute, not with reviewers.**
9. **The native substrate for scientific AI is semantic, not token text.**

1 The problem: science that does not accumulate

Computational materials science is a Tower of Babel. A single physical quantity, say the lattice thermal conductivity of a crystal, is computed by a dozen codes: kaldo, phono3py, ShengBTE, LAMMPS, VASP, Quantum ESPRESSO. Each has its own input formats, its own unit conventions, its own gauge and basis choices, and its own unspoken assumptions about what was held fixed. The physics is the same. The codes never quite agree, and deciding whether two of them *should* agree is itself expert work.

That work does not accumulate. Reconciling two calculations is artisanal, done by a handful of people who carry the conventions in their heads, and most published computational results

are effectively irreproducible because the real workflow lives in tribal knowledge and one-off scripts rather than in the paper. The paper is the tarball we email. Once it is sent, the object that produced it is gone.

2 The diagnosis: meaning is implicit

The reason the codes do not agree mechanically is that their meaning is implicit. Scientific knowledge today lives in three forms, and each loses something. Prose, in papers, carries rich meaning but does not run. Code runs, but its meaning is entangled with implementation: the physics is mixed in with array layouts, loop orders, and file formats. Data is numbers stripped of their provenance. None of the three carries meaning that is at once machine-checkable and runnable.

So the physics is never stored directly. It is buried inside whichever artifact happened to carry it, and recovering it, deciding what a given number actually claims, is left to the reader. What cannot be recovered mechanically cannot be checked mechanically, and so it cannot accumulate.

3 The bet: store the physics, not the printout

The move is to store the object. A result becomes a typed quantity, thermal conductivity or a phonon frequency or a heat capacity, declared as what it is. The formula that produces it rides on the edge that produces it, written symbolically rather than implied by code, and the quantity carries its lineage, the ordered chain of operations behind it, along with its units and gauge. A code is then no longer the result; it is one *representation* of it, a projection of a single shared operator layer into one discretization and one set of conventions.

Every representation factors through that shared layer, never code to code. That is what makes reconciliation mechanical: two codes that computed the same observable are two projections of one object, so comparing them is a structural operation on the graph rather than a feat of expert translation. It is merge, for science.

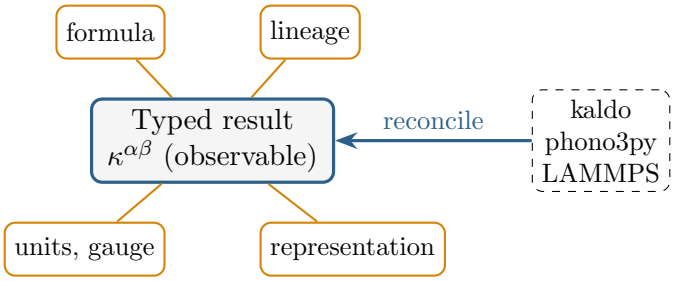

Figure 1: A scientific result as a structured object: a typed quantity carrying its formula, lineage, units and gauge, with each code a representation that reconciles into it at the observable. Merge, for science.

This is the same decision git made. A commit is identified by its content and its parents; a result here is identified by its type and its lineage. Git won on its object model, the blobs and trees and commits, not on its interface, and the contribution here is the analogous object model for a scientific result. Getting that object right is the whole game.

One difference is worth stating plainly, because it is where the real work lives. Code is discrete and born digital, while a scientific result carries approximation, continuous quantities, and gauge freedom: a per-mode lifetime depends on a basis choice no two codes share, even though the conductivity it sums to does not. That is why the object is typed, why units travel with it, and why the layer separates *observable* quantities, which must agree across codes, from representation-dependent scaffolding, which need not. Defining what is allowed to differ is as much of the design as defining what must match. The project already ingests a *code* this way;

the vision is to ingest a *paper* the same way, turning its claims into objects the substrate can hold, re-run, and check.

4 Proof it is real

None of this is a promise about future software; the substrate runs today. Take the lattice thermal conductivity of silicon. Computed through the operator graph, it comes out directly in watts per meter per kelvin, with the unit conversion applied automatically from the typed dimensions rather than pasted in by hand. The same conductivity assembled along two different routes, the total on one side and the sum of its population and coherence parts on the other, agrees to within about $10^{-6}\,\mathrm{W/(m\,K)}$ on a value of order 100, because the two routes are views of one object and the substrate holds them to it.

The repository ships a smaller version you can run in seconds. `examples/quickstart.py` builds the operator graph (54 quantities, 55 operators), derives the molar heat capacity of a crystal from a phonon frequency array, $11.83\,\mathrm{J/(K\,mol)}$ at $300\,\mathrm{K}$, and cross-checks two independent inputs at a gauge-invariant observable; the validation report returns expected agreement with zero residual. This is merge, for science, and it already runs.

The live operator graph is at openmaterials.ai/map; the architecture is Part III of this document.

5 The payoff: knowledge that compounds

When results live as typed, reproducible objects, they stop being isolated. Each one added to the graph makes the whole denser: a new code is a new representation that can be checked against the others, and a new quantity opens new routes that cross-validate the old ones. Reproducibility is the proof that a result was captured faithfully; compounding is the dividend. The value is no longer locked in any single paper but accrues in the shared, checkable object that all of them project into.

This is the network effect git set off. Git made a standard object for code, and on top of it grew GitHub, continuous integration, pull requests, and code review, an entire ecosystem that exists because the object underneath was shared and mechanical. A standard object for science invites the same: reproducibility infrastructure, automated review, and a body of physics that grows more valuable, and more trustworthy, with every result poured into it.

6 The horizon: a semantic action space for AI

There is a reason to care about this beyond tidiness. Today's models reason about science in token space: they manipulate text that describes physics, and they re-derive correctness from statistical pattern every time they are asked. That is why a model can produce a fluent, confident, wrong step whose error only surfaces three papers downstream. The substrate underneath is lossy, so the reasoning on top cannot be trusted by construction.

A typed substrate offers a different footing: a semantic action space. An agent that acts on the graph does not write a description of a calculation; it composes an edge, materializes a quantity in a chosen code, contracts a hidden quantity into an observable, cross-checks two routes. Each move is type-checked and dimensionally sound by construction, so an invalid step becomes a type error rather than a silent number. And because the routes are redundant, the substrate is its own verifier, which is the scarce ingredient for reinforcement learning in any open-ended domain: an agent can search and be told, mechanically, whether the answer holds.

The analogy closes the loop. The structured corpus of code that accumulated in git is what made today's code models possible; a structured corpus of science is what will let an agent reason over physics rather than over prose about physics. This is, deliberately, Lean for computational physics, and like a proof assistant its worth is that correctness is checked, not

trusted. It is only partly built, but the shape is already here.

7 Where we are, and what is next

Today the framework is real and growing. It has an operator layer and a representation layer, 31 ingested representations (kaldo, phono3py, phonopy, pymatgen, ShengBTE, Quantum ESPRESSO, VASP, LAMMPS, GPUMD, i-PI, xtb, the machine-learned potentials matgl / MACE / fairchem, the database rail mp-api, and the CALPHAD, amset, ORCA, OpenMM, and AtomisticSkills mat-* / chem-* workflows), and a validation engine that executes the graph, composes edges symbolically, and cross-checks redundant routes. Units reconcile automatically from the typed dimensions, and a dimensional gate checks every closed-form edge mechanically. The worked domains are lattice thermal transport (54 quantities, 55 operators), the Quantum ESPRESSO DFT ground state (the first contribution to land through the protocol gates), the mechanics domain (the elastic tensor, its Voigt moduli, and the pressure), the stability, thermochemistry, quasi-harmonic, molecular, and electronic-transport domains (grown from the AtomisticSkills scans), a first materials-diffusion subgraph, a thermodynamic-identities domain that closes the map's formulas together, and a composites domain (composite effective thermal conductivity with interfacial resistance); the unified map of 114 quantities and 114 operators is content-addressed in the log-first store (Part IV), grown past its frozen genesis version through the contribution gates. Its dimensional layer is proven in Lean 4 against physlib and recompiled in CI on every change to the map or its generators, its evidence carries pinned conformance targets a re-run must reproduce, and the whole is held together by a suite of 1732 passing tests.

What is next is mostly upward and outward. The upstream from the Born-Oppenheimer potential, the force constants from which everything descends, is currently stubbed rather than modeled in full. The symbolic composition of error, the approximation error carried by an operation and the discretization error carried by a representation, is designed and not yet built. And the largest reach, the one this document is really about, is to ingest not only codes but whole papers: to turn a published result into an object the substrate can hold, re-run, and check, the same way it already ingests a code. That reach has begun: the paper parser already turns a PDF into gated, quote-anchored evidence proposals and mints the share links that carry a whole paper's lineages; what remains is scale.

Git gave code a standard object, and an ecosystem grew on it. Give science the same, and a result stops being a dead end: it becomes a contribution to a compounding, machine-usable, verifiable body of physics.

The standard to store science.

Part II
The Product

openmaterials is a versioned map of physics knowledge, built like git: typed physical quantities (nodes) related by executable formulas (edges), every element content-hashed, every change logged. The structure comes from codes and theory: a respected codebase or a published derivation is what adds nodes and edges. Experiments fill it in: each simulation run or measurement attaches a value to a node, with its material, conditions, and source. Parsers turn the world's codes, papers, and data files into both, and an index in the same repository collects them. The map compounds: every code mapped and every value attached makes the neighborhood more cross-checked, more reproducible, and more trustable.

The picture has two tiers, the way git has GitHub. The **openmaterials protocol** is free and open forever: the map format, the hashing rules, the versioned change-log, the schemas, the parser contracts. The **openmaterials app** is a hosted service on top: upload a map and see it rendered instantly, or store your own custom and private maps. The protocol is the commons; the app is convenience built on it.

8 Protocol and app

The protocol is a standard anyone can implement, self-host, and fork. The mother map and its index are the shared artifact the community builds through the protocol; they belong to no one and to everyone.

The app (the website, a separate project) adds the hosted conveniences: paste and upload a map and have it rendered immediately in rich traversal views, the way mermaid.live renders a diagram with no setup, and optionally store custom or private maps. Anything hosted lives entirely app-side; it never touches the protocol or the commons. The app is free while we build it, and how it is funded later is out of scope here. The priority is the protocol and the map.

The priority is explicit: first build the community that builds the mother map, and a rich initial map worth using. The app comes second. Protocol adoption is what makes the app valuable, the same way git had to matter before GitHub could.

The site already carries a first taste of the app: a Learn page that runs the paper parser as a public demo. Because that demo calls a paid model, it runs through a cost-gated Cloudflare Worker (`infra/learn-proxy`) rather than from the browser: the Worker holds a daily budget and per-IP limits and keeps the API key server-side, never shipping it to the page. The Learn page is also a contributor on-ramp: a value a reader accepts from the parser's output is exported as an instance JSON in the committed schema, with a prefilled GitHub pull request link, so contributing a value from a paper is one reviewed PR through the same gates the maintainers use.

9 Structure and evidence

Two kinds of thing live on the map, and they come from different places. This is the load-bearing distinction.

Structure is nodes and edges. It comes from **sources**: a code (the kaldo, QE, or LAMMPS codebase) or a theory paper. A source asserts that a quantity exists and that a formula relates it to other quantities. Codes are the ground truth we seed from, because a respected code is an executable statement of the physics; theory papers and, later, new derivations extend the structure.

Evidence is values on nodes. It comes from **instances**: one committed value, computed or measured. An instance attaches a value to a node, with the material, the conditions, and a citation (a whole run is the separate simulation record). Evidence never creates structure: a simulation cannot invent a formula, it can only put a number on a node that already exists.

The split is also a separation of concerns in the app. The map view renders the structure alone (nodes, edges, formulas, and the code representations that map onto them); the four data record kinds beside the ledger (instances, configurations, spectra, and simulations) are a distinct evidence layer that consumers load against node uids and attach locally, never carried on the map page itself. The test suite and the contribution gates exercise that layer directly, so its correctness is checked against the same uids the view is drawn from.

The two relate through a small type hierarchy. A code is a **representation**: a named mapping of part of the map onto a concrete tool, with its units, gauge, and conventions. One committed value that code produced is an **instance**: the kappa a run outputs is a **value on the kappa node**. A growth process such as CVD is an experiment type, also a representation;

a specific synthesis and measurement is an instance; its measured number is a value on a node. So "is ShengBTE a tag?" resolves cleanly: ShengBTE is a representation, a ShengBTE run is an instance, and its result is a value on a node.

One name per concept is enforced throughout this document. The canonical terms, and the near-synonyms they subsume:

Concept	Canonical term	Notes
A typed quantity on the map	**node**	`Space` is the code class.
A relation producing a node	**edge**	`Operator` is the code class.
A code's mapping onto the map	**representation**	its on-map rendering is a *rail*.
A committed value	**instance**	the {`id, kind, lineage, source`} record.
A semantic type parameter	**label**	e.g. `bte_solver=rta`, not a "variant".
A content hash	**uid**	sha256; a 12-hex prefix is displayed.

10 The map

The map is a curated reference graph. Today it holds 114 nodes (111 observable and hidden quantities plus 3 promoted parameters) and 274 edges. Nodes are typed physical quantities. An edge produces one node from a list of input nodes, carrying the symbolic formula that relates them (parameter inputs are marked as inputs rather than derived through a formula).

Identity is structural plus one curated semantic tag. A node's id is the hash of its quantity tag, its field signatures (dimension and index kinds per field), its gauge class, and its labels; an edge's id is the hash of its output and input node ids, its formula fingerprint, and its schemes. The quantity tag exists because physics is full of same-typed distinct quantities (entropy and heat capacity share dimension, indices, and gauge); tags, index kinds, labels, and gauge groups live in small controlled registries that are part of the protocol. Names and symbols are not part of identity, so your `kappa` and my `k_L` are the same node when you both map to the registered quantity, and a formula written `c v^2 tau` or `tau v^2 c` is one edge because both normalize to the same fingerprint. Genuinely different decompositions stay separate graphs and are reconciled where they meet, at a shared observable; production routes are never part of a node's identity, so adding an alternative derivation of an existing quantity never re-mints the node. The consequence is that two people who contribute the same physics converge through the shared registries, while a real difference in decomposition surfaces as a parallel route rather than a false merge. The implemented form of these rules, with the collision rationale that forced the quantity tag, is Part IV.

Three properties propagate, rather than being hand-stamped on every node:

- **Units** live on leaf nodes and propagate through operations; a derived node's units are computed, never stored.
- **Gauge** lives on index kinds. Mode and branch indices, and the phase of an eigenvector, carry a representation's gauge freedom; a node is **observable** once every gauge-carrying index has been contracted away, and **hidden** until then. Cross-source values may be compared only at observables.
- **Symmetry** lives on index kinds too. A quantity's intrinsic index symmetry (the thermal-conductivity tensor satisfies $\kappa^{ab} = \kappa^{ba}$) belongs to the node; a material's crystal symmetry (cubic silicon making κ isotropic) acts on the spatial indices at the material and instance level, not on the node. Symmetry is what proves that one code reporting κ_{xx} and another reporting the trace over three are reporting the same observable.

The map is versioned like git, log first. The change-log is the map: an ordered sequence of operations (add, edit, or deprecate a node; add or remove an edge; supersede an element), each

carrying a date, a reason, and an author. A materialized current version is always kept in sync, so reading the map is immediate. The map version hash chains the log, computed as the hash of the previous version hash combined with the new change record, which makes the history tamper-evident and path-dependent. Because identity is structural, editing a foundational leaf re-mints every node above it (a Merkle ripple); the supersede record ties the old subgraph to the new one. The design gives a value two currencies, stay pinned for reproducibility or follow the supersede chain; today the committed records carry no pin, and the regenerated projection stamps each value with the live node uid, so published values follow the current map.

Provenance and confidence ride on every element. A node or edge records the sources that assert it (which codes and papers) and accumulates the instances that put values on it. Many independent instances are a confidence signal, but independence is not the same as multiplicity: two solvers run on the same interatomic potential are not independent confirmations, so the map keeps the full provenance (potential, method, code) and lets independence be judged rather than counting raw repetitions. A claim no instance has reached yet still lives on the map, simply unconfirmed.

11 The semantic layer

The map speaks symbolic language: typed nodes, executable formulas, proven dimensions. The literature and the language models that read it speak semantic language: labels like "quasi-harmonic approximation" or "phonon-thermal-conductivity". The semantic layer (2026-07-12) connects the two as a first-class artifact: every node and edge carries its cloud of surface labels (auto-derived forms plus a curated alias registry, reviewed like vocabulary), published uid-pinned in `docs/data/semantics.json` and regenerated with the map. A deterministic resolver turns any fuzzy label into ranked typed identities: normalization, exact alias hits, token containment, no embeddings, so every resolution is auditable. An agent that grounds its phrase through the resolver inherits everything the map knows about the element: the formula, the dimensional proof, the producing codes with their citations, the evidence with its provenance. A label that resolves to nothing is an honest coverage gap and feeds the encode queue. External method vocabularies (the first being a 16,012-paper skill map extracted by large-scale literature analysis) will align to the map through reviewed alignment files, each unmatched label ranked by its usage frequency in the field; today the curated aliases carry that role while the corpus is awaited. This is the sense in which the map is a semantics for AI: fuzzy language in, checkable identity out.

12 The parsers

A parser is an assisted on-ramp, one per artifact type. Each proposes structure, values, or both; the map validates them; a human confirms. One of the three exists today; the other two are designed and not yet built.

- The paper parser turns a paper's measured or computed numbers into values on nodes (and, in its eventual full form, its formulas into edges). Its value-extraction half is built (`omai/paper_parser`, gated evidence), detailed under the parser contract below.
- The code parser will import a codebase as a representation: mapping the code's input, output, and intermediate files and key lines onto the nodes and edges the code implements (structure), and letting a run of the code be recorded as an instance (values). The 31 representations on today's map were authored by hand against this contract; the parser that automates the mapping is not yet built.
- The run parser will read a concrete run and record its inputs (from the input file) and its outputs as values on the nodes they belong to.

Proposals must pass the map's types as continuous integration: dimensional agreement; reachability, so a value lands only on a node that already exists and a new edge is introduced only by a code or a theory source, never by a bare experiment; observable discipline, so cross-source equality is asserted only at observables; completeness, so every input and output of a run is recorded; and coherence, so one source's contributions form a connected subgraph. All the difficulty of joining the commons concentrates here, in the parsers, which is what keeps the store itself simple.

13 The index

The index is a subfolder of the map repository, in two clearly separated parts.

The canonical registry is organized by source: `papers/`, `codes/`, `experiments/`. Each entry holds that source's structure contributions and the values its instances produced, pinned to a specific element hash at a specific map version.

The derived lookups are regenerated from the registry and the map, never edited by hand: symbol to node, value to instances, source to coverage.

One clone gets you the map and the world's evidence together.

14 Contracts (for builders)

These are the interfaces the protocol defines. The store, index, and identity contracts below are implemented (Part IV); of the parsers, the paper parser's value-extraction half is built and the code and run parsers remain the eventual on-ramp, each its own spec, plan, and build cycle.

Store operations. `push(change)` appends a change record (op, target, date, reason, author) and advances the version hash. `read()` returns the materialized current map. `read(hash)` returns the map at a given version. `diff(hashA, hashB)` returns the change records between two versions. Identity is by content: a node id = hash(quantity tag, field signatures (dimension + index kinds per field), gauge class, labels); an edge id = hash(output ids, input ids, formula fingerprint, schemes); the version hash = hash(previous version hash + change record), unchanged (quantity tags, index kinds, labels, and gauge groups live in controlled registries).

Structure contribution (from a source). A code or a theory paper proposes new nodes and edges. Inputs to any new edge must resolve to nodes that already exist or are introduced by the same source in the same contribution, processed in dependency order. Reviewed before it lands in the canonical map.

Value record (a lineage instance). A committed value is a *lineage instance*: one construct, {`id, kind, lineage, source`}. The `lineage` is the complete computational identity of the value: the node, the material, every condition that determines the number, and the values block. `id` is the sha256 of the lineage's canonical JSON, so two records with the same id are the same computation, and any change to the canonicalized lineage (floats are rounded to six decimals before hashing, so re-serialization noise never re-mints) yields a different id. The implemented schema, shown as the committed instance, with the one long `detail` string wrapped for print (`docs/data/instances/si-thermalconductivity-bte-solver-rta-kaldo.json`):


```
{
  "id": "04c6dbdb9b046aceefb0c7b44e4219574cb2259280bc6893c0180b620fbcea52",
  "kind": "simulation",
  "lineage": {
    "node": "ThermalConductivity[bte_solver=rta]",
    "material": "Si",
```

```
    "conditions": {"T": 300, "mesh": "8x8x8", "potential": "Si.tersoff"},
    "values": {"value": 19.46, "units": "W/(m K)"}
  },
  "source": {
    "kind": "simulation",
    "ref": "kaldo",
    "detail": "RTA; Tersoff; 8x8x8 mesh (cross-code meaningful,
               absolute under-converged)"
  }
}
```

`lineage.node` always names a node, because a value belongs to a quantity; at bundle time each record is additionally pinned to that node's content uid (`node_uid`, injected into the flat, regenerated projection `docs/data/instances.json`, never into the hash). A lineage may also carry its own `source` as a namespaced `scheme:ref` string (`paper:...`, `doi:...`, `zotero:...`, `arxiv:...`; the scheme set is open), and that in-hash source is identity-bearing: the same claim from the same source mints the same id, while one paper's claim and another paper's identical number stay distinct records. The top-level `source` block is display provenance and rides outside the hash forever; a record whose two sources conflict is malformed. The material is deliberately NOT part of the map's structure: the map's nodes are quantities of physics in general, and the material enters only here, on the value, as the point where the quantity was evaluated (with `conditions` carrying the rest of the evaluation point). A value record never introduces structure; that is a separate, reviewed contribution through the gates.

Configuration record (structure-valued evidence). The `Structure` node is one opaque node on the map, but a value of it is a periodic atomic configuration: lattice vectors, fractional coordinates, and species. That payload lives in a configuration record, the structural sibling of the value record. It is content-addressed by a canonical uid: the sha256 of the spglib-standardized primitive cell (`symprec` 10^{-3}), origin-anchored so a rigid translation is invariant, with species and sites sorted and coordinates rounded to five decimals (a hundredth of a milliangstrom: coarse enough to absorb refetch-level numerical noise, fine enough that physically distinct cells never collide; the matcher gate reviews the boundary). The same cell, its sites shuffled, and a supercell of it therefore hash to one uid; a strained cell does not. On top of the hash a validation gate runs `StructureMatcher`: a physically-equivalent but not hash-identical cell is flagged for human review rather than silently merged, and provenance appends to the one record on a confirmed duplicate. The implemented schema, shown as the first committed record with the site list elided (`docs/data/configurations/si-diamond-primitive-mp-149.json`):

```
{
  "name": "Si diamond primitive (mp-149)",
  "formula": "Si",
  "natoms": 2,
  "structure": { ...pymatgen Structure.as_dict()... },
  "canonical": {
    "uid": "55bf22ca8186...",
    "spacegroup": 227,
    "natoms_primitive": 2
  },
  "external_ids": {"materials_project": "mp-149"},
  "provenance": [
    {"kind": "database", "ref": "materials-project",
     "detail": "mp-149, GGA-relaxed primitive cell"}
  ],
```

```
  "files": null
}
```

The record is pinned at bundle time to the `Structure` node's content uid, the same `node_uid` discipline a value record follows, so a configuration travels with the node through supersede chains. Cells at or below 1000 atoms embed the full structure dict inline; larger cells (a 17k-atom amorphous cell) point to a committed structure file and keep only reduced metadata inline so the record stays searchable. A value record may now carry an optional `configuration` key naming this uid; `material` stays the display string, so the addition is backward compatible and the existing instances are untouched. This is where the structure-ish numbers a run reports (atom count, cell volume) belong: not as scalar evidence on their own nodes, but as derived views of a configuration.

Spectrum record (function-valued evidence). Some nodes are function-valued in practice (a phonon density of states, a diffraction pattern, a dielectric function): the evidence is not one scalar but an array of ordinates against a monotonic axis. A spectrum record keeps the flat pre-lineage shape (its migration to the lineage construct is pending): the scalar is replaced by an axis and an array, attached to the same nodes and pinned the same way (`node_uid` at bundle time). Each spectrum-capable node's representation declares a *canonical axis* (a registered unit and its dimension) so the axis and value units can be validated against a fixed convention; the axis must be strictly monotonic and the arrays equal-length. The implemented schema, shown as a real committed record with the long arrays elided (`docs/data/spectra/li-phonondos-mat-phonon.json`):

```
{
  "variable": "PhononDOS",
  "material": "Li",
  "conditions": {"structure": "BCC",
                 "model": "TensorNet-MatPES-r2SCAN-v2025.1-PES",
                 "supercell": "3x3x3",
                 "normalization": "states/THz per unit cell"},
  "axis": {"name": "omega", "units": "linear_THz",
           "values": [-0.771482, -0.724914, ..., 8.542130]},
  "values": [0.0, 0.0, ..., 0.0],
  "units": "states/THz",
  "uncertainty": null,
  "source": {
    "kind": "simulation",
    "ref": "mat-phonon",
    "detail": "AtomisticSkills mat-phonon Li BCC example, total_dos.dat
               verbatim (201 bins, linear THz axis) ..."
  }
}
```

The DOS density carries no registered unit: its normalization (per cell versus per formula unit) rides in `conditions`, so the canonical axis pins the frequency axis and leaves the ordinate unit open. Diffraction is deferred: the XRD pattern's canonical axis (the wavelength-free plane spacing d_{hkl}, with the radiation wavelength a required condition of any served 2θ axis) is recorded as a representation note, and a spectrum against it becomes admissible only when an `XRDPattern` node is minted.

Simulation record (a light, lineage-identified, URL-first run). The value, configuration, and spectrum records each capture one number, one cell, or one curve; a *simulation*

record captures a whole experiment, and it is deliberately *light*: it stores whatever we have, with no fixed reproducibility guarantee and no required heavy input files. Its core is the `lineage`: the map node when one is known (with a content-uid pin), else a `template` with its `hyperparameters` and setup `values`, plus `material`, `conditions`, and `params`. Identity is the lineage alone, content-addressed by the sha256 of its canonical JSON with the same float-rounding discipline the configuration uid uses. Everything else rides *outside* the hash: an optional `execution` block (code, version, seeds, wall time, whatever the run recorded); optional `artifacts`, which are pointer-only, each {`path, role, url?, sha256?`} where only path and role are required and the url points at the heavy bytes on MaterialsCodeGraph (MCG), the cheap host, never embedding them; a `mirrors` resolver; and the run's `results`. Attaching a trajectory pointer, moving its bytes, or re-running on another node therefore never re-mints the record. Because the record is light, it is URL-encodable: gzip and base64url pack it into a `#x=` link fragment that opens in the openmaterials playground and, when the lineage names a map node, lights that node on the map, so an experiment is a link, no server, that carries the lineage and its receipts together. The get is that link; the heavy bytes are a separate, optional enrichment. A record with no artifacts and no map node is valid and normal: validation resolves a node by id and uid when the lineage names one (a stale pin is a mismatch, never a silent pass), and simply flags node-unresolved otherwise rather than rejecting it (the honesty rule). A full, checksummed bundle from MCG can additionally enrich a record with verifiable byte pointers, which a verification tool then checks against their checksums as a dated report, never a gate; but that heavy path is optional and never required for a record to exist or to have identity. This is the same bytes-out-of-git rule the governance commits to (the open format holds the lineage and its identity, the platform holds the artifacts, referenced by pointer), made into a first-class record kind.

The share envelope (multi-lineage bundles). A parsed paper rarely yields one lineage, so the `#x=` wire carries an *envelope*, {`v: 1, doc, lineages: [...]`}: shared document metadata (`doc.source` as a `scheme:ref`, plus title, authors, year, journal) and one or more lineage records, gzip-compressed and base64url-encoded into the fragment. The reader is dual forever: a bare single record decodes as a one-element envelope, so every link minted before bundles keeps rendering unchanged. A bundle opens as one page of stacked datasheets, one per lineage, each with its derivation drawn as an excerpt of the map; `doc.source` is inherited by members at read time for display only, never hashed into any member's id. The bundle itself is content-addressed (`bundle_id`: the sha256 of the canonical document metadata plus the member ids in order). A link that would exceed 8000 characters is refused with a download-the-JSON fallback rather than silently truncated: the honesty rule, applied to the wire.

Conformance target (a pinned expectation). The check side of the evidence. A target is {`id, lineage, code, expected, tolerance, evidence?`}: the number a named code is expected to reproduce for one committed computation, within a stated absolute or relative tolerance. A target never invents its lineage: it carries the evidence instance's lineage verbatim, so the hash of `target.lineage` equals `target.id` equals the instance's id, a cryptographic not-invented proof rather than an asserted link, and `expected` mirrors the instance's own values. (A literature target may carry no evidence instance; its lineage then names a `doi:` or other `scheme:ref` source in-hash, and the citation itself is the provenance.) The committed targets are projected into a byte-stable index that the playground datasheets read: a record's page lists, under *Reproduce this*, the codes that can compute its quantity and the pinned runs for its node, with a target whose id equals the record's id marked as the same computation. The first engine-adapter targets (kaldo and phono3py silicon thermal conductivity, every knob pinned in-hash down to the sha256 of the potential file) landed 2026-07-18.

Parser contract. Input: one artifact (a codebase, a paper, a run). Output: proposed structure and value records. Every proposal must pass validation below before a human confirms it into the index.

The paper parser, implemented. The first parser exists: `omai/paper_parser` turns a PDF into a gated evidence proposal in six stages. INGEST extracts the text with page offsets. DETECT enumerates every reported value with a verbatim, page-located quote through a citations-enabled model pass, run as an ensemble: a single pass is unreliable (its recall swung between 0.375 and 0.875 across runs because one pass can miss values), so three independent passes with diverse prompts (a broad sweep, a table/caption sweep, a prose sweep) are unioned, two claims counting as one finding when they share a normalized value, a compatible quantity name, and an overlapping quote or adjacent page. MAP aligns each detected value against the node catalog via structured outputs; a claim that lands on a source or parameter node (`Structure`, `Temperature`, `AtomCount`, and the rest of the input set, flagged `evidence_target: false` in one place in the catalog builder) is classified as a run *condition*, not a value. Printed numerals are normalized before the value gate (thousands separators, the comma-space artifact of a line break, the unicode minus, and $a \pm b$ forms, whose central value is the claim), because PDF extraction mangles real numbers into shapes a bare parser rejects; on the first paper this recovered the three known claims that formatting alone had killed. Such claims survive into the proposal as context and are counted separately, but are excluded from instance minting at apply time: an atom count or a temperature is a condition of a run, whose structural home is the configuration record and the `conditions` field, never a scalar instance on its own node. VALIDATE runs deterministically over the kernel, killing any claim whose quote is not found verbatim in the document (the hallucination gate) and flagging values that duplicate a committed instance. REVIEW is an adversarial model pass. PROPOSE emits a proposal file that a human confirms with `--apply` before any instance is written. The pipeline never prints or logs the API key. It is golden-evaluated against this very document, against a committed expected set of eight values: the acceptance bar is recall ≥ 0.8 with the worst of the ensemble runs still clearing it, no hallucinated quote surviving validation, and every recovered known correctly flagged as a duplicate of its committed instance. The ensemble was built to hold that recall bar against the single-pass variance, which is the honest edge of a model-driven extractor.

Index schema. Registry: `index/papers/`, `index/codes/`, `index/experiments/`, each entry pinned to (element hash at map version). Derived files (regenerated, never hand-edited): symbol-to-node, value-to-instances, source-to-coverage.

Validation rules. Dimensional agreement on every edge. Reachability: a value lands only on an existing node, and a new edge is introduced only by a code or theory source. Observable discipline: cross-source equality is asserted only at observables. Completeness: every input and output of a run is recorded. Coherence: a single source's contributions form a connected subgraph.

15 Governance

The protocol is forkable: anyone can self-host a map or fork the mother map. Landing a contribution in the canonical maintained repository goes through a reviewer list. Content-addressing handles identity and deduplication on its own; the reviewers decide what enters the commons.

16 Status and build order

Today's map is one hundred thirty-nine change records past v1, its frozen genesis version: 111 typed quantities plus 3 promoted parameters, the symbolic formula on every relational edge, 31 representations mapped (kaldo 34 variables, phono3py 31, phonopy 22, pymatgen 22, ShengBTE 20, mp-api 13, QE 13, LAMMPS 12, VASP 10, GPUMD 8, pycalphad 7, i-PI 4, and a further nineteen codes and mat-* / chem-* skills at five or fewer each), and real instances computed through the framework: cross-code silicon thermal conductivity (Tersoff potential, 8x8x8 mesh, 300 K: kaldo 19.46 RTA and 26.91 direct, phono3py 16.74 RTA and 24.30 direct, in W/m K) plus an LGPS activation energy (0.152 eV) from the materials domain. It is live and browsable as an interactive map.

The contributor on-ramp today is curated pull requests reviewed by the maintainer list; the silicon values above arrived that way. The genesis version hash is now frozen and the log-first store and the index exist, so the map's source of truth is data (an append-only log plus a materialized view) and the Python operator layer is an authoring client that emits change records; Part IV documents all three. The parsers are the automated on-ramp, not the day-one one: the paper parser's value-extraction half now exists, and the code and run parsers remain the next build after the store and index.

See also

- The vision, why this matters: Part I (Part I of this document).
- The architecture, how the operator and representation layers work: Part III (Part III of this document).
- The map, live: `docs/map/` (openmaterials.ai/map).
- The codes bibliography, every cited code with its interface, citation, and license: `docs/codes/` (openmaterials.ai/codes).
- The version badge, a repository's statement of the map version it used: openmaterials.ai/badge/<version>.svg (the current version lives at /badge.svg).
- The verified layer and its roadmap, live: `docs/lean/` (openmaterials.ai/lean and /lean/roadmap/).
- This document, browsable with a PDF download: `docs/document/` (openmaterials.ai/document).
- The deck, a short walkthrough: `docs/deck/`.

Part III
The Architecture

This part records the design of the typed operator/representation framework: the two-worlds separation, the formal definitions, the gauge discipline, and the first thermal-transport demonstration. It is carried from the standalone architecture reference essentially verbatim; the implemented kernel that supersedes its content-identity and store sketches is Part IV.

17 Principles

The operator layer is built on a small set of commitments. They are stated here without their alternatives or justifications; supporting definitions follow in later sections.

1. **Two worlds, one bridge.** The operator layer keeps the operator physics world (typed states, operator operations, approximation errors) strictly separate from the numeric world (discretized arrays, computed values, discretization errors). They are connected by exactly one bridge: the representation functor.

2. **Operator states are unit-free.** Dimensionful quantities (frequency, energy, length, time) exist at the operator layer, but specific unit choices (THz vs eV vs rad/s, Å vs Bohr) do not. Unit choice belongs to the representation and is declared by the adapter that produced it; the framework normalizes to a canonical unit before any cross-representation comparison.

3. **The atom is an operator operation.** The atomic unit of meaning is a typed transformation between operator states, not an instruction in an input file or an algorithm. Every operation carries a *operator formula*—a sympy expression (closed-form output) or sympy.Eq (implicit equation defining the output)—stating what it computes. The formula is the operator claim, machine-readable and Lean-projectable, against which adapter conformance can be checked.

4. **States are typed witnesses; nodes are ObservableSpaces or HiddenSpaces.** A state is the typed claim that an operator physics object exists with a particular history. The operator layer splits states into two structural kinds: *ObservableSpaces* (gauge-invariant, first-class, required to agree across adapters) and *HiddenSpaces* (gauge-dependent intermediate scaffolding whose per-element values reflect basis / phase / BZ-summation choices the framework does not pin down). Each state declares fields, and each field has an index signature (e.g. $\omega(\mathbf{q}, \nu)$ has indices $(\mathbf{q}, \nu)$; $\kappa^{\alpha\beta}$ has (α, β)). Numerical content lives in the representation, separate from the witness.

5. **Workflows are directed acyclic graphs.** A workflow is a graph of states connected by operations, not a linear sequence. Cross-code comparison reduces to graph alignment over the same operator template.

6. **Identity is content, not history.** A operator state is identified by its *content*: the quantity tag plus the type content (its field signatures, meaning each field's dimension and index kinds) plus its gauge class plus its labels. Provenance refers to a quantity instance's `source.ref`; a lineage is the shareable run record. Neither is part of a node's identity: production routes are recorded but a route never re-mints a node, so adding an alternative derivation of an existing quantity leaves the node's id unchanged. The implemented form of this rule, with the collision rationale that forced the quantity tag, is Part IV.

7. **Operation identity is parameterized.** An operation is identified by its name together with the parameters that affect physics. Different physics-affecting parameter values produce different operations and therefore different output states. Function-family parameterizations (e.g., Gaussian broadening by standard deviation vs. by half-width) are canonicalized at the operator layer to a single choice; adapters translate to their internal convention.

8. **Discretization belongs to representation.** Mesh densities, supercell sizes, displacement steps, and broadenings live on the representation, not on the operator state. Convergence is a relationship between an operator state and a family of its representations.

9. **Cross-code agreement is required at ObservableSpace nodes only.** Two adapters' representations of an ObservableSpace must agree (after unit and normalization conversion) to within the observable's declared tolerance. Representations of a HiddenSpace are

not cross-code comparable per-element; the operator layer's `compare` returns the status `NOT_COMPARABLE` and reports residuals as diagnostic information only. To make a HiddenSpace cross-code comparable, contract it into an ObservableSpace (e.g., per-mode $\Gamma_{q\nu}$ is a HiddenSpace; $\sum_{q\nu} \Gamma$ contracted via the BZ sum is an ObservableSpace that does agree).

10. **Errors are designed-for.** Approximation errors live on operations; discretization errors live on representations. The type system reserves slots for both; operator composition of error formulas is a Phase 2 capability.

11. **The operator root is the potential.** Every representation in the framework ultimately traces back to the Born–Oppenheimer potential of the material. Force constants of every order are derived states obtained by extracting derivatives of the potential and truncating the Taylor expansion. The upstream is stubbed in Phase 1.

12. **Sources are produced by nullary operations.** Source states (the potential, but also temperature, lattice constant, etc.) are the outputs of nullary `provide_*` operations rather than nodes with empty provenance. This unifies the DAG: every node is the output of some operation, and the operator layer has no special "input" category. The representation of a source state carries the externally supplied value (e.g., $T = 300\,\mathrm{K}$); the operation is just a structural tag.

13. **Lean-compatible by structure.** The operator layer is implemented in Python but designed so that its types transliterate cleanly into Lean. Three disciplines protect this option: closed unions for physics types, no Python-encoded physics invariants, operation parameters always recorded in output state metadata.

18 Background and Motivation

Modern computational materials science has many codes, many workflows, and many opinions about how to compose them. Despite a generation of effort on workflow managers (AiiDA, Atomate, FireWorks, Atomate2, Aflow), one quality is still missing: a workflow should expose, as typed and composable data, *the physics it computes*.

Today the standard unit of shared meaning between codes is the *input file*: an ordered sequence of instructions that, together with a particular code's internal logic, produces an output. This unit is opaque to two questions:

- **Given a result, what physical assumptions and approximations does it embed?** An input file declares parameters but not their meaning. A k-mesh is just a triple of integers; the relationship between that mesh and the convergence of the integrated observable is not part of the file.

- **Given two results from two codes, where exactly do they diverge?** Identical force constants run through kaldo and through ShengBTE typically yield slightly different thermal conductivities. The difference is not localized to a particular operation, even though—in principle—it is the result of distinct computational choices at specific steps.

A previous attempt of ours, `MaterialsCodeGraph` (MCG), addressed parts of this gap by treating each instruction in an input file as the atomic unit of a knowledge graph and tracking inputs, parameters, and outputs at that level. MCG's tool-encapsulation principle (all tool knowledge in YAML/Markdown configs, all code generic over tool) was the right design at that level of abstraction. But the level of abstraction itself is too low: an instruction in an input file

is a syntactic unit, not a semantic one. It tells you what a code is told to do, not what *physics* the code thereby computes. Two codes that compute the phonon dispersion via diagonalization of the dynamical matrix express that operation as different instructions; yet they perform the same physics. A graph indexed by instructions sees them as two unrelated tasks.

The operator layer proposed here *starts from the physics* and treats input files as derived artifacts. The atomic unit is no longer an instruction; it is a *operator operation* between typed physics states. The states are the load-bearing entities: a state is what changes when an operation is applied, and a workflow is the graph that records those state transitions. Concrete numerical computation is then a separate, typed projection of the operator workflow into a discretization scheme.

This is a deliberate departure from the input-file paradigm. We argue below that it pays for the extra abstraction with three concrete returns: cross-code reconciliation at canonical intermediate states, derived (rather than empirical) error bounds on computed observables, and a clean foundation for future formal verification.

19 Architectural Principle: Two Worlds, One Bridge

The single most important design decision is a strict separation between two type universes:

- **The operator world** holds typed physics states (Hamiltonians, dispersion relations, scattering rates, distribution functions, partition functions) and operator operations between them. Errors here are *approximation errors*—the $O(\cdot)$ terms introduced by truncating perturbation series, linearizing equations, or imposing physical assumptions such as the Born–Oppenheimer or harmonic approximation. The operator world contains no numerical content. Two operator states that differ only in their numerical estimates are the same operator state; two operator states that differ in the chain of approximations leading to them are different operator states.

- **The numeric world** holds discretized realizations of operator states: arrays sampled on finite grids, mode-resolved quantities at finite k-meshes, distributions evaluated at finite temperature. Errors here are *discretization errors*—the $O((1/N)^k)$ or $O(h^k)$ terms introduced by sampling a continuum quantity on a finite scheme. The numeric world contains no operator content; it does not know what theory the numbers are estimates of.

The two worlds are connected by exactly one bridge, the *representation functor*, which takes (i) an operator state and (ii) a discretization scheme and produces a numeric representation. Approximation errors compose in the operator world along provenance chains; discretization errors compose in the numeric world along execution graphs; the two combine only at representation time, where the total error of a numerical result decomposes into its operator and numeric contributions.

This separation is load-bearing. It is also somewhat unusual. Most computational physics codes mix the two worlds: a "Hamiltonian" object in such a code is sometimes a typed mathematical operator and sometimes a 4-D `numpy` array. The operator layer refuses to do this. If you are holding a Hamiltonian, you know which world you are in and you cannot accidentally mix the two. This refusal is what makes errors composable, theorems statable, and cross-code comparisons well-defined.

19.1 The full chain: two layers plus the external codes

In implementation terms, the project has *exactly two layers* under its own control plus the external physics codes that produce numerical data. Nothing else is a layer: tests, visualization, and experiment scripts are supporting infrastructure.

Layer 1 — operator. Types, declarations, formulas, gauges. Machinery in `omai.operator` (the `Space` and `Operator` types, `Field`, `Dimension`, `GaugeAction`, `SymmetryGroup`, `validate_dag`); domain instances in `omai.thermal_transport.operator` (the 54-node, 55-edge DAG with sympy formulas on each edge, gauge actions). No values, no units, no per-code knowledge.

Layer 2 — representation. Per-code declarations plus the runtime that operates on actual code outputs. Machinery in `omai.representation` (the `SpaceRepresentationSpec` and `OperatorRepresentationSpec` types, `Unit` and unit conversions, the `Representation` runtime data class, `compare()` with its five-status verdicts); domain instances in `omai.thermal_transport.representa` (one file per code: `kaldo.py`, `phono3py.py`, `phonopy.py`, `shengbte.py`, `qe.py`, `ase.py`, `lammps.py`, `gpumd.py`).

External codes. kaldo, phono3py, ShengBTE: do the actual physics computation in their own native code, output arrays in their native units and normalizations.

The chain in motion. A typical cross-code verification flows through these layers like this:

1. User invokes the codes externally (e.g. kaldo and phono3py), which produce raw output arrays in their own formats.
2. User wraps each array via `represent(spec, field_name, array)`: this validates the field exists on the operator state and produces a `Representation` object tagged with the right adapter spec.
3. User calls `compare(m_a, m_b)`. `compare` consults the operator layer for the cross-code factor (units × normalizations), applies it, optionally contracts, and consults the operator layer again for the space's gauge kind to decide whether an AGREE/DISAGREE verdict is even meaningful or the comparison is NOT_COMPARABLE.
4. User gets a `RepresentationComparisonResult` with status, residuals, and the applied factor.

No additional layers, no other indirection. The two layers expose their content in matched pairs (machinery in `omai/<layer>/`, domain instances in `omai/<domain>/<layer>/`), and the codes attach at the edge.

20 Formal Definitions

We collect the formal definitions implied by the decisions above.

20.1 Operator state

A operator state s is unit-free: its physics type carries the dimension of any quantity it represents (a frequency, an energy density, a tensor of given rank), but no unit choice. Numerical content and units appear only at representation time.

A operator state s is a tuple (κ, τ, π, F) where:

- $\kappa \in \{\text{OBSERVABLESPACE}, \text{HIDDENSPACE}\}$ is the *gauge-invariance kind*: ObservableSpaces are gauge-invariant and cross-code comparable; HiddenSpaces carry gauge freedom (eigenvector phase, BZ-summation choice, degenerate-subspace rotation, ...) that the operator layer doesn't pin down, so their per-element values differ across adapters by design;

- τ is a *physics type* drawn from a finite registry. Derived types in the thermal-transport scope include `ForceConstants[order=`n`]`, `DynamicalMatrix`, `Frequency`, `Eigenvectors`, `GroupVelocity`, `HeatCapacity`, `Linewidth`, `MeanFreeDisplacement`, `ThermalConductivity`; sources include `Potential`, `Temperature`. Some types are further parameterized by upstream operation choices (e.g., `ThermalConductivity[bte_solver= rta]` is a HiddenSpace since RTA's $1/\Gamma$ non-linearity breaks gauge invariance, whereas `ThermalConductivity[bte_solver=direc` is an ObservableSpace because the full LBTE preserves it);

- π is the *provenance*: an ordered list $[\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n]$ of operator operations whose composition produced s. Provenance is never empty—source states have a single nullary operation (`provide_potential`, `provide_temperature`, ...);

- $F = (f_1, f_2, \ldots)$ is the state's tuple of *fields*. Each field has a name, a dimension (frequency, energy, length, ...), and an index signature (e.g., ω has indices $(\mathbf{q}, \nu)$; v has $(\alpha, \mathbf{q}, \nu)$; κ has (α, β)).

Two operator states are equal iff their content ids match, where the content id is the hash of the quantity tag, the field signatures (each field's dimension and index kinds), the gauge class, and the labels (Part IV); names and provenance are not part of it. The composed approximation error on s (a Phase 2 capability) would be

$$\epsilon_{\text{approx}}(s) = \text{compose}\big(\epsilon_{\mathcal{O}_1},\, \epsilon_{\mathcal{O}_2},\, \ldots,\, \epsilon_{\mathcal{O}_n}\big). \tag{1}$$

Source versus derived states. Operator states fall into two kinds, both produced by some operation:

- *Source states* are the outputs of *nullary* operations (`provide_potential`, `provide_temperature`, ...): operations with no operator-state inputs. They are empirical inputs (lattice constant, temperature, pressure, applied field), the operator `Potential` itself (Phase 1 stubs the upstream of the potential), or measured observables (a thermal conductivity reported from a TDTR experiment, a Raman peak). The representation of a source state carries the externally supplied value (e.g., 5.43 Å for the lattice constant of silicon, or $T = 300\,\text{K}$); its discretization scheme is trivial (a singleton choice rather than a refinable mesh) but its error is still meaningful (experimental uncertainty, or definitional precision).

- *Derived states* are the outputs of operations that consume one or more upstream states. Examples: `ForceConstants[order=`n`]` (derived from `Potential` via `compute_force_constants[order=`n`]`, currently stubbed), `Frequency` and `Eigenvectors` (multi-output of `compute_dispersion`), `Linewidth`, `MeanFreeDisplacement`, the computed κ.

Both kinds follow the same machinery for representation. The unification of source and derived states under the same "output of an operation" rule answers several otherwise awkward cases: empirical inputs ("where does the lattice constant live?"), experimental observables ("how does a measured $\kappa(T)$ enter the framework?"), and the unmodeled operator root in Phase 1 ("what is the silicon potential, before we differentiate it?"). All three are representations of source states whose provenance is a single nullary operation; the second is what makes the unified theory–experiment comparison framework native rather than bolted on; the third is how Phase 1 stubs the upstream of `Potential` without yet modeling it in detail.

What "witness-as-state" means, concretely. The phrase *witness-as-state* comes from the Curry–Howard correspondence in type theory: a typed value is read as evidence (a "witness") for a proposition. We borrow the intuition without borrowing the formal proof obligations. In this framework a state is not the data it contains; it is the *claim* that something exists, expressed in a typed form rich enough to carry meaning.

Concretely, three layers are conflated in most computational physics codes; the operator layer keeps them apart:

1. **The proposition** – "the phonon dispersion of crystalline silicon under the harmonic approximation, derived by truncating the Born–Oppenheimer Taylor expansion at second order, exists as a function $\omega(\mathbf{q}, \nu)$ from the Brillouin zone to positive reals."

2. **The witness** – a typed object `s = (Dispersion, π)` that records precisely this claim. The physics type `Dispersion` encodes the kind of object being asserted to exist; the provenance π records the chain of operations (extraction, truncation, Fourier transform, eigendecomposition) that produced it. The witness carries no numerical content; it is the assertion that such an object exists with this history.

3. **The representation** – a tuple $(s, \Sigma, x, \epsilon_{\text{disc}})$ in which x is a concrete 4-D `numpy` array of frequencies sampled on a chosen k-mesh, Σ is the discretization scheme, and ϵ_{disc} is the associated discretization error. The representation is the numerical evidence that the witness asserts to exist.

A code that conflates these layers (treats the dispersion *as* the array) cannot distinguish "the same physics computed two different ways" from "different physics computed the same way." A framework that keeps them apart can: two witnesses with identical types but different provenance are different states, regardless of whether their representations happen to have similar numerical content; one witness with two representations on different k-meshes is the same state, regardless of how different the arrays look numerically.

This is the load-bearing distinction. It is also the answer to several otherwise hard questions. "What is the dispersion of silicon?" is the question the witness answers. "What does the dispersion of silicon look like on a $14 \times 14 \times 14$ mesh in kaldo" is the question its representation answers. "Are these two computed dispersions the same?" resolves into two sub-questions—are the witnesses equal (a question about provenance), and are the representations consistent within their discretization errors (a question about numerics)—each answerable without confusion with the other.

Read in this light: the operator states described elsewhere in this document are all witnesses: they assert the existence of a particular physics object derived in a particular way. The numerical content, when present, lives in the representation, separated from the witness by the operator/representation boundary. Operations between operator states are operations between witnesses, with no numerics involved; they are essentially structural moves on a graph of typed assertions.

20.2 Operator operation

A operator operation $\mathcal{O}$ is a typed map between operator states,

$$\mathcal{O} : \tau_{\text{in}_1} \times \tau_{\text{in}_2} \times \cdots \to (\tau_{\text{out}_1}, \ldots, \tau_{\text{out}_m}), \tag{2}$$

equipped with three pieces of metadata:

- a *operator formula* $\mathcal{F}_{\mathcal{O}}$ stating what the operation computes, encoded as a sympy expression for closed-form ops or a sympy.Eq for implicit ones (eigenvalue equations, linear systems). *Every* operation carries a formula—source operations have identity formulas (e.g., `provide_temperature`: $T = T_{\text{provided}}$);

- an *approximation error formula* $\epsilon_{\mathcal{O}}$, an operator expression in physical small parameters (e.g., λ for the strength of a perturbation, T/T^* for a Sommerfeld expansion). Exact operations have $\epsilon_{\mathcal{O}} = 0$; approximating operations have $\epsilon_{\mathcal{O}} = O(p^k)$ for some parameter p and order k;

- a possibly empty set of *schemes* (per §20.2) and *parameters* (dimensioned but unit-free at the operator layer).

Multi-output operators (e.g. `compute_dispersion` producing `Frequency` and `Eigenvectors` together) are first-class. The formula $\mathcal{F}_{\mathcal{O}}$ is the operator layer's machine-readable claim about what the operator produces: adapter conformance can be expressed as "the adapter's runtime behavior matches $\mathcal{F}_{\mathcal{O}}$ under its declared unit, normalization, and scheme choices," in principle automatable rather than reverse-engineered from kernels.

Operation identity is parameterized. Several operations in the operator layer carry method choices that affect physics: the scattering processes included (3-phonon only, or 3-phonon plus isotopic, or with boundary scattering), the Boltzmann solver (relaxation-time approximation, iterative, direct inversion), the broadening scheme (Gaussian, Lorentzian, adaptive), and so on. The operator layer treats these as part of the operation's *identity*, not as runtime configuration that gets discarded.

Concretely, an operation is identified by the tuple (operation name, parameters affecting physics). Two invocations of `compute_scattering_rates` with different `scattering_processes` are different operations; they appear differently in the provenance, and they produce operator states that are formally distinct (different type metadata, different total approximation error). The *name* `compute_scattering_rates` is the same; the *parameterized identity* differs.

This convention has three consequences. First, the vocabulary of operations remains small (one entry per named transformation, not one entry per parameter combination), but the provenance remains expressive (because provenance records the full parameterized identity). Second, output states' types are mildly dependent: `ScatteringRates` is parameterized by the scattering-processes choice, even though the operator layer does not require Python's type system to enforce this dependence. Third, the operator layer is structurally compatible with dependent type theory: when transliterated to a system like Lean, the parameter values lift to type-level indices and the dependence becomes formal.

20.3 Operator workflow

A operator workflow is a finite directed acyclic graph $W = (V, E)$ where V is a set of operator states and E is a set of operator operations. The graph is rooted at one or more *source* states (typically empirical inputs such as the crystal structure) and terminates at one or more *observable* states (e.g., the lattice thermal conductivity).

20.4 DAG extension rules

When a new physical quantity or new way of computing an existing quantity must be added to the DAG, three patterns are available. The choice has structural consequences, so we record the rules here rather than leaving them to taste.

Where formulas live. *Edges* carry the sympy formula and the declared schemes; *spaces* are typed places (a claim that "this physical quantity exists, with this dimension, this gauge type, this label vocabulary"). Different production formulas do not, by themselves, force different spaces — they force different edges.

Pattern A — label on the space. Use when the label changes the *gauge type* (ObservableSpace vs. HiddenSpace) of the produced quantity, *and* the labelled space is terminal (or its labelled consumers are themselves a closed sub-branch). Existing examples:

- `MeanFreeDisplacement[bte_solver=rta|direct_inverse]` — RTA is HiddenSpace, direct is ObservableSpace. Propagates to `ThermalConductivity[bte_solver=...]` and stops there.
- `ThermalConductivity[transport_model=lbte|wigner|qhgk]` — adds the orthogonal axis for Wigner and QHGK transport. All terminal.

This pattern is safe *only* for terminal or near-terminal nodes. Putting a type parameter on an intermediate state forces every downstream consumer to be parameterised in turn, polluting the entire downstream sub-DAG.

Pattern B — sibling states converging through an explicit edge. Use when several variants represent physically distinct contributions, with different inputs, that must combine before a downstream consumer uses them. The variants are *sibling named states*; a converging edge produces the combined "total" state, and the downstream sees only the total. Existing example:

- `AnharmonicLinewidth` (H, inputs FC^3, e, ω, T), `IsotopicLinewidth` (H, inputs IsotopeAbundances, e, ω), `BoundaryLinewidth` (H, inputs v, length scale) all flow into `TotalLinewidth` via the `sum_linewidths` edge. `solve_bte_*` consumes only `TotalLinewidth`.

The advantage over Pattern A here is that the per-channel inputs differ; sibling states make that visible in the DAG diagram and let adapters declare the channels they support independently.

Pattern C — shared output node, alternative producing edges. Use when several formulas produce the *same-typed* output with the same gauge classification, but from different inputs. The downstream is unaware of which path was taken; provenance and adapter specs record it. A literal in-place modifier — a single state with an edge that consumes and produces itself — would create a cycle and is rejected by the DAG validator. The canonical workaround is to introduce a small intermediate node that names the pre-modification version. Existing example:

- `compute_dynamical_matrix(FC²)` → `BareDynamicalMatrix`, followed by either `apply_nac_correction(BornCharges, ε∞)` → `DynamicalMatrix` (polar branch) or the identity edge `identity_dm(BareDM)` → `DynamicalMatrix` (non-polar branch). `compute_dispersion` consumes only `DynamicalMatrix` and does not need to know which producing edge fired.

Decision flow. When adding a variant of an existing quantity:

1. Does the variant change the gauge type? *Yes:* Pattern A.
2. Does the variant have different inputs that physically combine? *Yes:* Pattern B.
3. Same gauge type, same output type, alternative production? *Yes:* Pattern C.

In all three patterns, *edges carry the formulas* and *states carry the typed places*; the patterns differ only in which nodes are shared and which are distinct.

Learned shortcut edges. A machine-learned surrogate, seen from the map, is Pattern C with a precise extra claim: the edge approximates the *composite of a declared path of exact edges*, usually while amortizing away that path's most expensive boundary input. The kernel makes the claim first-class (`omai/operator/learned.py`): a `LearnedOperator` names the exact edges it shortcuts, and `validate_learned` checks the objective part in the spirit of the contribution gates. Its outputs must equal the shortcut path's terminal outputs, exactly; its

inputs may read nothing the path produces, nor anything downstream of the path's outputs (a shortcut may not create a cycle); scheme entries of the path's terminal edges are inherited verbatim, because a surrogate trained on labels computed under scheme S is a surrogate of the path under scheme S; and provenance is mandatory, a content-addressed `model_ref` for the trained artifact (retraining mints a new reference, hence a new claim) plus `trained_on` citations for the label sources, the same citation discipline as evidence. A learned edge is never authoritative: it is not sympy-executable (a formula, when present, is a structural ansatz, never a definition), it cannot settle cross-code agreement at an observable, and its predictions enter the record only as evidence tagged with the `model_ref`, comparable against the exact path wherever both exist. `amortized_inputs` computes what the shortcut buys. The motivating case is the third-order wall of anharmonic transport: the exact three-phonon linewidth consumes `ForceConstants[order=3]`, and a surrogate producing the same `Linewidth` node from harmonic geometry alone amortizes exactly that input into weights. No learned edge sits on the exported map yet: the first declaration should land together with a released model artifact and its evidence instances, and whether a surrogate is any *good* is settled by that evidence next to the exact path's values, never by the gates.

Two-tier validation. Validation runs at two layers. *Declarations* are checked at module load: `validate_dag` on `NODES` and `EDGES` verifies the gauge-discipline invariants (every HiddenSpace declares its gauge group and gauge-invariant contractions; scheme names referenced by adapters exist on their operators; no cycles), and now also the sympy-layer invariants (every edge's `formula.free_symbols` are derivable from its inputs and parameters; LHS indices match the output space's field; auxiliary formulas close over the main formula's vocabulary). *Cross-code data* is checked at comparison time: `compare_operators` on a pair of `SpaceRepresentationSpec`s verifies that two codes are claiming the same operator (compatible units, canonicalisable normalizations), and `compare_representations` on a pair of `Representation`s applies the spec-derived conversion and emits the five-status `RepresentationComparisonRes` Mismatches at the declaration layer are caught before any code runs; mismatches at the data layer are caught at the operator↔representation boundary, never silently.

20.5 Discretization scheme

A discretization scheme Σ is a finite tuple of parameters $(N_1, \ldots; h_1, \ldots)$ specifying how a continuum quantity is sampled. Examples include the k-point mesh density, the supercell size, the finite-difference displacement step, the broadening width, and the integration order. The scheme also carries an error model $\epsilon_\Sigma(N_1, \ldots; h_1, \ldots)$ giving the asymptotic dependence of the discretization error on these parameters.

20.6 Representation

A representation m is a runtime data class wrapping one adapter's emission of one field on one space:

$$m = (\text{SpaceRepresentationSpec}, \texttt{field\_name}, x)$$

where SpaceRepresentationSpec carries the operator `Space` s, the representation name, and the adapter's declarations (the *unit* of each field and the *normalization* of each field, plus notes); `field_name` selects which field of s is represented; and x is a concrete `numpy.ndarray`. The discretization scheme Σ and discretization error ϵ_{disc} are deferred to Phase 2.

Why the representation spec ships with the data: unit × normalization. Different codes express the same physics under two independent multiplicative choices, and the spec records both:

- **Unit** (measure choice). kaldo's linewidth is in angular-frequency THz while phono3py's is in linear-frequency THz; kaldo's heat capacity is in J/K, phono3py's in eV/K. Each `Unit` entry carries a `to_operator` multiplier into the operator-canonical unit for its dimension.
- **Normalization** (definitional choice). kaldo emits $\Gamma = 2\,\mathrm{Im}\,\Sigma$ while phono3py emits $\Gamma = \mathrm{Im}\,\Sigma$; ShengBTE reads FC3 in the mixed-dimension form $\mathrm{eV}/(\mathrm{Å}^2 \cdot \mathrm{nm})$ while kaldo / phono3py read $\mathrm{eV}/\mathrm{Å}^3$. Each `Normalization` entry carries a `to_operator` multiplier into the canonical definitional form.

The two choices are orthogonal: a raw array without its spec is unsafe to compare against another; with the spec attached, the operator layer computes both factors mechanically and surfaces unit / normalization mismatches as type-level declarations rather than as numerical mysteries.

Star topology: the operator layer is the unique hub. The cross-representation conversion factor is *never* a direct representation-to-representation primitive. The two primitives are `representation_to_operator`(A, obs) and its inverse `operator_to_representation`(A, obs), each defined as a clean composition of the unit's and normalization's `to_operator` multipliers:

$$\texttt{representation\_to\_operator}(A, \mathrm{obs}) \;=\; \texttt{UNITS}[u_A].\texttt{to\_operator} \cdot \texttt{NORMALIZATIONS}[n_A].\texttt{to\_operator}.$$

The cross-adapter A→B factor is the composition through the operator hub,

$$\texttt{operator\_to\_representation}(B, \mathrm{obs}) \;\cdot\; \texttt{representation\_to\_operator}(A, \mathrm{obs}),$$

derived and never primitive. Geometrically the operator layer is the unique hub of a star topology with each representation as a spoke; an N-representation domain requires N spec sets to the operator hub, never N^2 pairwise conversions. Lean-side this is the right factoring: the operator layer is the *theory*, each representation is a *model*, and any representation-to-representation morphism factors through the theory by construction.

The same discipline applies edge-by-edge in the executor: when the operator-side runtime walks the DAG and dispatches an edge to code A (i.e. to representation A), the value returned MUST be carried back through A's `representation_to_operator` before the next edge runs — whether that next edge is sympy-executable (closed-form on the operator layer) or dispatched to a different representation B (which then applies B's `operator_to_representation`). Operator-form intermediates are the lingua franca between heterogeneous edges; no edge-to-edge handshake bypasses the operator hub.

20.7 Cross-representation comparison: `compare`

The bridge from spec to verified empirical fact is the function

$$\texttt{compare}(m_a, m_b; \mathrm{rtol}, \mathrm{atol}, \mathrm{contraction}) \;\to\; \textsc{RepresentationComparisonResult}$$

which takes two representations of the same space and field, applies the cross-representation conversion factor derived from their declared units and normalizations, optionally contracts both arrays via a user-supplied callable (e.g. `numpy.sum`), and reports residuals together with a *status* drawn from five values:

Expected_Agree the space is an ObservableSpace; the arrays agree within $(\mathrm{rtol}, \mathrm{atol})$. The operator layer's prediction holds.

Expected_Disagree the user predicted disagreement (override) and the arrays disagree. Useful for intermediate contractions of a HiddenSpace that the user knows is only partially gauge-invariant.

Not_Comparable the space is a HiddenSpace and no contraction was supplied. The operator layer refuses to make an agree/disagree verdict; residuals are still computed and returned for diagnostic inspection, but they carry no normative weight.

Unexpected_Disagree the space is an ObservableSpace but the arrays disagree. *Real anomaly*: either the spec is missing a normalization, or the codes genuinely compute different physics, or rtol is too strict.

Unexpected_Agree the user predicted disagreement and the arrays agreed. Rare; the per-element protocol may be tighter than declared.

The status taxonomy separates three things that earlier versions of this framework conflated: the operator layer's *prediction* (would these agree?), the empirical *outcome* (did they?), and whether the comparison is even *meaningful* for the given state kind. Only UNEXPECTED_DISAGREE flags an anomaly that needs investigation; EXPECTED_AGREE and EXPECTED_DISAGREE are successful predictions; NOT_COMPARABLE is a HiddenSpace saying "don't ask this question."

20.8 The validation engine: execute, compose, cross-check

The representation layer carries a runtime that *runs* the operator DAG, not merely compares emitted arrays. `compute(target, sources)` lazily resolves a target space: a space with a registered source is loaded and lifted to operator form (units and normalizations applied automatically), and every other space is derived by executing its producing operator's sympy formula via `apply_edge`. Closed-form paths can also be fused symbolically (`compose_executable`) into one synthetic edge; the composed-then-executed value must equal the edge-by-edge value, so the symbolic composer and the numerical executor validate each other. Finally `cross_check` computes a target by several redundant routes (different codes per leaf, or different operator paths) and pairwise-compares them, with agree/disagree verdicts governed by the target's Observable/HiddenSpace typing: routes to an Observable must agree, routes through a HiddenSpace need not. On Si-Tersoff the engine derives molar heat capacity from frequencies and matches phonopy's emitted value to 0.1%, and contracts κ_{LBTE} from loaded group velocity, mean free displacement, and a derived heat capacity, matching kaldo's emitted conductivity.

The executor reconciles units dimensionally: each dimension's canonical unit carries an absolute SI scale, and when an operator's formula is a pure contraction (a monomial in its input fields and declared parameters), the executor rescales the raw canonical-unit result into the output's declared canonical unit automatically. This is what lets the κ_{LBTE} contraction of Å/THz-canonical group velocity and mean free displacement with an $\mathrm{Å}^3$ cell volume emerge directly in $\mathrm{W}/(\mathrm{m}\cdot\mathrm{K})$, with no hand-applied conversion. Closed-form edges (whose constants already carry the SI conversion) and additive edges are not monomials and are left untouched.

20.9 HiddenSpace discipline and gauge actions

The ObservableSpace / HiddenSpace distinction is load-bearing, so the operator layer enforces a discipline on how HiddenSpaces are declared. Each HiddenSpace carries three additional fields:

- `gauge_group`: a named identifier for the gauge equivalence acting on this state (e.g. "U(1)_phase_on_eigenve "bz_summation_permutation").
- `kind`: one of `scaffolding` (the HiddenSpace is consumed by a downstream operation producing an ObservableSpace, so the gauge orbit is eventually summed away) or `approximation` (the HiddenSpace is terminal — an approximation of an ObservableSpace that happens to break gauge invariance, with no downstream recovery; κ`[bte_solver=rta]` is the canonical example).

- `gauge_invariant_contractions`: names of ObservableSpaces in the DAG that capture this state's gauge-invariant content (empty for `approximation` kind).

A `validate_dag` function walks the node and edge sets at load time and rejects: undeclared gauge groups, invalid kinds, scaffolding HiddenSpaces with empty or non-resolving contractions, approximation HiddenSpaces that incorrectly declare contractions, and edges that reference unknown nodes. The thermal-transport DAG passes this validator; tests check it on every CI run.

Operator invariance via `GaugeAction`

Beyond named declarations, the operator layer supports *operator proofs* of gauge invariance for the tractable cases. A `GaugeAction` is a sympy-encoded transformation: a pattern to match (e.g., $e_{i,q,\nu}$) plus a transform (e.g., $e^{i\theta_{q,\nu}} \cdot e_{i,q,\nu}$). Given an operation's sympy formula, the operator layer substitutes the gauge action and runs `sympy.simplify` on the difference; if the result is zero, the formula is machine-verified invariant.

Example: applying the U(1) phase $e_{i,q,\nu} \to e^{i\theta_{q,\nu}} e_{i,q,\nu}$ to

$$v^{\alpha}_{q,\nu} = \frac{1}{2\omega_{q,\nu}} \sum_{i,j} e^{\dagger}_{i,q,\nu} \frac{\partial D_{ij}}{\partial q^{\alpha}} e_{j,q,\nu}$$

gives $e^{-i\theta_{q,\nu}} \cdot e^{i\theta_{q,\nu}} = 1$, and sympy mechanically reduces the difference to zero. This is the operator layer's first machine-verified physics claim — per-mode group velocity is U(1)-phase invariant, by construction of the formula.

What's tractable today: simple operator substitutions (U(1) phase), finite group actions encoded as patterns over indexed expressions (crystal point groups), and other gauges where the transformation is algebraically substitutable. *Not tractable today:* continuous Lie group actions on subspaces (e.g., U(d) on degenerate-subspace), and data-dependent gauges (e.g. degenerate-subspace rotation that only acts at degenerate ω). For those, the operator layer falls back to the Level 1 named-gauge declarations.

Crystal symmetry as an operator declaration. Crystal symmetry at the operator layer level is purely a *declaration*: a `SymmetryGroup` carries a Hermann-Mauguin / Schoenflies name (`Oh`, `D6h`, `Ci`, ...) and an order $|G|$. No element data — no rotation matrices, no translation vectors, no group-element enumeration — lives in the operator layer.

The concrete handling — extracting symmetry from a structure, applying group operations to FC tensors, building irreducible-BZ reductions — belongs to the materials codes (phonopy, kaldo, ShengBTE, ...), typically via spglib. The operator layer's role is just to *name* the group so that operations can declare which symmetry they assume as a parameterized identity (`compute_force_constants[order=2, symmetry=Oh]` differs from `[..., symmetry=C1]`), so adapter specs can declare which group a particular run assumed, and so cross-code comparison can refuse to compare two representations whose declared symmetry groups disagree.

This is the same pattern `Potential` follows: declared symbolically as a labelled source state without the operator layer modeling its functional form. "How $\Phi^{(2)}$ is computed under crystal symmetry" is a code concern; "which symmetry group was assumed" is an operator-level declaration.

20.10 Representation functor

A representation functor $\mathcal{F}$ is associated with a particular code (kaldo, phono3py, ShengBTE). Given an operator workflow W and a discretization scheme Σ,

$$\mathcal{F}(W, \Sigma) \tag{3}$$

produces a directed graph of representations whose vertices are in one-to-one correspondence with a subset $V_{\mathcal{F}} \subseteq V(W)$ and whose edges are the computational realizations of the operator operations restricted to $V_{\mathcal{F}}$. Different codes implement different functors over the same operator workflow; their represented graphs are aligned by their shared operator template.

20.11 Total error

The total error on the representation m of an operator state s is

$$\epsilon_{\text{total}}(m) = \text{compose}\big(\epsilon_{\text{approx}}(s),\, \epsilon_{\text{disc}}(m)\big). \tag{4}$$

The composition rule depends on the operation type but is in all cases derivable from the typed metadata. The two contributions $\epsilon_{\text{approx}}(s)$ and $\epsilon_{\text{disc}}(m)$ are recovered separately, so a result that is converged in discretization but uncertain in approximation is distinguishable from one that is fully approximated but undersampled.

21 Lean-compatibility disciplines

"Lean-compatible" is structural rather than syntactic. The operator layer's Python implementation will not be Lean code, but it should be *shaped* so that the Lean version is a transliteration rather than a redesign. We commit to three disciplines that protect this option.

1. **Closed unions for physics types.** The registry of physics types (`Potential`, `ForceConstants`, `Dispersion`, `ScatteringRates`, ...) is exhaustive and closed—implemented in Python via sealed enums or tagged unions, not via open class inheritance. Lean represents these as inductive types with a fixed list of constructors; closed unions in Python preserve this structure exactly. Open inheritance would require redesign at the Lean stage.

2. **No physics invariants encoded in Python types.** Properties such as "a force-constant tensor is symmetric under the appropriate index exchanges" or "the eigenvectors of a dynamical matrix are orthonormal" are documented as informal invariants on each type but *not* enforced by the Python type system. In Lean these invariants become formal proof obligations attached to the corresponding types. Trying to encode them in Python (via runtime assertions or pydantic validators) creates a layer of checks that does not transliterate and risks coupling the implementation to Python-specific idioms.

3. **Operation parameters always recorded in output state metadata.** When an operation carries parameters that affect physics (§20.2), Python's type system will not enforce the dependence of the output type on those parameters, but the output state's metadata must still record them. In Lean these parameters lift to type-level indices, and the type-level dependence is enforced formally; in Python they are runtime metadata that downstream operations and provenance machinery treat as part of the operation's identity.

These disciplines are inexpensive to follow from day one and prohibitive to retrofit. They keep the transliteration cost low without committing the operator layer to Lean tooling we do not yet need.

22 First Demonstration: Lattice Thermal Transport

We instantiate the operator layer on the canonical workflow for lattice thermal conductivity κ, computed from second- and third-order interatomic force constants via the linearized Boltzmann transport equation.

22.1 The operator DAG

The operator DAG has 54 nodes and 55 edges. Two source nodes (`Potential`, `Temperature`) are produced by nullary `provide_*` operations; the `MeanFreeDisplacement` and `ThermalConductivity` states are each split into two parameterized variants by the upstream `bte_solver` choice (one gauge-invariant, one not). A second tier of derived observables — `PhononDOS`, `Gruneisen`, `PhaseSpace3Phonon`, `VolumetricHeatCapacity`, `MolarHeatCapacity` — hangs off the harmonic chain to capture the quantities ShengBTE and phonopy emit directly. On top of the BTE chain, a parallel *MD-primitive tier* captures the time-resolved quantities a classical molecular-dynamics run produces: `Trajectory`, `HeatCurrent`, `HeatCurrentACF`, `VelocityAutocorrelation`, and `MeanSquaredDisplacement`; from this tier the three MD-based κ contractions (Green-Kubo from HeatCurrentACF, NEMD and HNEMD from time-averaged HeatCurrent) close the cross-paradigm κ map alongside the LBTE / Wigner / QHGK variants.

Nodes (ObservableSpace / HiddenSpace):

- `Potential` (**O**): the operator root, stubbed in Phase 1;
- `Temperature` (**O**): scalar source T;
- `ForceConstants[order=2]` (**O**), `[order=3]` (**O**): real-space interatomic force constants $\Phi^{(n)}$;
- `DynamicalMatrix` (**O**): Bloch sum $D_{ij}(\mathbf{q})$;
- `Frequency` (**O**): $\omega(\mathbf{q}, \nu)$, eigenvalues (gauge-invariant);
- `Eigenvectors` (**H**): $e(\mathbf{q}, \nu)$, phase + degenerate-subspace freedom;
- `GroupVelocity` (**H**): $v^\alpha(\mathbf{q}, \nu)$, inherits eigenvector freedom at degenerate ω;
- `HeatCapacity` (**O**): $c(\mathbf{q}, \nu, T) = (\hbar\omega)^2/(4k_BT^2 \sinh^2(\hbar\omega/2k_BT))$;
- `MeanFreeDisplacement[bte_solver=rta]` (**H**): RTA $F = v/(2\Gamma)$, inherits Linewidth's looseness;
- `MeanFreeDisplacement[bte_solver=direct_inverse]` (**O**): full LBTE solution, gauge-invariant;
- `ThermalConductivity[bte_solver=rta]` (**H**): the $1/\Gamma$ non-linearity propagates Linewidth's looseness into κ_{RTA};
- `ThermalConductivity[bte_solver=direct_inverse]` (**O**): LBTE off-diagonals preserve gauge-invariance.
- `VolumetricHeatCapacity` (**O**), `MolarHeatCapacity` (**O**): the two BZ-and-mode contractions of `HeatCapacity` that ShengBTE (volumetric) and phonopy (molar) expose directly;
- `PhononDOS` (**O**): histogram of $\omega(\mathbf{q}, \nu)$ — independent of eigenvectors, hence gauge-invariant;
- `Gruneisen` (**O**): mode $\gamma_{q,\nu}$ from FC2, FC3 and the harmonic eigensystem;
- `PhaseSpace3Phonon` (**O**): three-phonon kinematic availability $P_3(\mathbf{q}, \nu)$ — the bare counting underlying Γ before $|V_3|^2$ enters;
- `BareDynamicalMatrix` (**O**), `BornCharges` (**O**), `DielectricTensor` (**O**) — the NAC inputs and the intermediate pre-correction DM (Pattern C; both `identity_dm` and `apply_nac_correction` converge on `DynamicalMatrix`);
- `HelmholtzFreeEnergy` (**O**), `Entropy` (**O**), `InternalEnergy` (**O**), plus their `Molar*` contractions — sibling ObservableSpaces of `HeatCapacity`;

- `Linewidth[channel=anharmonic_3ph|isotope|boundary|total]` (**H**) — four channels with different inputs, summed via `sum_linewidths`;
- `IsotopeAbundances` (**O**) — source-tier per-atom mass-variance factor;
- `ThermalConductivity[transport_model=wigner]` (**O**) decomposed into `wigner_populations` (**O**) and `wigner_coherences` (**O**); `[transport_model=qhgk]` (**H**) — inherits Γ's gauge type;
- `CumulativeKappa[wrt=omega]` (**O**), `[wrt=mfp]` (**O**) — distribution observables derived from κ_{LBTE} ingredients.
- *MD-primitive tier (P2):* `Trajectory` (**H**) with fields r, v indexed (i, α, t) — gauge-dependent under MD ensemble noise; `HeatCurrent` (**H**) with $J(\alpha, t)$, the Irving–Kirkwood snapshot; `HeatCurrentACF` (**O**) with $J^{corr}(\alpha, \beta, \tau)$, the Green–Kubo integrand; `VelocityAutocorrelation` (**O**) with $C_v(\tau)$, whose cosine Fourier yields a phonon DOS via Wiener–Khinchin; `MeanSquaredDisplaceme` (**O**) with $M(\tau)$, the diffusion-limit observable.
- *MD-based κ terminals (P3):* three Pattern-A `transport_model` variants of `ThermalConductivity`, all ObservableSpaces — `[transport_model=green_kubo]` from the time-integrated heat-flux ACF; `[transport_model=nemd]` from the imposed-gradient/Müller-Plathe steady state; and `[transport_model=hnemd]` from the homogeneous-NEMD driving force. With `[lbte]` (rta + direct_inverse), `[wigner]` (populations + coherences + total), and `[qhgk]`, this closes the cross-paradigm κ map across BTE and MD.
- *Amorphous-branch per-mode diagnostics (kaldo delta scan):* alongside the QHGK terminal κ, the amorphous branch now carries the two per-mode quantities the QHGK-paper parse surfaced, `ParticipationRatio` (**O**, dimensionless, indexed (q, ν)) — the Bell/Dean $1/N$ localization ratio — and `ModalDiffusivity` (**O**, L^2T^{-1}, indexed (q, ν)), the QHGK / Allen–Feldman per-mode heat-mode diffusivity in mm^2/s, kept apart from the mass-transport `Diffusivity` by name and tag despite the shared dimension.
- *Nuclear-quantum-effects layer (Atomistic Cookbook audit, i-PI slice):* sampled off the same MD `Trajectory`, path-integral MD (i-PI, the sixth Trajectory producer) opens the nuclear-quantum layer with a genuinely new scalar `QuantumKineticEnergy` (**O**, ENERGY), the centroid-virial estimator that exceeds the classical $\frac{3}{2}Nk_BT$ equipartition value, and a method-tagged `HeatCapacity[method=pimd]` (**O**, ENERGY PER TEMPERATURE), the PIMD scaled-coordinates C_V estimator that joins the harmonic `HeatCapacity` family on the same tag and dimension, distinguished only by the `method=pimd` label (the carrier / transport_model precedent), a producer variant with no re-mint.

Edges. Verb-headed names; each carries a sympy formula:

- `provide_potential` (identity), `provide_temperature` (identity);
- `compute_force_constants[order=2]`, `[order=3]` (stubbed): $\Phi^{(n)} = \partial^n V / \partial u^n|_{u=0}$;
- `compute_dynamical_matrix`: Bloch sum $D_{ij}(\mathbf{q}) = \frac{1}{\sqrt{M_i M_j}} \sum_R \Phi^{(2)}_{ij}(R)\, e^{i\mathbf{q}\cdot R}$;
- `compute_dispersion` (multi-output): eigenvalue equation $\sum_j D_{ij}(\mathbf{q})\, e_{j,q,\nu} = \omega^2_{q,\nu}\, e_{i,q,\nu}$;
- `compute_group_velocity`: Hellmann–Feynman $v^\alpha_{q,\nu} = \frac{1}{2\omega_{q,\nu}} e^\dagger_{q,\nu}\, \partial D / \partial q^\alpha\, e_{q,\nu}$; scheme `gv_method` ∈ {`hellmann_feynman` (canonical), `finite_difference`} captures the two estimators codes use;
- `compute_heat_capacity`: closed form above;

- `compute_anharmonic_linewidth`: triple BZ sum (Fermi's golden rule); scheme `broadening_param` $\in$ {stdev, halfwidth, adaptive_scaled}, plus an auxiliary equation `auxiliary_formulas[0]` that spells out the Maradudin–Fein kernel $|V_3|^2 = |\sum_{ijk,RR'} \Phi^3_{ijk}\, e\, e\, e/\sqrt{m\, m\, m}|^2/(8\,\omega\,\omega'\,\omega'')$ — the same kernel reappears verbatim in the LBTE collision matrix Ξ (`solve_bte[direct_inverse].auxiliary_formulas[0]`);
- `solve_bte[bte_solver=rta]`: closed form $F = v/(2\Gamma)$;
- `solve_bte[bte_solver=direct_inverse]`: implicit linear system $\sum_{q'\nu'} \mathcal{M}_{q\nu,q'\nu'}\, F^\alpha_{q'\nu'} = c_{q\nu}\, v^\alpha_{q\nu}$;
- `contract_kappa[bte_solver=rta]` and `[bte_solver= direct_inverse]`: same BZ contraction $\kappa^{\alpha\beta} = \frac{1}{VN_q} \sum c\, v^\alpha\, F^\beta$; the two operations differ only in their input and output state types, ensuring κ_{RTA} is typed as a HiddenSpace and κ_{LBTE} as an ObservableSpace;
- `contract_volumetric_heat_capacity`, `contract_molar_heat_capacity`: BZ-and-mode sums of `HeatCapacity`, divided by cell volume or multiplied by N_A/N_q respectively, so adapters that emit only the contracted form (ShengBTE, phonopy) land on a shared operator state;
- `compute_dos`: $g(\omega) = N_q^{-1} \sum_{q\nu} \delta(\omega - \omega_{q\nu})$, with scheme `dos_broadening` $\in$ {`gaussian` (canonical), `tetrahedron`, `adaptive_scaled`};
- `compute_gruneisen`: Maradudin–Fein closed form, with scheme `gruneisen_method` $\in$ {`maradudin_fein` (canonical), `finite_difference`} — phonopy uses the latter (deformed-cell finite difference), the other three codes the former;
- `compute_phase_space_3phonon`: $P_3(\mathbf{q}, \nu)$ from energy + crystal-momentum conservation, with scheme `delta_broadening` mirroring `dos_broadening`;
- `provide_born_charges`, `provide_dielectric_tensor`, `provide_isotope_abundances` — nullary sources for the polar / isotope inputs;
- `identity_dm` and `apply_nac_correction`: Pattern-C siblings into `DynamicalMatrix` (scheme `nac_scheme` $\in$ {gonze_lee canonical, wang, ewald});
- `compute_free_energy`, `compute_entropy`, `compute_internal_energy` and their `contract_molar_*` counterparts — harmonic-oscillator closed forms with $x = \hbar\omega/(k_B T)$;
- `compute_isotope_scattering` (Tamura), `compute_boundary_scattering` (Casimir), and `sum_linewidths` (Matthiessen) — the sibling channels joining `compute_anharmonic_linewidth` above;
- `compute_kappa_wigner_populations`, `compute_kappa_wigner_coherences` (Lorentzian band-overlap), `combine_kappa_wigner`, `compute_kappa_qhgk`; the `compute_kappa_wigner_coherences` mode-pair weighting was corrected on 2026-07-07 to the frequency-weighted Simoncelli form $(\omega + \omega')/2 \cdot (c/\omega + c'/\omega')$, transcribed from the vendored phono3py SMM19 solver (`kappa_solvers.py:122--126`);
- `contract_cumulative_kappa[wrt=omega|mfp]` — Heaviside-θ encoded cumulative thresholds with `binning` scheme (linear for ω, log for MFP).
- *MD-primitive tier (P2):* `run_md` — Velocity-Verlet recurrence with `ensemble` $\in$ {nve, nvt, npt}, `thermostat` $\in$ {berendsen, langevin, nose_hoover, csvr, none}, `integrator` $\in$ {velocity_verlet, leapfrog}; `compute_heat_current` — Irving–Kirkwood / Hardy / virial decomposition (`definition` scheme); `autocorrelate_heat_current` — direct/FFT time correlation of $J(t)$ (numerically equivalent under periodic padding, so no `correlation_method` scheme is declared); `compute_velocity_autocorrelation` — direct/FFT $\langle v(0) \cdot v(\tau) \rangle$;

`compute_msd` — $\langle |r(t+\tau) - r(t)|^2 \rangle$, with `unwrap_pbc` scheme; `fourier_to_dos` — Wiener–Khinchin $g(\omega) = (1/\pi) \int C_v(\tau) \cos \omega\tau \, d\tau$, a Pattern-C alternative producer of `PhononDOS` alongside `compute_dos`.

- *MD-based κ contractions (P3):* `contract_kappa[transport_model=green_kubo]` — $\kappa^{\alpha\beta} = V/(k_B T^2) \int_0^{\tau_{max}} \langle J^\alpha(0) J^\beta(\tau) \rangle \, d\tau$; `contract_kappa[transport_model=nemd]` — $\kappa = -\langle J \rangle / (\partial T / \partial z)$, with `nemd_method` scheme $\in$ {`direct_two_ reservoir`, `muller_plathe`, `ehex`} (LAMMPS-side method); `contract_kappa[transport_model=hnemd]` — $\kappa^{\alpha\beta} = \langle J^\alpha \rangle / (T \, V \, F_e^\beta)$ in the linear-response limit (GPUMD-side method). Cross-paradigm κ comparison between BTE, equilibrium MD, and non-equilibrium MD is the cumulative reach of this set.

Edges are exact operator operations (Fourier transform, eigendecomposition, mode contraction) or approximating ones (truncation of the Taylor expansion at second or third order, the relaxation-time approximation when chosen). Each approximating edge carries an explicit error formula (Phase 2 capability). The Phase 1 demonstration exercises the path from `ForceConstants` downward; the upstream operations from `Potential` to `ForceConstants` are declared but executed only as opaque adapter-level steps recorded as provenance, not modeled by the framework.

22.2 Representation functors (codes)

Each adapter is a representation functor over the operator DAG. Fifteen adapter modules are now ingested in the thermal-transport domain alone (31 rails map the whole graph), in three tiers. The four BTE / harmonic codes that consume force constants:

- **kaldo** (Python). Computes harmonic and anharmonic phonon transport from force constants, supporting iterative BTE, RTA, and the Wigner formulation. Most intermediate states are exposed via Python attributes on the `Phonons` object. kaldo's `Conductivity(method='inverse'` realizes the canonical `bte_solver=direct_inverse`.

- **phono3py** (C/Python). Computes lattice thermal conductivity from third-order force constants. Intermediate quantities are exposed in HDF5 outputs. `run_thermal_conductivity(is_LBTE=True)` realizes the canonical `bte_solver=direct_inverse`.

- **phonopy** (C/Python). The harmonic-only sibling of phono3py: emits frequencies, group velocities, the DOS, the molar heat capacity, and (via `PhonopyGruneisen`) mode Grüneisen parameters from a finite-difference $\omega(V)$ rather than the canonical Maradudin–Fein closed form. No three-phonon scattering, so the adapter covers the harmonic half of the DAG only.

- **ShengBTE** (Fortran/MPI). Solves the full linearized BTE iteratively. The harmonic chain is delegated to phonopy/QE upstream; ShengBTE consumes FC2/FC3 files and writes `BTE.KappaTensorVsT_{RTA,CONV}`, `BTE.{omega, v, gruneisen, P3, dos}`, etc. Per-mode heat capacity is not exposed (only `BTE.cv`), so the adapter targets `VolumetricHeatCapacity` rather than `HeatCapacity` on that branch.

and three phase-2 additions — an ASE Potential anchor and two molecular-dynamics codes that realise the MD-primitive tier and the MD-based κ paths:

- **ase** (Python protocol). Not a thermal-transport code: it supplies only the shared *Potential* / force-evaluation anchor (`provide_potential` via `ase.Atoms.calc`). The four BTE codes above cite the `ase` adapter as their canonical Potential source, so a cross-code κ comparison rests on one declared force field rather than an implicit shared assumption.

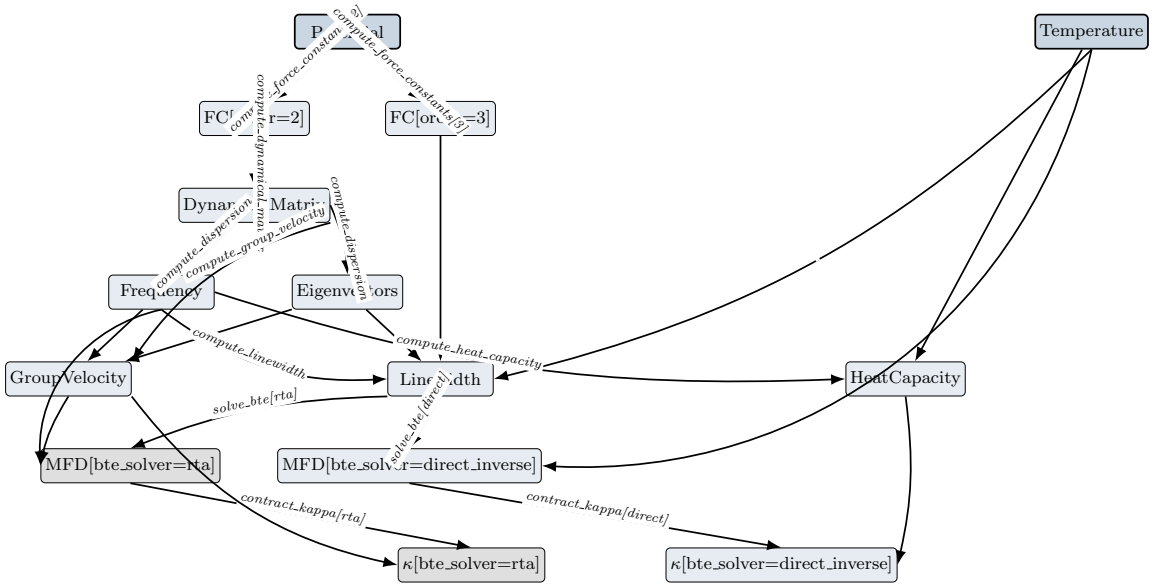

Figure 2: Core κ chain of the operator DAG for lattice thermal transport (12 nodes / 14 edges shown; the full DAG has 54 nodes / 55 edges once the derived-observable tier — PhononDOS, Grüneisen, PhaseSpace, VolumetricHeatCapacity, MolarHeatCapacity, the thermodynamic siblings (HelmholtzFreeEnergy, Entropy, InternalEnergy and their molar contractions), the polar-correction inputs (BareDynamicalMatrix, BornCharges, DielectricTensor), the per-channel Linewidth split, the Wigner/QHGK transport variants, the cumulative-κ distributions, the MD-primitive tier (Trajectory, HeatCurrent, HeatCurrentACF, VelocityAutocorrelation, MeanSquaredDisplacement), and the three MD-based κ Pattern-A terminals (Green-Kubo, NEMD, HNEMD) — is included; see the interactive map at `docs/map/` for the live view). The two darker top boxes (`Potential`, `Temperature`) are source nodes; the dashed edges from `Potential` represent the stubbed Phase-1 upstream. `MeanFreeDisplacement` and κ split into two parameterized variants by `bte_solver`: the gray-shaded `[rta]` variants are HiddenSpaces (RTA's $1/\Gamma$ non-linearity breaks gauge invariance), the unshaded `[direct_inverse]` variants are ObservableSpaces (the full LBTE preserves it). Each derived edge carries a sympy formula; sources carry identity formulas. All nodes are outputs of some operator; cross-code comparison happens at ObservableSpace nodes (after the operator layer's spec-derived unit and normalization conversion). Other HiddenSpaces (`Eigenvectors`, `GroupVelocity`, `Linewidth`) are not visually distinguished in this diagram but are gauge-dependent in the same sense.

- **LAMMPS** (C++/Python). Classical molecular dynamics. Native `pair_style` Potential, `Temperature` (global, region, and chunked-profile variants), plus the MD-primitive tier (`Trajectory`, `HeatCurrent`, `HeatCurrentACF`, `VelocityAutocorrelation`, `MeanSquaredDisplacement` via `compute heat/flux`, `compute vacf`, `compute msd`, `fix ave/correlate`) and the equilibrium Green-Kubo and Müller-Plathe NEMD κ paths. HNEMD is recorded as not-exposed (it routes through GPUMD).
- **GPUMD** (CUDA/C++). GPU-accelerated classical MD specialised for thermal transport. NEP-potential (neuro-evolution) anchor, the same MD-primitive tier, and the Green-Kubo (`compute_hac`) and HNEMD (`compute_hnemd`) κ paths — HNEMD being GPUMD's signature method; direct NEMD is recorded as not-exposed (routes through LAMMPS).

and one source-tier addition grounding the upstream the BTE codes consume as given:

- **Quantum ESPRESSO** (Fortran/MPI). First-principles DFT and DFPT. Grounds the source tier: `ForceConstants[order=2]` (`q2r.x` real-space constants in Ry/bohr^2), `BornCharges`

and `DielectricTensor` (`ph.x` linear response, turning the `provide_*` sources into computed quantities), `BareDynamicalMatrix` / `DynamicalMatrix` (raw un-mass-weighted $C(\mathbf{q})$ files; mass weighting applied only at diagonalization), `Frequency` (linear cm^{-1} / THz), `Eigenvectors`, and `PhononDOS` via `matdyn.x`. The adapter records the DFPT method (versus finite displacements), QE's internal space-group machinery, and the Gonze NAC scheme; derived from the anchored scan catalog `scans/qe-phonon.json`.

Each adapter spec declares per-space units, normalization overrides, and notes (`SpaceRepresentationSpec`) plus per-operator parameter units, scheme overrides, and discretization choices (`OperatorRepresentationSpec`). Differences from the operator layer's canonical declarations surface as conversion factors at spec-load time.

22.3 Verification on silicon (Tersoff potential)

The operator layer's predictions have been verified end-to-end against numerical runs of kaldo and phono3py on silicon at the Tersoff potential, $8 \times 8 \times 8$ q-mesh, $T = 300\,\mathrm{K}$, broadening stdev 0.1 THz:

ObservableSpace / contraction	Operator prediction	Empirical residual
`Frequency` per-mode (**O**)	factor 1 (both linear THz)	5×10^{-4}
`HeatCapacity` per-mode (**O**)	factor $1/e \approx 6 \times 10^{18}$	3×10^{-5}
$\sum_{q\nu} \Gamma$ contracted (**O**)	factor $1/(4\pi)$	5×10^{-4}
κ`[bte_solver=direct_inverse]` (**O**)[1]	factor 1	4×10^{-3}
`Linewidth` per-mode (**H**)	NOT_COMPARABLE	25% (diagnostic only)
κ`[bte_solver=rta]` (**H**)	NOT_COMPARABLE	4% (diagnostic only)

All ObservableSpace comparisons pass at $\leq 1\%$ tolerance; all HiddenSpace per-element comparisons return NOT_COMPARABLE. The $1/(4\pi)$ factor on Γ decomposes into a $1/(2\pi)$ *unit* factor (kaldo angular vs phono3py linear THz) and a 1/2 *normalization* factor (kaldo $\Gamma = 2\,\mathrm{Im}\,\Sigma \Rightarrow$ `linewidth_2x_imag_self_energy` with `to_operator` $= 0.5$; phono3py uses the canonical normalization). Both factors were derived by the operator layer from declared adapter specs, then applied mechanically to the diagnostic data.

The 4% disagreement on κ_{RTA} is the operator layer's structural prediction working as designed: RTA's approximation breaks κ's gauge invariance via the $1/\Gamma$ non-linearity, so the per-mode Linewidth redistribution propagates into κ. Typing κ_{RTA} as a HiddenSpace makes this a non-anomaly; the gauge-invariant κ_{LBTE} agrees to 0.4%.

Cross-paradigm κ (phase 2 P4). On the same Si-Tersoff worked example, the LAMMPS Green-Kubo path (`contract_kappa[transport_model=green_kubo]` reached through the P2 MD-primitive tier and the P3 contraction edge) returns a κ value that, in the small ensemble of equilibrium-MD runs the framework drives, is expected to land within $\sim$ 20–30% of κ_{LBTE}. That tolerance is Green-Kubo's empirical signal-to-noise floor on a few-hundred-atom supercell, not a framework concern. The driver at `experiments/silicon_tersoff/run_lammps_gk.py` generates the input script and the supercell data file regardless of whether `lmp` is on `PATH`; the `spec_demo.py` cross-paradigm audit section and the corresponding `test_silicon_consolidation.py` smoke test gracefully skip when the resulting `kappa_lammps_gk.npy` is absent. The NEMD and HNEMD analogues land in P5 and P6.

[1] This 0.4% residual and the $\sim$ 10% figure that the map's published instances and the end-to-end cross-code experiments (`experiments/silicon_tersoff`, the sigma sweep) show for kaldo versus phono3py at $8 \times 8 \times 8$ with independently computed force constants are not in contradiction: they describe different comparisons. The $\sim$ 10% number is what two codes give when each computes its own force constants; the 0.4% figure comes from a more controlled comparison whose exact setup is being re-established. Until that setup is re-pinned, treat the two numbers as measuring different things.

22.4 Phase 1 scientific capabilities

Phase 1 yields two immediate scientific capabilities, each of which is an artifact the field does not currently produce. A third capability is the long-term goal toward which the operator layer is oriented but is deferred to Phase 2.

- **Cross-code reconciliation at canonical intermediate states.** When kaldo and ShengBTE disagree on a final κ, the operator layer localizes the divergence to the first operator state at which the representations stop agreeing, and attributes it to a specific operation along the provenance. This is a structural diagnosis: the framework identifies where two codes do something different, regardless of the magnitude of the difference.
- **A unified framework for theory-experiment comparison.** The same operator DAG accepts experimental measurements as additional evidence on the same nodes, with the same alignment machinery as for cross-code comparison. The two Glassbrenner–Slack 1964 steady-state thermal conductivities of bulk single-crystal silicon and germanium (Part V) are the map's first measurement instances, sitting on the same κ node as the computed values.
- **An agreement layer over the committed values.** The map already holds same-conditions comparisons, and a derived agreement view (`docs/data/agreement.json`, the `/agreement/` page) surfaces them as honest reporting of what the data contains. It groups the committed instances that name the same base observable, the same material, and the same physical conditions, differing only in the method or the code that produced them, and reports each group's spread. The concrete cross-code case is silicon at the Tersoff potential, $8\times8\times8$ mesh, $T = 300\,\mathrm{K}$: kaldo and phono3py agree on direct inversion at 26.9 versus 24.3 and on RTA at 19.5 versus 16.7 (in W/m K), the same run both cross-method and cross-code; the silicon DFPT LDA case, $19\times19\times19$ q-grid, quantum statistics, contrasts direct inversion at 147 against RTA at 140. The honesty caveat is the equivalence rule itself: two values are compared only when their base quantity, material, and every physical condition match and they differ solely in estimator, so a spread is a real method or code disagreement, never an apples-to-oranges artifact (the source paper is not a physical condition, so two solvers from one paper compare, and two codes on the same inputs are the strongest comparison).

Deferred to Phase 2:

- **Decomposed error bounds on κ.** The operator layer is designed to eventually produce the central value of κ together with an operator decomposition of its total error into the truncation error of the perturbation series and the discretization error of the chosen mesh. The architecture supports this; the implementation of the composition algebra is a research problem reserved for Phase 2.

23 Implementation layout

The Python package `omai` factors into two top-level sub-packages plus one per domain:

- `omai.operator` — operator layer machinery for the operator world. `Space` (base class), `ObservableSpace` and `HiddenSpace` (subclasses, with gauge-discipline fields on the latter), `Field`, `Operator`, `Parameter`, `Dimension`, `topological_order`, `validate_dag` (Level 1 discipline enforcement), and `GaugeAction` / `check_invariance` (Level 2 operator gauge proofs). Code-agnostic and domain-agnostic.
- `omai.representation` — the bridge.

- `units.py`: the `Unit` class, named unit instances, strict same-dimension `conversion_factor`.
- `normalizations.py`: the `Normalization` registry mirroring `units.UNITS` in shape — each entry carries a `to_operator` multiplier into canonical definitional form (e.g. `linewidth_2x_imag_sel` with factor 0.5, `eV_per_A2_per_nm` with factor 10).
- `adapter.py`: `SpaceRepresentationSpec` and `OperatorRepresentationSpec`; the two primitives `representation_to_operator` and `operator_to_representation` (each defined as the composition `unit.to_operator` · `normalization.to_operator`); and the diagnostic helpers `representation_scheme_match` and `representation_discretization_match`.
- `instance.py`: the `Representation` runtime data class and `represent()` constructor.
- `compare.py`: numerical and structural comparison APIs. `to_operator(m)` canonicalises a representation into operator form; `to_representation(m_op, target_spec)` is the inverse. `compare_operators(spec_a, spec_b)` returns an `OperatorComparisonResult` (categorical: do the two adapter specs describe the same operator?). `compare_representations` (aliased as `compare`) returns a `RepresentationComparisonResult` with the five-status verdict and residuals.

Code-agnostic.

- `omai.thermal_transport` — the lattice-thermal-transport domain.
  - `operator/nodes.py`: the domain's 54 spaces as `ObservableSpace` or `HiddenSpace` instances with their fields.
  - `operator/edges.py`: the domain's 55 operators with sympy formulas and `auxiliary_formulas` (the `compute_linewidth` $|V_3|^2$ kernel and the `solve_bte[direct_inverse]` collision matrix M); sympy symbols and `IndexedBase` declarations.
  - `operator/gauges.py`: concrete `GaugeAction` instances for the domain (e.g. U(1) phase on Eigenvectors). Mechanically verifies operator formulas are gauge-invariant.
  - `representation/kaldo.py`, `representation/phono3py.py`, `representation/phonopy.py`, `representation/shengbte.py`, `representation/qe.py`, `representation/ase.py`, `representation/l` `representation/gpumd.py`: per-code adapter specs (one file per code), declaring units, normalizations, schemes, and discretization choices. Every operator in scope for a code now carries an `OperatorRepresentationSpec`; trivial source / contraction ops carry no schemes but exist to mark coverage on the map. Coverage renders on the site map (`docs/map/`, generated data in `docs/data/`); the standalone per-code pipeline page was retired 2026-07-08 as redundant with the map's code rails. Per-operator scheme and discretization detail lives in the representation modules and this document; surfacing it in the map's edge panel is an open enhancement.

The split mirrors the operator layer's architectural commitment: `omai.operator` is the operator world; `omai.representation` is the bridge to the numeric world; `thermal_transport` is the first domain instantiated against them. Further domains follow the same pattern in their own folder (electronic transport has since landed exactly this way); no change to the operator-level packages is required to add one.

24 Open Questions and Deferred Decisions

We list the architectural questions that remain open and the principled grounds on which they were deferred.

24.1 Algebra of error composition (deferred to Phase 2)

The operator layer is designed to carry error formulas as operator expressions, but Phase 1 does not implement the algebra by which they compose. Composing two errors in independent small parameters ($O(\lambda^4)+O(N^{-2})$) is well-defined; composing two errors in the same parameter requires care; composing constant-offset errors (e.g., basis-set incompleteness) with asymptotic ones is yet another case. The right algebra is itself a research subproblem and will be developed alongside concrete worked examples in Phase 2. The Phase 1 type system reserves the slots required for error formulas (operation metadata, discretization error model, representation total error) but populates them with placeholders rather than computed compositions.

24.2 When to lift to Lean

We have committed to keeping the operator layer Lean-compatible without being Lean-dependent. The question of when (and whether) to actually project the operator layer into Lean is deferred until the Python prototype has stabilized. A reasonable trigger for this decision is the moment at which we have enough concrete examples that designing Lean types in isolation becomes counterproductive.

T3-evolve as a pre-Lean realisation. The sympy chain composer (`omai.operator.compose.compose_path`) is the in-sympy realisation of the Lean commitment in this subsection. Given a path of explicit-equation edges in the operator DAG, it substitutes each edge's RHS into the next, producing a composed symbolic expression at the path's terminal node. Implicit-equation edges (e.g., `solve_bte_*`) raise `ImplicitEdgeBoundary` with the partial expression attached — they mark the open research subproblem of composing across an implicit BTE solve, which is the natural next step toward a full Lean projection. The composer's existence is evidence that the operator-layer formulas are not merely documentation: they are mechanically composable today, and the same substitution machinery transliterates to a Lean tactic when the projection is eventually performed.

24.3 Provenance equivalence

The implemented kernel replaces provenance-based identity with content identity (Part IV): two states with the same content are the same state regardless of history, and adding an alternative producing route never re-mints a node. What remains genuinely open is the *edge*-level version of this question. Because an edge id includes its formula fingerprint, a factored and an expanded form of the same physics mint different edge ids and coexist as parallel producing edges. When the two are provably equivalent (e.g., Fourier transform of a real symmetric matrix and direct frequency-domain assembly yielding the same `Dispersion`), an explicit, reviewer-approved `equate` record in the log ties them together at the edge level. That record now exists in the store (Part IV); the general equivalence calculus, deciding which such equivalences hold by proof rather than by curation, is still deferred because we want concrete examples before designing it.

24.4 Sprint 1 scope

The first concrete deliverable is a Python implementation. We have considered three scopes:

1. *Narrowest:* kaldo only, harmonic sub-DAG (force constants → dynamical matrix → dispersion). 4–5 states, fastest to ship.
2. *Recommended:* kaldo only, full thermal-conductivity DAG. 10–15 states, end-to-end on silicon. The operator layer skeleton plus one full adapter.

3. *Broader:* kaldo plus phono3py, full DAG. Validates the cross-code story earlier but adds adapter work in parallel with framework design.

The recommended scope (option 2) is the smallest scope that is still a credible proof of concept of the framework, and the choice of one adapter (kaldo) lets us validate the operator layer design before generalizing. Sprint 2 adds phono3py and ShengBTE adapters, at which point the cross-code demo of §22 becomes real. *Status as of 2026-05*: the recommended scope shipped; Sprint 2 also shipped, with phono3py, phonopy, and ShengBTE all ingested and the cross-code Si silicon comparison (`experiments/silicon_shengbte/`) demonstrating κ_{RTA} agreement to $\leq 8\%$ across the three transport codes after the operator layer's spec-derived conversions.

24.5 Implementation language

We have committed to Python for Sprint 1. The trade-offs were:

- **Python (chosen).** All target codes have Python bindings or wrappable interfaces. The Python type system, augmented with `pydantic` or `attrs` and runtime checks, is sufficient for the operator layer's first-cut type discipline. The development speed is dominant, since adapter work is the bulk of the time cost.
- *Typed compiled language (Rust, OCaml, TypeScript).* Stronger type guarantees from the start. Rejected for Sprint 1 because the foreign-function interface to existing Python codes adds friction proportional to the operator layer's surface, and we want framework iteration to be cheap.
- *Lean.* Not the right tool for running scientific computations; reserved as a verification target, a reservation since cashed in: the verified layer of Part V compiles today.

25 Outlook

The operator layer is an architectural commitment, not an implementation. The implementation began as a Python skeleton plus adapters for three thermal-transport codes and has since grown to the 31-rail map of Part V. The artifact that motivates the project—a paper-worthy demonstration of cross-code reconciliation and decomposed error bounds for lattice thermal conductivity—is downstream of the operator layer but is the operator layer's first proof-of-value.

The project's stewardship follows its architecture (2026-07-12). The map, its evidence, and the rules that govern changes are the commons, held by OpenMaterials-AI, an open initiative structured as a foundation in formation: map data is licensed CC BY 4.0, the kernel Apache 2.0, and the governance document states the boundary rule plainly: the commons owns the ledger and its laws; companies own tools and products built on top. The interfaces and the AI improvement engine are built by Da Vinci Labs against that boundary. The same document states data ownership and fairness (2026-07-13): evidence stays its owner's (contribution grants a non-exclusive CC BY 4.0 license, never a transfer, with no copyright assignment and no CLA), raw simulation and experimental artifacts are never ingested, and attribution is enforced in both directions, upstream through code rails and page-anchored quotes as a merge gate, downstream through the map version's provenance.

Four longer-term directions follow naturally from the design:

- **Verification.** Project the operator layer into Lean, allowing approximation theorems to be formally proved rather than asserted. The provenance mechanism makes this projection well-defined: each operation in π corresponds to a Lean lemma whose statement is the operation's error formula. The first tiers are built, verified, and CI-gated (2026-07-17): `lean/` is a standalone lake package on Lean 4.31.0 depending on physlib, the community

Lean 4 physics library, pinned by revision. Tier 1 declares every exported node as a physics `Dimension` and proves one theorem per verified executable edge: 89 nodes and 12 edge theorems compile against physlib's five bases, and a generated seven-base extension adds the amount-of-substance, current, and luminous-intensity quantities, so 106 of the map's 114 nodes carry a machine-checked dimension. Of the eight left out, five (`Structure`, `Potential`, and the opaque autocorrelation kinds) have no dimensional content to prove; three (`CellVolume`, `AtomicMass`, `AtomCount`) have dimensions the exporter does not yet walk, generator work, not proof work. Tier 2 states the map's executable identities as real-valued theorems against Mathlib, 13 in all: composition theorems that substitute an upstream identity into a downstream one and prove the chained form equals the flat form, and law theorems for standalone rational operators; a units file anchors the map's SI convention. A CI gate recompiles the full proof on every pull request that touches the map or the generators, with warnings promoted to errors, so the verified badge is the live state of the claim. A public formalization roadmap rates all 114 operators by what a proof of the formula would take (12 rational identities closed by the generator, 44 needing real analysis, 18 trivially reflexive, 40 opaque code outputs with nothing algebraic to prove), and a hand-reviewed crosswalk links map nodes to the physlib declarations that formalize the same concepts. The approximation-error theorems remain future work.

- **Theory-experiment integration.** Extend the representation functor to ingest experimental observables. The operator layer's graph-alignment machinery applies uniformly, so a measured κ becomes another representation of the abstract κ, and "does the theory match experiment" becomes graph-alignment with one functor being empirical.

- **Grounded LLM agents.** An LLM operating over typed operator states (rather than text tokens) constructs workflows by emitting operator operations. The framework type-checks each emission, eliminating the drift problem that plagues current text-token agents. The action space is finite, typed, and physically meaningful.

- **Learned shortcuts.** The kernel already types ML surrogates as declared, non-authoritative shortcuts of exact-edge paths (`LearnedOperator`, 2026-07-13): the shortcut names its path, inherits its schemes, cites its model artifact and training labels, and stays permanently subordinate to the exact edges and the evidence. The first map declaration ships when a released model artifact and its evidence instances land together; the natural candidate is the three-phonon linewidth surrogate that amortizes `ForceConstants[order=3]` into weights.

The unifying claim of the design is that scientific computing has been organized around the wrong atomic unit. The instruction-in-an-input-file unit is too low; the natural-language description is too high; the equation-in-a-paper unit is too disconnected from execution. The right unit, at least for materials science, is the operator operation between typed physics states. Once that commitment is made, much of the rest of the architecture writes itself.

Part IV
The Kernel

The architecture of Part III is a design; this part records what was built. The kernel is the protocol core of Part II made real, first on the 51-node genesis map and since grown through its gates to today's map (Part V): composable dimensions with a dimensional gate, content identity by hashing, a log-first versioned store with a tamper-evident hash chain, a frozen genesis, and the source index. After the kernel landed, the map's source of truth is data, an

append-only log plus a materialized view; the Python operator layer became an authoring client that emits change records; and every future scan-derived contribution (Quantum ESPRESSO's DFT ground state, LAMMPS, the remaining materials skills) enters through validation gates rather than hand-edited Python.

The kernel was built before bulk growth from code scans, and deliberately so. Identity hashes include a node's dimension, gauge class, and index signature. Had those changed representation later, for instance flat dimension tags becoming composable exponent vectors, every hash in the map would re-mint on day one of the protocol. The map was small enough at genesis (then tens of nodes) to verify the genesis migration by hand; after the code scans it would not be.

26 Dimension algebra and the dimensional gate

Dimensions enter the identity hash, so they were made composable first. The flat `Dimension("thermal_cond` tags were replaced with exponent vectors over the seven SI base dimensions (M, L, T, Θ, N, I, J), with an `opaque` escape hatch (`exponents = None`) for deliberately unmodeled internals: the `Potential`, `Structure`, `Trajectory` fields, and the autocorrelation kinds that were typed opaque.

`Dimension(name, exponents: tuple[Fraction, ...] | None)` *(None = opaque)*

Multiplication, division, and integer powers are defined on dimensions; equality compares exponents, and opaque dimensions compare by name. Every existing named constant kept its name and all of its call sites (`ENERGY` $= M \cdot L^2 \cdot T^{-2}$, `THERMAL_CONDUCTIVITY` $= M \cdot L \cdot T^{-3} \cdot \Theta^{-1}$, and the rest), so the change was zero churn outside `dimensions.py`. The algebra preserves and extends `dimension_si_scale` compositionally: a derived dimension's SI scale is the product of its factors' scales, which the executor's monomial-contraction rescaling (Part III) depends on.

On top of the algebra sits a new validation gate. For closed-form `Eq` edges, the gate checks dimensional consistency of the left-hand side against the right-hand side wherever every symbol's dimension is known, drawn from the per-space fields and the edge parameters. This is the mechanical form of Part II's first validation rule ("dimensional agreement on every edge"), and it is the cheapest strong filter for everything the parsers will later propose. The gate partitions every closed-form `Eq` edge into *ok*, *documented-schematic violation*, or *skipped* (a formula the gate cannot check: an implicit solve, a procedural fit); the live tally is in Part V. The only two violations are `compute_gruneisen` and `compute_phase_space_3phonon`, kept deliberately schematic (see §33).

27 Content identity

Names and symbols are metadata, never identity. Hashes are `sha256` over canonical JSON (sorted keys, no whitespace); the store keeps full digests and displays a 12-hex prefix.

Node id. A node's id is

$$\mathrm{H}(\text{quantity\_tag},\ \text{field\_signatures},\ \text{gauge\_class},\ \text{sorted(labels)}),$$

where each ingredient is:

- **quantity_tag** is a curated identifier (`entropy`, `heat_capacity`, `potential`, `structure`, `dynamical_matrix`, `bare_dynamical_matrix`, ...) from a controlled, versioned registry in core, exactly like labels and index kinds. It is the semantic distinction the structural fields cannot carry.
- **field_signatures** is the sorted multiset of (`dimension_canonical`, `index_kind_signature`) over *all* of the space's fields (`Trajectory` has two), not a single dimension. The `index_kind_signature`

is the tuple of index *kinds* drawn from a small controlled registry (`atom`, `cartesian`, `qpoint`, `branch`, `lattice_vector`, `timestep`, `lag`, `omega_bin`, `mfp_bin`, ...); kinds carry the gauge and symmetry semantics Part II assigns to indices.

- **gauge_class** is `observable` for an observable node, `parameter` for a promoted parameter, or the string `hidden/<kind>/<gauge_group>` for a hidden node, where `kind` is `scaffolding` or `approximation` and `gauge_group` is a registered ascii identifier.
- **labels** are the semantic type parameters (`bte_solver=direct_inverse`, `transport_model=wigner`, `channel=isotope`, `order=2`). Label keys and values live in a controlled, versioned registry, never free-form strings.

The production route is not part of the node id: Pattern C nodes (a shared output with alternative producing edges) keep one stable id, and adding a producer never re-mints. `tier` and names, symbols, and descriptions are presentation metadata, never identity. Promoted parameters (`CellVolume`, `AtomicMass`, `AtomCount`) receive quantity tags and dimensions like any node.

The collision rationale: why the quantity tag. An earlier candidate made identity purely type-content, hashing only the field signatures, gauge class, and labels with no curated tag. It was stress-tested against the live map and *false-merged seven real pairs of distinct quantities*, because physics is full of same-typed distinct quantities:

- `Entropy` = `HeatCapacity` (same dimension, indices, gauge);
- `HelmholtzFreeEnergy` = `InternalEnergy`, and their two `Molar*` counterparts (two more pairs);
- `Potential` = `Structure` (both opaque source leaves);
- `BareDynamicalMatrix` = `DynamicalMatrix`;
- `Gruneisen` = `PhaseSpace3Phonon`.

Seven false merges is disqualifying, so the semantic distinction is carried by a curated `quantity_tag` from the registry, and convergence becomes registry-mediated: two contributors converge by mapping to the same registered quantity, and the structural fields are validated against the tag (a wrong dimension for a claimed quantity fails validation). This softens Part II's "converge automatically" wording: convergence is through the shared registries, not by structural coincidence.

The consequence to hold onto is that *labels carry the disambiguation work.* Same-typed variants that must stay distinct (`wigner_populations` vs `wigner_coherences` vs `wigner`; cumulative κ with respect to ω vs with respect to mean free path) are distinct only because of their labels, which is why label keys and values are part of the protocol registry. A contribution using an unregistered label key fails validation. Two parallel derivations of the same-typed unlabeled quantity (for example a second producer of κ) converge onto one node with two producing edges, which is the intended reconciliation-at-observables behavior, not a false merge: the distinct physics stays on the edges, whose formula fingerprints differ. The gauge class already separates RTA-style approximations from observables without any label.

Edge id. An edge's id is

H(output node id, unordered set of input node ids, formula_fingerprint, sorted(schemes)).

The `formula_fingerprint` is the normalized `sympy.srepr` of the formula. Sympy's canonical `Add`/`Mul` ordering already collapses $c\,v^2\,\tau$ and $\tau\,v^2\,c$ to one tree; operators whose formula is

a LaTeX string fingerprint the string after whitespace normalization. Editing a formula re-mints the edge and writes a supersede record. This is strict but honest: "every edge carries its formula" is the product's core claim, so a different formula is a different edge. Two points follow. First, the operator *name* is not in the hash: names are metadata (edited via `edit_meta`), and identity is structural, which matches Part III's "operation identity is parameterized" once "parameters affecting physics" is read as schemes, which are in the hash. Second, the fingerprint canonicalizes commutative reordering but *not* general algebraic equivalence: a factored and an expanded form of the same physics mint different edge ids, coexist as parallel producing edges, and are reconciled only by an explicit, reviewer-approved `equate` record in the log. That record realizes, at the edge level, the "provenance equivalence" mechanism Part III defers; it is a curation action, never automatic. Approximation-error formulas (Part III's Phase-2 slots on operations) are metadata, not identity: attaching or refining an error formula later must not re-mint an edge.

Version hash. The version hash chains the log, H(previous version hash ‖ canonical(change record)), the tamper-evident chain from Part II, unchanged. The consequences kept from Part II are structural convergence (same physics, same hash), parallel routes for genuinely different decompositions, and the Merkle ripple on foundational edits, handled by supersede records.

28 The protocol registries

Every string that enters a hash lives in a controlled, versioned registry in core, so that a `qpoint` means the same kind in every domain and cross-domain convergence is well-defined. There are four such registries:

1. **Quantity tags** (§27): the curated per-quantity identifiers.
2. **Index kinds**: one global registry of about twelve kinds (`atom`, `cartesian`, `qpoint`, `branch`, `lattice_vector`, `timestep`, `lag`, `omega_bin`, `mfp_bin`, ...). Index kinds enter node identity and cross-domain convergence depends on them, so the registry is shared, not per-domain.
3. **Label keys and values**: `bte_solver`, `transport_model`, `channel`, `order`, `wrt`, and their permitted values.
4. **Gauge-group names**: the named gauge equivalences on hidden spaces, normalized to ascii identifiers before they enter any hash.

A contribution referencing an unregistered tag, index kind, label key, or gauge group fails validation. Registering a new one is a deliberate, versioned protocol change.

29 The log-first store

The versioned artifact lives in `map/` at the repository root, a peer of `omai/` and `docs/` rather than a view inside the docs site, because it is the protocol artifact anyone can fork and it stands on its own.

Record shapes. `map/log.jsonl` is the append-only change log, one canonical-JSON line per record with the shape

{`seq`, `op`, `payload`, `author`, `date`, `reason`, `prev`, `version`},

where `op` is one of `add_node`, `add_edge`, `edit_meta`, `deprecate`, `supersede`, `equate`. The `edit_meta` op covers names, symbols, and descriptions, the non-identity content that must be editable without re-minting; `supersede` ties an old subgraph to its replacement after a foundational edit; `equate` records a reviewer-approved equivalence between two parallel producing

edges. A payload carries everything needed to rebuild the map without importing any Python module: the full identity dict that was hashed, and for edges the formula as a sympy srepr string plus its display LaTeX.

The materialized view. `map/current/nodes.json` and `map/current/edges.json` are the materialized current view, regenerated from a full log replay on every push and never hand-edited. They are a reviewable mirror keyed by uid. `docs/data/graph.json` is re-derived from `map/current` with the same shape as before plus an `id` field, so the site keeps reading it unchanged.

The hash chain. The `version` field chains the log,

$$\texttt{version} = \text{sha256}(\texttt{prev} + \text{canonical(record without version)}),$$

where `prev` is the previous record's version and the genesis record's `prev` is sixty-four zeros. This makes the history tamper-evident and path-dependent.

Store API. `omai/store.py` exposes `push(change) -> version_hash`, `read() -> Map`, `read(version) -> Map` (by log replay), `diff(a, b) -> [records]`, and `verify()`. `verify()` recomputes every link in the chain, checks the chain and the dense `seq`, replays the full log, and compares the materialized view structurally against the replay, so a stale or truncated `current/` file is caught rather than trusted, as is any dangling reference. Contributions enter through the gated `Store.propose`, which validates an ordered contribution against the six gates (registry, identity, reachability, connectivity, gauge, and dimensional) before any record lands, pushes all of them or none, and treats an exact re-proposal as a no-op that leaves the head unchanged. The `python -m omai.sync` tool diffs the Python operator layer against the materialized view and proposes change records for review, applying additions and metadata edits behind a flag while always routing a re-mint (an identity-content change) through an explicit supersede record rather than an automatic rewrite.

The first supersedes. The store has now exercised that supersede machinery for real, twice. A whole-map physics review reformulated two edges from opaque solver calls into closed form: the reaction energy went from $E^{\text{rxn}}[\Delta H_f]$ to the stoichiometric sum $\Delta E_{\text{rxn}} = \sum_i c_i^{\text{rxn}} H_{f,i}$ (records 193-194), and the ionic conductivity went from $\sigma^{\text{NE}}[D, T, \text{Structure}]$ to the executable Nernst-Einstein relation $\sigma = n_c z^2 e^2 D/(k_B T)$, promoting the carrier density n_c to a first-class node (records 200-201). In each case the formula fingerprint changed, so the edge id changed: the new edge is minted, the old edge carries a `superseded_by` pointer, and a sync of the operator layer skips the retired entry rather than re-proposing it. This is the versioning contract holding up under a genuine change of physics, not a metadata edit: a reformulation that alters what an edge computes mints a new identity, and the chain records the retirement rather than rewriting history in place.

30 Genesis

The genesis migration walks the Python `DOMAINS`, emits one `add_node` or `add_edge` record per element in topological order (author `genesis`), and freezes the resulting version hash. The migration reads no clock and no randomness, so the hash is reproducible from the code alone.

The frozen genesis version hash, held in `map/GENESIS`, is

`e6e8044e92039696417b53b220b0f3f10559a286b0eaabbe7ea4167ff510f6cd`

produced by the deterministic migration on the genesis date **2026-07-07**, covering **51 nodes, 49 edges, 100 records**. Genesis is the frozen prefix, not the head: a repository-state test replays the first 100 committed records and asserts they reproduce the `GENESIS` hash exactly, while the Python-vs-store drift alarm is the sync-clean check (`python -m omai.sync` proposing nothing on a pristine repository). The store has since had its first two live uses of the protocol: record 101, an `edit_meta` unifying the BareDynamicalMatrix display symbol, and records 102-108, the DFT ground-state domain (Structure, TotalEnergy, Forces, Stress and their three producing operators), proposed by the sync tool and admitted through all six gates. Record 109 (the finite-displacement route from Forces to the force constants, the first post-genesis Pattern C producer) and records 110-117 (the mechanics domain: the elastic tensor, its Voigt moduli, and the pressure, whose first apply attempt the connectivity gate correctly rejected until the elastic edge took the stress route) followed through the same gates. All landed as ordinary change records on top of the untouched genesis prefix, which is exactly the accumulation the kernel was built to support.

The thermodynamic-identities domain. A whole-map physics review of the combined formulas added a small domain (records 183-192) whose whole job is to tie the map's separate branches together with six executable, gate-proven relations: the Grüneisen identity $\gamma_{th} = K\alpha_V/C_V$, the total thermal conductivity $\kappa_{\mathrm{tot}} = \kappa + \kappa_e$, the molar volume $V_m = N_A V_{\mathrm{cell}}$, the $C_P - C_V$ relation $C_P = C_V + KTV_m\alpha_V^2$, the power factor $\mathrm{PF} = S^2\sigma_{el}$, and the thermoelectric figure of merit $ZT = \mathrm{PF}\,T/\kappa_{\mathrm{tot}}$. Because a production route is never part of a node's identity, several of these close as second and third producers of quantities the map already carried: the thermal Grüneisen parameter and the constant-pressure heat capacity each now have two independent routes, and the bulk modulus three. That is the multi-producer pattern the identity design was meant to admit: a new derivation of an existing quantity is a parallel edge into the same node, not a re-minted node, so the routes stand side by side as cross-checks.

The composites domain. A composites domain (records 228-239) carries the effective thermal conductivity of a two-phase composite, a dispersed filler in a continuous matrix, with interfacial (Kapitza) resistance, via Nan-type effective-medium theory (Nan et al., J. Appl. Phys. **81**, 6692 (1997)). Its new physics is the interface: a thermal boundary conductance G ($\mathrm{W/(m^2\,K)}$, the new `INTERFACE_CONDUCTANCE` dimension) whose Kapitza radius $a_K = k_m/G$ is a length, the crossover below which a conductive filler *lowers* the composite conductivity. The matrix and filler conductivities enter as role-labeled members of the one thermal-conductivity family (`[role=matrix]`, `[role=filler]`), and the effective conductivity is a family member produced by the effective-medium theory (`[effective_medium=nan,orientation=random]` and its aligned sibling), which resolves onto the method-neutral `ThermalConductivity` observable a measurement of the composite reports. Four of its five edges are closed-form sympy the dimensional gate proves; the Hasselman-Johnson spherical-limit formula (J. Compos. Mater. **21**, 508 (1987)) is a second producer of the random effective conductivity that must agree with Nan at aspect ratio 1, a gate-level cross-check pinned to machine precision. The reference DGEBA epoxy + 5 vol% graphene-nanoplatelet draft gives $\kappa_c = 1.2452\,\mathrm{W/(m\,K)}$ from the map's own closed-form edges (the evidence instance keeps its contribution provenance in its source ref).

Instance uid pinning. Instances are additive JSON that attach a value to a node. The uid pin lives in the regenerated projection, not the committed record: at build time each value is stamped with its node's live uid, so the genesis migration re-pinned the harvested instances to element hashes mechanically. A value pinned to an exact element and map version either stays pinned for reproducibility or follows the supersede chain for currency, as Part II requires.

31 The index

The index is the source registry beside the map, entries organized by source, each pinning that source's coverage to a specific element hash at a specific map version. The genesis migration writes `index/codes/` stubs for the nine existing representations: each `codes/<rep>.json` is {`representation`, `map_version`, `covers`}, where `covers` lists the nodes the code maps (node uid, the code's API name, its declared unit), sorted by node, and `map_version` is the frozen genesis hash. `papers/` and `experiments/` arrive with their first entries later. The index files are generated, never hand-edited (`python -m omai.index_data`). This keeps Part II's build order (store, then index, then parsers) intact rather than deferring the index indefinitely.

32 Resolved decisions and their rationale

Four design decisions were resolved during the kernel's design. They are recorded here with the reasoning that settled them, because each has structural consequences.

1. **Node identity is quantity tag + type-content + labels** (amended after the collision stress-test). The alternative, Part II's original "derived id = hash(operation, unordered input ids)", was resolved in favor of type-content identity because the operation-based rule mis-handles Pattern C nodes (a shared output reached by alternative edges would get a different id per producing operation, re-minting the node whenever a producer is added). Pure type-content identity was then amended to add the curated quantity tag after it false-merged the seven pairs of §27. The final rule keeps one stable id for a Pattern C node and never re-mints on adding a producer.

2. **Edge identity includes the formula fingerprint** (§27). Editing a formula re-mints the edge and writes a supersede record. The rule is strict but honest, since "every edge carries its formula" is the product's core claim. The operator name stays out of the hash (metadata), and general algebraic equivalence is reconciled by an `equate` record, not automatically.

3. **One global index-kind registry in core** (§28). Index kinds enter node identity and cross-domain convergence depends on a `qpoint` meaning the same kind everywhere, so the registry is shared rather than per-domain. The same registry treatment extends to every string that enters a hash: quantity tags, label keys and values, and gauge-group names. The six gauge-group strings previously in use were free-form (one contained a unicode multiplication sign) and were normalized into registered ascii identifiers before entering any hash.

4. **Store location `map/` at the repository root** (§29). The versioned artifact (`map/log.jsonl`, `map/current/`, `map/GENESIS`) is a top-level directory, peer of `omai/` and `docs/`, because it is the protocol artifact anyone can fork, so it stands on its own rather than living inside the docs site. `docs/data/graph.json` is re-derived from `map/current`.

33 Pre-genesis formula normalization

Because edge identity includes the formula fingerprint, formulas were normalized *before* genesis, so the frozen log is born clean instead of opening with supersede chains. The scope came from the P1 dimensional gate:

- The three schematic `n_BE(omega/T)` arguments (`compute_entropy`, `compute_internal_energy`, `compute_anharmonic_linewidth`) became the dimensionless $\hbar\omega/(k_B T)$, matching their sibling closed forms (`compute_heat_capacity`, `compute_free_energy`), which already passed the gate.

- `compute_kappa[transport_model=wigner_coherences]` had its missing $1/\omega$ weighting corrected. The corrected form was transcribed, not recalled from memory, from the vendored phono3py Wigner (SMM19) solver, at `phono3py/phono3py/conductivity/ms_smm19/kappa_solvers.py:` (the prefactor $0.25\,(\hbar\omega_s + \hbar\omega_{s'})(C_s/\hbar\omega_s + C_{s'}/\hbar\omega_{s'})$), and cross-checked against kaldo's off-diagonal QHGK kernel. This is the frequency-weighted Simoncelli form $(\omega+\omega')/2 \cdot (c/\omega + c'/\omega')$ of Simoncelli, Marzari, and Mauri, Nat. Phys. **15**, 809 (2019); it carries one fewer power of frequency than the earlier encoding, restoring κ_C to the thermal-conductivity dimension. The dimensional gate then classifies the edge as ok.
- The `DynamicalMatrix` and `BareDynamicalMatrix` field dimension was corrected from `FREQUENCY` to `FREQUENCY**2` (the mass-weighted Hessian, whose eigenvalues are ω^2). The catalog dimension string changed accordingly; this is a truth fix, not drift.
- `Gruneisen` and `PhaseSpace3Phonon` stay documented-schematic (the two known dimensional-gate violations of §26), by choice.

34 What remains

The dimension algebra and gate (P1), the identity module and index-kind registry (P2), the store, genesis migration, and `verify()` (P3), and the contribution gates and sync tool (P4) are built: the six gates behind `Store.propose` and the `python -m omai.sync` tool are the *Store API* paragraph of §29. P5 is done in its minimal honest form: the map page shows each node's content-hash prefix on the provenance panel and the store head version it renders from (a `docs/data/version.json` stamp written next to the graph data), and the index pins every coverage file to the head at generation time. The Quantum ESPRESSO DFT ground-state domain landed through the gates as records 102-108: the first contribution pushed through the store, the worked example the kernel was built for, validated by the Si cross-check per the `ingest_code` discipline of Appendix A (verify the fresh code against a trusted one on the same material before any single-code result is trusted). What remains, in order:

- **The parsers**, the automated on-ramp. The paper parser's value-extraction half is built (Part II); the code and run parsers remain, each its own spec, plan, and build cycle.
- **Further domains through the same gates**: the deferred ground-state quantities (ChargeDensity, Wavefunctions, dvscf) and the `mat-*` bulk encode. The finite-displacement Forces → force-constants route landed 2026-07-08 as record 109, the map's first post-genesis Pattern C producer; the mechanics domain followed the same day as records 110-117.

Part V
Status

35 Where the map is today

The map is at store head version `9802d9e854c915eb47d867575730a556ab7f4a565e392bf5b29da58338f084` (2026-07-13, 239 records), one hundred thirty-nine change records past the frozen genesis hash `e6e8044e92039696417b53b220b0f3f10559a286b0eaabbe7ea4167ff510f6cd` (2026-07-07, the untouched 100-record prefix). Since the lineage became the one named artifact, the published stamp also carries the unified *lineage version* (`f69b18c18fb7...`): one rolling content hash over the derivation graph and the instance-format rules together, appended to a chain whose root is the genesis, so a change to either the physics or the record format moves the one version the published stamp and every page cite (records themselves stay version-free: the projection

stamps each value with the live node uid at build time, so committed values follow the current map; a per-record frozen pin is designed, not yet carried on the records). The live numbers:

- **Structure.** 114 nodes (111 observable and hidden quantities plus 3 promoted parameters), 274 rendered links, 114 deduplicated operators, rendered in sixteen physics tiers (Sources, Harmonic, Thermodynamics, Scattering, Transport, Molecular dynamics, Ground state, Mechanics, Stability, Thermochemistry, Quasi-harmonic, Molecular, Electronic transport, Diffusion, Thermoelectric, Composite). The newest tier (Composite) and its interface-conductance and effective-medium nodes landed with the composites domain (composite effective thermal conductivity with interfacial resistance, cross-checked against the Hasselman-Johnson spherical limit); the Thermoelectric tier before it landed with the thermodynamic-identities domain. The rendered-link count exceeds the operator count because the site graph draws one link per (input $\rightarrow$ output) pair while an operator with several inputs is a single edge in the store.
- **Representations.** 31 codes mapped, with per-code variable coverage: kaldo 34, phono3py 31, phonopy 22, pymatgen 22, ShengBTE 20, mp-api 13, Quantum ESPRESSO 13, LAMMPS 12, VASP 10, GPUMD 8, pycalphad 7, matgl 5, mat-elasticity 5, amset 5, ORCA 5, pymatgen-analysis-diffusion 5, i-PI 4, fairchem 4, MACE 4, OpenMM 4, ASE 3, mat-diffusion-analysis 2, mescal 2, plumed 2, smol 2, xtb 2, mat-equation-of-state 1, mat-surface-adsorption 1, rxn-network 1, diffcsp 1, mattergen 1.
- **Instances.** 91 recorded values, 87 simulation and 4 measurement. They span the newer domains (formation and hull energies, elastic and equation-of-state moduli, CALPHAD Gibbs energies, quasi-harmonic thermal expansion, molecular NEB barriers and bond dissociation energies) as well as the original thermal-transport and ground-state values from the Si and Ge cross-checks (the SCF total energy and the Γ optical frequency). Thirty-five of the values are paper-sourced evidence, carrying an identity-bearing `paper:` source in the lineage and landed with a verbatim, page-located quote; they come from ten published papers (kaldo 2020, the QHGK/Green-Kubo Isaeva 2019, Esfarjani 2011, the 2021 carbon-nanotube and 2025 kappa-limit BTE studies, $PtSe_2$ 2021, quantum-elastic PIMD/TDEP 2024, SiGe disorder 2024, amorphous alloys 2020, and the Balandin 2008 graphene measurement). The four measurements are the Glassbrenner–Slack 1964 steady-state thermal conductivities of bulk single-crystal silicon ($156\,\mathrm{W/(m\,K)}$) and germanium ($60.2\,\mathrm{W/(m\,K)}$) at $300\,\mathrm{K}$ and natural isotopic abundance, the Balandin 2008 suspended-graphene Raman value, and the $PtSe_2$ 2021 in-plane FDTR value. The four newest simulation values (2026-07-18) are fully-specified engine-adapter evidence: kaldo and phono3py silicon thermal conductivities, RTA and direct-inverse, with every knob in-hash down to the sha256 of the vendored Tersoff potential, each paired with a committed conformance target (eight targets total) that a re-run must reproduce within its stated tolerance.
- **Dimensional gate.** Of the 114 operator edges the gate walks, 43 are ok, 2 are documented-schematic violations (`compute_gruneisen` and `compute_phase_space_3phonon`, kept schematic by choice), and 69 are skipped (not closed-form `Eq` edges the gate can check, most of them the opaque solver edges of the newer domains, plus the composites depolarization `Piecewise` and the resolve identity). The two ground-state derivatives (`compute_forces_hf` and `compute_stress_cell`) and all four mechanics edges (the elastic tensor as $-\partial\sigma/\partial\varepsilon$, the pressure trace, and both Voigt moduli) are proven ok, as are the six thermodynamic-identity edges (the Grüneisen identity, total thermal conductivity, molar volume, the $C_P - C_V$ relation, the power factor, and ZT) and the three composite effective-medium edges (the Nan random and aligned mixing formulas and the Hasselman-Johnson spherical cross-check).
- **Verified layer.** 89 nodes and 12 edge theorems compile against physlib (Lean 4.31.0, pinned by revision), the generated seven-base extension lifts dimensional coverage to 106 of

114 nodes (five of the eight remaining are opaque by design; three await exporter coverage), and 13 identity theorems check against Mathlib; a CI gate recompiles the proof on every relevant pull request with warnings promoted to errors. The formalization roadmap page rates every operator by what proving its formula would take.

- **Tests.** The suite is 1732 passing.

Code	Variables mapped	License
kaldo	34	BSD-3-Clause
phono3py	31	BSD-3-Clause
phonopy	22	BSD-3-Clause
pymatgen	22	MIT
ShengBTE	20	GPL-3.0
mp-api	13	BSD-3-Clause
Quantum ESPRESSO	13	GPL-2.0
LAMMPS	12	GPL-2.0
VASP	10	Proprietary
GPUMD	8	GPL-3.0
pycalphad	7	MIT
matgl	5	BSD-3-Clause
mat-elasticity	5	MIT
amset	5	BSD-3-Clause
ORCA	5	Proprietary
pymatgen-analysis-diffusion	5	BSD-3-Clause
i-PI	4	GPL-2.0-or-later / MIT
fairchem	4	MIT
MACE	4	MIT
OpenMM	4	MIT (LGPL parts)
ASE	3	LGPL-2.1-or-later
mat-diffusion-analysis	2	MIT
mescal	2	MIT
plumed	2	LGPL-3.0
xtb	2	LGPL-3.0
smol	2	BSD-3-Clause
mat-equation-of-state	1	MIT
mat-surface-adsorption	1	MIT
rxn-network	1	BSD-3-Clause
diffcsp	1	MIT
mattergen	1	MIT

Every code we represent must be cited and must carry its license. The License column above is read from the actual license text (the vendored LICENSE files for kaldo, LAMMPS, Quantum ESPRESSO, phonopy, phono3py, ShengBTE, and the AtomisticSkills skills; package metadata for the pip-distributed codes; the project's GitHub LICENSE for GPUMD, MatterGen, DiffCSP, and OpenMM; the i-PI repository's `licenses/LICENSE.md` for i-PI, distributed under both the GPL and MIT at the user's choice). VASP and ORCA are proprietary: VASP under the commercial license at vasp.at, ORCA free for academic use with commercial licensing through FACCTs. The mat-* rows are AtomisticSkills skills and carry that project's MIT license. The single source of truth is `omai/representation/credits.py`, and an enforcement test (`tests/test_code_credits.py`) fails if any mapped rail lacks a citation or a license, so no future rail can land uncredited.

35.1 Code citations

The principal code rails on the map are cited below (the remaining rails, i-PI, PLUMED, and MESCAL, carry their citations in `omai/representation/credits.py`, the single source of truth the enforcement test reads). Where one code serves nodes through different methods, a per-node override (`PER_NODE_CREDITS`) carries the method citation, so a quantity is never credited to a formula that did not produce it. References were taken from each code's own citation guidance where it exists (kaldo, LAMMPS, phonopy, phono3py, and the AtomisticSkills paper were read this way) and otherwise from the canonical method paper, verified against the project's citation page.

- **kaldo**: G. Barbalinardo, Z. Chen, N. W. Lundgren, D. Donadio, *Efficient anharmonic lattice dynamics calculations of thermal transport in crystalline and disordered solids*, J. Appl. Phys. **128**, 135104 (2020). doi:10.1063/5.0020443.
- **ShengBTE**: W. Li, J. Carrete, N. A. Katcho, N. Mingo, *ShengBTE: A solver of the Boltzmann transport equation for phonons*, Comput. Phys. Commun. **185**, 1747 (2014). doi:10.1016/j.cpc.2014.02.015.
- **phonopy**: A. Togo, L. Chaput, T. Tadano, I. Tanaka, *Implementation strategies in phonopy and phono3py*, J. Phys. Condens. Matter **35**, 353001 (2023). doi:10.1088/1361-648X/acd831.
- **phono3py**: A. Togo, L. Chaput, I. Tanaka, *Distributions of phonon lifetimes in Brillouin zones*, Phys. Rev. B **91**, 094306 (2015). doi:10.1103/PhysRevB.91.094306.
- **GPUMD**: Z. Fan, W. Chen, V. Vierimaa, A. Harju, *Efficient molecular dynamics simulations with many-body potentials on graphics processing units*, Comput. Phys. Commun. **218**, 10 (2017). doi:10.1016/j.cpc.2017.05.003.
- **LAMMPS**: A. P. Thompson, H. M. Aktulga, R. Berger, et al., *LAMMPS: a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales*, Comput. Phys. Commun. **271**, 108171 (2022). doi:10.1016/j.cpc.2021.108171.
- **Quantum ESPRESSO**: P. Giannozzi, S. Baroni, N. Bonini, et al., *QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials*, J. Phys. Condens. Matter **21**, 395502 (2009). doi:10.1088/0953-8984/21/39/395502.
- **VASP**: G. Kresse, J. Furthmüller, *Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set*, Phys. Rev. B **54**, 11169 (1996). doi:10.1103/PhysRevB.54.11169.
- **MACE**: I. Batatia, D. P. Kovács, G. N. C. Simm, C. Ortner, G. Csányi, *MACE: Higher order equivariant message passing neural networks for fast and accurate force fields*, NeurIPS 35 (2022), arXiv:2206.07697.
- **matgl (M3GNet)**: C. Chen, S. P. Ong, *A universal graph deep learning interatomic potential for the periodic table*, Nat. Comput. Sci. **2**, 718 (2022). doi:10.1038/s43588-022-00349-3.
- **fairchem (UMA)**: B. M. Wood, M. Dzamba, X. Fu, et al., *UMA: A Family of Universal Models for Atoms* (2025), arXiv:2506.23971.
- **amset**: A. M. Ganose, J. Park, A. Faghaninia, R. Woods-Robinson, K. A. Persson, A. Jain, *Efficient calculation of carrier scattering rates from first principles*, Nat. Commun. **12**, 2222 (2021). doi:10.1038/s41467-021-22440-5.
- **pymatgen**: S. P. Ong, W. D. Richards, A. Jain, et al., *Python Materials Genomics (pymatgen): A robust, open-source python library for materials analysis*, Comput. Mater. Sci. **68**, 314 (2013). doi:10.1016/j.commatsci.2012.10.028.

- **mp-api (Materials Project)**: A. Jain, S. P. Ong, G. Hautier, et al., *Commentary: The Materials Project: A materials genome approach to accelerating materials innovation*, APL Mater. **1**, 011002 (2013). doi:10.1063/1.4812323.
- **smol**: L. Barroso-Luque, J. H. Yang, F. Xie, et al., *smol: A Python package for cluster expansions and beyond*, J. Open Source Softw. **7**, 4504 (2022). doi:10.21105/joss.04504.
- **pymatgen-analysis-diffusion**: Z. Deng, Z. Zhu, I.-H. Chu, S. P. Ong, *Data-Driven First-Principles Methods for the Study and Design of Alkali Superionic Conductors*, Chem. Mater. **29**, 281 (2017). doi:10.1021/acs.chemmater.6b02648.
- **rxn-network**: M. J. McDermott, S. S. Dwaraknath, K. A. Persson, *A graph-based network for predicting chemical reaction pathways in solid-state materials synthesis*, Nat. Commun. **12**, 3097 (2021). doi:10.1038/s41467-021-23339-x.
- **DiffCSP**: R. Jiao, W. Huang, P. Lin, J. Han, P. Chen, Y. Lu, Y. Liu, *Crystal Structure Prediction by Joint Equivariant Diffusion*, NeurIPS 36 (2023), arXiv:2309.04475 (the map uses the space-group-constrained variant DiffCSP++, arXiv:2402.03992).
- **MatterGen**: C. Zeni, R. Pinsler, D. Zügner, et al., *A generative model for inorganic materials design*, Nature **639**, 624 (2025). doi:10.1038/s41586-025-08628-5.
- **pycalphad**: R. Otis, Z.-K. Liu, *pycalphad: CALPHAD-based Computational Thermodynamics in Python*, J. Open Res. Softw. **5**, 1 (2017). doi:10.5334/jors.140.
- **OpenMM**: P. Eastman, J. Swails, J. D. Chodera, et al., *OpenMM 7: Rapid development of high performance algorithms for molecular dynamics*, PLoS Comput. Biol. **13**, e1005659 (2017). doi:10.1371/journal.pcbi.1005659.
- **ORCA**: F. Neese, *Software update: The ORCA program system, Version 6.0*, WIREs Comput. Mol. Sci. **15**, e70019 (2025). doi:10.1002/wcms.70019.
- **ASE**: A. Hjorth Larsen, J. J. Mortensen, J. Blomqvist, et al., *The atomic simulation environment: a Python library for working with atoms*, J. Phys. Condens. Matter **29**, 273002 (2017). doi:10.1088/1361-648X/aa680e.
- **mat-elasticity, mat-diffusion-analysis, mat-equation-of-state, mat-surface-adsorption** (AtomisticSkills skills): B. Deng, B. Li, M. Cox, et al., *Harnessing AtomisticSkills for Agentic Atomistic Research* (2026), arXiv:2605.24002. doi:10.48550/arXiv.2605.24002.

36 The build order ahead

The kernel (dimension algebra and gate, content identity, the log-first store, the frozen genesis, the source index) is built (Part IV). The order from here:

1. **P4 (built)**: the six contribution gates run behind `Store.propose`, and `python -m omai.sync` proposes change records from the operator layer for review (Part IV).
2. **P5 (built)**: the site renders from the store's materialized view and shows its provenance: each node's uid prefix on the map panel and the store head version on the legend, stamped by `docs/data/version.json`; the index pins coverage to the head at generation time.
3. **The Quantum ESPRESSO DFT ground-state domain (done)**: the first contribution through the gates (records 102-108), validated by the Si cross-check (`experiments/qe_si_crosscheck`) with the first QE instances attached to the map.
4. **The parsers**, the automated on-ramp. The paper parser's value-extraction half is built (Part II); the code and run parsers remain to build, each its own spec, plan, and build cycle.

5. **Further domains and the `mat-*` bulk encode**, growing the map through the same gates. Mechanics landed 2026-07-08 as records 110-117 (the elastic tensor, its Voigt moduli, and the pressure, via LAMMPS ELASTIC and mat-elasticity, with the two Cu moduli as its first instances); the AtomisticSkills scans then grew the map through record 182 (2026-07-10) with the stability, thermochemistry, quasi-harmonic, molecular, and electronic-transport domains and the pymatgen, mp-api, CALPHAD, amset, ORCA, and OpenMM rails, and a whole-map physics review added the thermodynamic-identities domain and two closed-form supersedes (records 183-203).
6. **The app**, once the protocol and a rich mother map exist.

Appendices

A Ingesting a code

Goal. Given an external materials-science code (kaldo, phono3py, ShengBTE, LAMMPS, Quantum ESPRESSO, ...), produce a Python module of `SpaceRepresentationSpec` and `OperatorRepresentationSpec` instances that pin the code's outputs and parameters to the operator DAG, so that `compare()` between this code and any other already-ingested code returns `EXPECTED_AGREE` on the appropriate Observables.
Prerequisites.

- The operator DAG (`omai.operator`, plus a domain instance like `omai.thermal_transport.operator`) already declares the relevant Spaces and Operators. If a state the code emits has no operator counterpart, that is out of scope for this skill: file it as a substrate-extension task.
- At least one reference adapter for the same domain is already ingested (kaldo and phono3py for thermal transport). The reference adapters are the ground truth against which the new one is validated.
- The external code's authoritative documentation is readable from the workspace (README, user manual, source).

A.1 Procedure

0. Working rules (apply throughout)

Two rules learned the hard way during prior ingestions; check yourself against them before each spec decision.

- **Grep before classifying as "out of scope."** Before declaring that a code does not expose a given state, run a literal grep across the code's source for the obvious API names (e.g. `phase_space`, `dos`, `gruneisen`). Several states that an initial reading of the docs missed turned out to have a clean public API one or two commits deep (kaldo's `Phonons.phase_space`, kaldo's `plotter.plot_dos`, phonopy's `PhonopyGruneisen.get_gruneisen()`). A `git grep` in the cloned repo is the cheapest authoritative check.
- **Leaves before intermediates.** Prioritize specs for leaf spaces of the operator DAG (DAG outputs: quantities downstream of *all* computations: κ, C_V, DOS, Grüneisen, P_3). They drive the visualization's hide-vs-dash decision and are the cross-code comparable observables. Intermediate spaces (DM, eigenvectors, MFD) are scaffolding; their adapter specs can stay implicit (dashed in the viewer) until a comparison call actually needs them.

1. Read the code's own documentation

Read in order of authoritativeness:

1. The repo's `README.md` (often documents inputs, outputs, units).
2. The user manual or formal docs site.
3. The source headers where output files are written. Search for filenames you expect to find in the docs; the surrounding code reveals what is actually being written, in what unit, and with what convention.

Catalog, in note form:

- **Every output file or returned array** the code can produce that corresponds to a quantity in the operator DAG.
- For each: its declared unit, shape, indexing (e.g. "irreducible wedge" vs "full grid"), and any qualifiers ("scattering rate" vs "linewidth" vs "imaginary self-energy", which differ by factors of 1, 2, or 4π).

2. Map outputs to operator Spaces

For each operator Space in the DAG, find the code's corresponding output (if any). Record:

operator Space	code's output	shape	per-mode / per-q / contracted

If the code does not expose a per-mode form of an Observable but only a contracted one (e.g. `BTE.cv` is volumetric J/m^{3}K, not per-mode J/K), **skip writing a `SpaceRepresentationSpec` for that state**. The skill does not silently invent missing data. Note the gap in the adapter module's docstring.

3. Extract unit and convention values

For each mapping:

- **Unit.** Check whether the unit already exists in `omai/representation/units.py`. If not, add it with a clear `to_operator` factor. Watch the canonical dimensions: heat capacity per mode (`J/K`) is *different* from volumetric heat capacity (`J/m^3K`); the latter cannot share a Unit table entry with the former.
- **Space-level normalizations.** Look at the normalization registry (`omai/representation/normalizatio` For each observable the code emits, decide which definitional choice applies and declare it via `observable_normalizations` on the spec. If the code uses a choice not in the registry, add a new named `Normalization` with its `to_operator` factor, never an ad-hoc multiplier on the adapter.
- **Schemes on producing Operators.** Each Operator declares its canonical `schemes` (e.g. `symmetry_group`, `broadening_param`, `bte_solver`). For each, decide the code's value and record overrides via `scheme_overrides`:
  - `symmetry_group`: most codes use `spglib_auto`. A code with no symmetry reduction uses `C1`.
  - `broadening_param`: `stdev` (canonical), `halfwidth`, or a code-specific scheme like `adaptive_scaled`.
  - `bte_solver`: `rta`, `direct_inverse`, or a parameterized identity realization (e.g. kaldo's `sc` is iterative SCF, the same canonical `direct_inverse`, different algorithm).
- **Discretization choices.** Diagnostic only: record on the `OperatorRepresentationSpec.discretizatio` dict. Examples: `bz_summation`, `linear_solver`, `delta_cutoff_sigmas`.

4. Write the adapter module

Create `omai/thermal_transport/representation/<code>.py` with one `SpaceRepresentationSpec` per ingested space and one `OperatorRepresentationSpec` per scheme-bearing operator. Match the existing `kaldo.py` / `phono3py.py` template:

- File docstring: code's role, links, output-file mapping table.
- One block per spec, named `<CODE>_<SPACE>` in upper snake case.
- Notes that cite the specific code API call or file used.
- Notes that name the corresponding kaldo/phono3py file/quantity so a reader can cross-check.

5. Wire it in

- Re-export in `omai/thermal_transport/representation/__init__.py`.
- Coverage is discovered automatically (any submodule of `representation/` exposing module-level spec instances is picked up by `omai.map_data.build_codes`); regenerate the map data with `python -m omai.map_data`.

6. Validate via `compare()`

Write at least three smoke tests in `tests/test_<code>.py`:

1. **Unit factor**: synthetic identical-physics data in the new code's units and a reference's units; after applying the spec-derived factor, `compare()` returns `EXPECTED_AGREE`.
2. **Convention factor**: where applicable (e.g. a $2\times$ or 4π factor between the new code and the reference), the spec captures it.
3. **HiddenSpace contraction**: per-element compare returns `NOT_COMPARABLE`; sum-contraction returns `EXPECTED_AGREE` on the reference data.

If any returns `UNEXPECTED_DISAGREE`, *do not lower rtol.* Find the missing convention.

A.2 Pitfalls observed during ingestion

These are recurring traps; check each one explicitly while writing specs.

- **Angular vs linear frequency.** rad/ps = angular_THz = $2\pi \times$ linear_THz. Codes that quote frequencies "in THz" rarely state which.
- **Compounding factors.** Watch for cases where a unit *and* a convention both differ between two codes. The cross-code factor multiplies, and the result can look like a single mystery factor. *Worked example:* ShengBTE Linewidth (angular_THz, $2 \times \mathrm{Im}\,\Sigma$) vs phono3py Linewidth (linear_THz, $1 \times \mathrm{Im}\,\Sigma$) gives a $1/(4\pi)$ factor: $1/(2\pi)$ from units and $1/2$ from convention. If you write a quick cross-code test against a reference and the residual is mysteriously off by a factor of "about 0.08" or "about 12.5", that is $1/(4\pi)$ or 4π; resist the urge to add a fudge factor, there is a convention you have not declared.
- **Linewidth vs scattering rate vs imaginary self-energy.** Three names, factors of 1, 2, and possibly 4π apart. Look for the formula $\Gamma = \ldots \mathrm{Im}\,\Sigma$ in the docs or source. Codes that emit "scattering rates in ps^{-1}" are typically expressing the same number ShengBTE does ($2 \times \mathrm{Im}\,\Sigma$ in angular frequency).

- **Per-mode vs integrated quantities.** A code may expose only the T-integrated form of an observable that the operator DAG declares per-mode. Do not fake the per-mode form. Skip the spec.
- **"Direct inverse" vs "iterative" BTE solvers.** Both can realize the canonical `bte_solver=direct_inve` identity (same fixed point, different algorithm). Distinguish on the `OperatorRepresentationSpec` via `discretization_choices`, not as a different state.
- **Default symmetry.** Most codes apply spglib reduction by default; kaldo (stable) is the exception. Record explicitly, do not assume.
- **Irreducible wedge vs full grid output.** Affects array shape; record on the SpaceRepresentationSpec note.
- **g/mol vs amu for masses, eV/Å^2 vs Ry/au^2 for force constants, nm vs Å for lattice vectors.** Common cross-code unit traps.
- **Force-constant unit conventions silently differ between codes that nominally use the same unit.** Phono3py's `fc3.npy` and ShengBTE's `FORCE_CONSTANTS_3RD` are both documented as "eV/Å^3", but ingesting phono3py's array verbatim into ShengBTE gives $\kappa \approx 100\times$ too small (the characteristic signature of FC3 values $\approx 10\times$ too large, since $\Gamma \propto |V_3|^2 \propto \text{fc3}^2$ and $\kappa \propto 1/\Gamma$). The empirical factor 0.1 reconciles them; the root cause is a per-distance vs per-displacement scaling difference in phono3py's internal representation that we have not yet fully traced through the source. **Always verify a freshly-ingested code against an already-trusted code on the same material before trusting a single-code run**, otherwise convention-mismatch errors of this magnitude can hide in plain sight. The Si-Tersoff cross-code agreement check in `experiments/silicon_shengbte/` is the worked example.

A.3 Worked example: ShengBTE ingestion (2026-05)

For a concrete walkthrough of this procedure on a real code, see `omai/thermal_transport/representation/sh` and the companion tests in `tests/test_shengbte.py`. Notable decisions made during that ingestion:

- **Skipped HeatCapacity** because ShengBTE exposes only `BTE.cv` (volumetric J/m^{3}K), not a per-mode form. Recorded the gap in the adapter docstring.
- **Added** `KM_PER_S = Unit(...)` to `units.py` with `to_operator = 10.0` (1 km/s = 10 Å · THz). This kind of one-line addition to `units.py` is routine: most codes drop a single new unit.
- `compute_force_constants_*` **is upstream.** ShengBTE reads FC2/FC3 from files written by another code (phonopy or QE for harmonic, `thirdorder.py` for cubic). The op-adapter `symmetry_group` value therefore reflects the upstream code's choice. Note this in the spec.
- $\kappa_{\text{CONV}} \neq$ **direct_inverse algorithm, but = direct_inverse canonical.** ShengBTE iterates $F = F_{\text{RTA}}$ + correction until κ stops changing; phono3py uses a LAPACK pseudo-inverse; kaldo's `inverse` uses `scipy.linalg.solve`. All three converge to the same linearized-BTE solution, so they share `bte_solver=direct_inverse`. The distinction goes on `discretization_choic` `linear_solver`.
- `broadening_param=adaptive_scaled` is a new scheme value the ShengBTE ingestion needed. (kaldo uses `halfwidth`, phono3py `stdev`; ShengBTE's adaptive Gaussian does not reduce to either.) If a new scheme *value* arises, add it as a `scheme_overrides` entry on the OperatorRepresentationSpec, no operator-layer change required. If a new scheme *name* arises, that *is* an operator-layer edit.

A.4 What this skill explicitly does not do

- It does not write code that calls the external program. Adapter specs describe outputs after the code has run; a separate ingestion function takes those outputs and wraps them as `Representation` instances.
- It does not modify the operator DAG. New nodes/edges, new schemes, or new normalizations on existing nodes are substrate work, not adapter work (see Appendix B).

B Extending the DAG

Goal. Add a new physical quantity, a new production formula for an existing quantity, or a new family of variants to the operator-layer DAG, in a way that respects the framework's commitments (one formula per edge, type discipline on spaces, no cycles) and does not pollute downstream consumers.

Prerequisites.

- The quantity you want to add fits the existing two-layer split: it is symbolic / code-independent, not a numerical artefact of one adapter.
- You have read Part III, "DAG extension rules" (the canonical statement; this appendix is the operational form).

B.1 The three patterns

Pick exactly one. They are decision-equivalent and Part III spells out the formal definitions; this appendix is the day-to-day checklist.

Pattern A: type-level parameter on the space

When the variants change **gauge type** (Observable vs HiddenSpace) and the parameterised space is **terminal** (or its parameterised consumers are themselves a closed sub-branch).

Examples in the repo:

- `MeanFreeDisplacement[bte_solver=rta]` (HiddenSpace) vs `[direct_inverse]` (Observable). Propagates to `ThermalConductivity[bte_solver=...]` and stops.
- `ThermalConductivity[transport_model=lbte|wigner|qhgk]`. All leaves.

Anti-pattern. Putting a type parameter on an *intermediate* space. Every downstream consumer then has to be parameterised in turn, and the entire downstream sub-DAG inherits the pollution. We rejected this for NAC specifically because parameterising `DynamicalMatrix[nac=...]` would have forced `compute_dispersion`, `compute_group_velocity`, `compute_anharmonic_linewidth`, `solve_bte_rta`, `solve_bte_direct`, and `contract_kappa_*` to all gain the parameter.

Pattern B: sibling spaces, converging edge

When several variants represent physically distinct contributions with **different inputs** that must **combine** before a downstream consumer uses them.

Examples in the repo:

- `AnharmonicLinewidth`, `IsotopicLinewidth`, `BoundaryLinewidth` → `sum_linewidths` → `TotalLinewidth`. `solve_bte_*` consumes only the total.
- `HelmholtzFreeEnergy`, `Entropy`, `InternalEnergy`, `HeatCapacity`: sibling Observables off `(Frequency, Temperature)`; no converging edge because there is no downstream that wants them combined, but the sibling structure is the same.

Why not Pattern A. The per-channel inputs differ (FC3+e vs IsotopeAbundances+e vs GroupVelocity), so the channels are not just parameter variants of one production formula. Sibling spaces make the different input chains visible in the DAG diagram and let each code's adapter declare independently which channels it supports.

Pattern C: shared output node, alternative producing edges

When several formulas produce the **same-typed** output with the **same gauge classification**, from different inputs. Downstream is unaware of which path was taken; provenance and adapter specs record it.

Example in the repo:

- `compute_dynamical_matrix(FC2)` → `BareDynamicalMatrix`, then either `apply_nac_correction(BareDM, BornCharges, ε∞)` → `DynamicalMatrix` (polar branch) or `identity_dm(BareDM)` → `DynamicalMatrix` (non-polar branch). `compute_dispersion` consumes only `DynamicalMatrix`.

The `BareDynamicalMatrix` intermediate is not cosmetic. A literal in-place modifier, one space with an edge that both consumes and produces itself, would create a self-loop, which the DAG validator rejects. The intermediate names the pre-modification version so the two alternative edges have a well-defined source and the converging node is a true sink of both.

B.2 Decision flow

When you have a new quantity / variant to add, walk this in order:

1. **Is it a new physical quantity** (not a variant)? Add a new space node; add one or more edges producing it; declare gauge type, dimension, conventions, indices.
2. **Is it a variant that changes the gauge type** (Observable ↔ HiddenSpace)? Pattern A: type-level parameter on the (terminal) space.
3. **Is it a variant with different inputs that physically combine** with the existing one? Pattern B: sibling state + a converging edge.
4. **Is it a same-typed variant, different production formula**? Pattern C: alternative producing edges into a shared output node, possibly with a small upstream intermediate to avoid a cycle.

If none of the above fits cleanly, the variant probably is not well-modelled as a graph operation. Reconsider whether it belongs at the operator layer at all, or whether it is a representation-layer convention.

B.3 Mechanical steps

After choosing a pattern:

1. **Operator layer.**
   - `omai/<domain>/operator/nodes.py`: declare new `Space` instances: `ObservableSpace`, or `HiddenSpace` with `kind="scaffolding"` or `"approximation"`, a `gauge_group`, and (for scaffolding) the `gauge_invariant_contractions`. Pattern A spaces carry the parameter in their name and a `labels` dict; Pattern B / C spaces have plain names. Declare `fields` with dimensions and index signatures.
   - `omai/<domain>/operator/edges.py`: declare new `Operator` instances. Each edge carries a sympy `formula` (use `auxiliary_formulas` when a kernel reappears in two edges, see `compute_anharmonic_linewidth` / `solve_bte_direct` for the $|V_3|^2$ pattern). Declare `schemes` only for choices a downstream comparison needs to distinguish.

- `omai/<domain>/operator/vocabulary.py`: register the sympy base-symbol names the new spaces' formulas carry (`register_space_symbols`) and any bare constants (`register_formula_con`) into `omai.operator.vocabulary`; the unified-validation test fails on unregistered symbols.
- `omai/<domain>/operator/__init__.py`: re-export.

2. **Adapters.** For every code that produces the new quantity, add a `SpaceRepresentationSpec` and (where the operator declares schemes) an `OperatorRepresentationSpec` in `omai/<domain>/representation/<code>.py`. If a code does not produce the quantity, do not write a placeholder spec: its absence is the signal.

3. **Tests.** Add a smoke test in `tests/test_operator.py` asserting the node and edge identities, the inputs/outputs, and any new schemes. If the new edge has a non-trivial formula, assert that `formula.free_symbols` contains the expected ingredients.

4. **Map data.** The unified map data (`docs/data/*.json`) regenerates from `python -m omai.map_data`; the site map (`docs/map/`) renders it directly. No code changes are needed unless you are adding a new pattern the map cannot lay out.

5. **Architecture reference.** Update the node / edge counts in Part III, "The operator DAG", and mention the new convention name (if any) in the relevant edge bullet.

B.4 Pitfalls

- **Cycles via in-place modifiers.** Pattern C looks like an "in-place refinement" but you must introduce the small intermediate node to keep it acyclic.
- **Cascading type parameters.** Pattern A on a non-terminal state. The validator does not flag this: discipline is on you.
- **Edge formulas that re-derive an existing kernel.** If a sub-expression appears in two edges, factor it into `auxiliary_formulas` on one of them and reference it from the other's description. The $|V_3|^2$ kernel is declared once on `compute_anharmonic_linewidth` and referenced from `solve_bte_direct`'s collision matrix.
- **Spaces with no clear gauge type.** Every Space must declare whether it is gauge-invariant (ObservableSpace) or gauge-dependent (HiddenSpace). HiddenSpaces must declare their `gauge_group` and the cross-code contractions that are gauge-invariant.

C Encoding a catalog skill

This appendix is the Plan 2 template. For every record in `omai/materials/skills_catalog.json`, follow these seven steps to produce the four artifacts that connect it to the operator/representation/instance layers. The worked example is `mat-diffusion-analysis`: nodes in `omai/materials/operator/nodes.py` (`DIFFUSIVITY_STATE`, `ACTIVATION_ENERGY`), edges in `omai/materials/operator/edges.py` (`contract_diffus` `fit_arrhenius`), and the representation spec in `omai/materials/representation/mat_diffusion_analysis.`

To read example values from catalog records you must have the AtomisticSkills repository cloned locally (it is gitignored: never commit it). The path is referenced by the `ref` field inside each `example_instances` entry.

C.1 Step 1: Produced-quantity nodes

For each entry in the record's `produces` list, determine whether an equivalent node already exists. Two nodes are equivalent when they share the same physical dimension and the same index set (and the same gauge: `observable` vs `hidden`).

Check `omai/thermal_transport/operator/nodes.py` first, then `omai/materials/operator/nodes.py`.

If an equivalent node already exists, reuse it by importing it from the file where it lives.

If no equivalent exists, add an `ObservableSpace` (gauge `observable`) or a `Space` subclass (gauge `hidden`) in `omai/materials/operator/nodes.py`, and add it to the module-level `NODES` tuple. Each node takes a `fields` tuple of one or more `Field(symbol, dimension, indices=())` items. If the physical dimension of the new field is not yet in `omai/operator/dimensions.py`, add a new `Dimension` constant there and include it in the `DIMENSIONS` dict.

Example: `mat-diffusion-analysis` produces `diffusivity` (dimension `DIFFUSIVITY = Dimension("diffusivity")`, no indices) and `activation_energy` (dimension `ENERGY`, already present). Both were new nodes, so `DIFFUSIVITY_STATE` and `ACTIVATION_ENERGY` were added to `omai/materials/operator/nodes.py`, and `DIFFUSIVITY` was added to `omai/operator/dimensions.py` with an entry in `DIMENSIONS`.

The two catalog entries `mean_square_displacement` and `room_temperature_conductivity` were intentionally not promoted to new nodes at this stage; they remain catalog-only until a future skill explicitly needs them in the operator graph.

C.2 Step 2: Consumed-quantity inputs

For each entry in the record's `consumes` list, resolve it to an existing node.

Check `omai/materials/operator/shared_primitives.py` first. That module re-exports leaf nodes from `omai/thermal_transport/operator/nodes.py` (`TEMPERATURE_STATE` as `TEMPERATURE`, `MEAN_SQUARED_DISPLACEMENT`, `TRAJECTORY`, `POTENTIAL`) and also defines `STRUCTURE` (the opaque crystal structure leaf used by most mat-* skills). Import the matching primitive from `shared_primitives` in the edges file for the new skill.

If a consumed quantity is genuinely new and will be shared by multiple future skills, add a new `ObservableSpace` to `shared_primitives.py` and append it to `SHARED_PRIMITIVES`. If it is unique to this skill and has already been added as a produced node in step 1, reference that node directly.

Example: `mat-diffusion-analysis` consumes `md_trajectory` (mapped to `MEAN_SQUARED_DISPLACEMENT`, which is imported from `shared_primitives`) and `temperature` (mapped to `TEMPERATURE`, also from `shared_primitives`).

C.3 Step 3: Operator edges

Add one `Operator` instance per operation in `omai/materials/operator/edges.py` (importing from `omai.operator.operator`). Append every new operator to the module-level `EDGES` tuple.

Attach a `sympy.Eq` as the `formula` when `operation.closed_form` in the catalog record is a concrete algebraic expression. For procedural operations (regression fits, ML inference, iterative solvers) also attach a sympy `Eq` showing the governing relation, but set `is_executable_in_sympy_override=Fal` to indicate that sympy cannot evaluate it directly.

Example: `contract_diffusivity` carries a closed-form sympy `Eq` for the Einstein relation `D = slope(MSD(t)) / (2 d)`. `fit_arrhenius` carries the Arrhenius equation as a sympy `Eq` but sets `is_executable_in_sympy_override=False` because the activation energy is extracted by a weighted regression over multiple temperature points, not by algebraic inversion of a single equation.

Register every symbol the new formulas reference in `omai/materials/operator/vocabulary.py`: the per-node sympy base names via `register_space_symbols` (keyed by node name) and any bare fit constants (like `d` or `D_0`) via `register_formula_constants`. `validate_dag` flags unregistered symbols as "not derivable from inputs", and `tests/test_unified_validation.py` enforces a clean unified map, so a missing registration fails the suite.

C.4 Step 4: Representation spec

Create `omai/materials/representation/<module_name>.py` where `<module_name>` is the catalog skill name with hyphens replaced by underscores (e.g. `mat_diffusion_analysis`). The file must define one `SpaceRepresentationSpec` per produced node that has a counterpart in the operator graph (step 1 nodes only; skip catalog-only quantities).

Each spec takes:

- `space`: the node object from step 1.
- `representation_name`: the catalog skill name with hyphens, exactly as it appears in the catalog (e.g. `"mat-diffusion-analysis"`).
- `observable_units`: a dict mapping field symbol to a display unit string (e.g. {`"D": "cm^2/s"`}). These strings are consumed by `build_codes` as display labels; they are not registered in `omai/representation/units.py` unless a numerical unit-conversion test actually requires a registered `Unit` object for that dimension.
- `code_api`: a dict mapping field symbol to the path of the script or entry point that computes it (e.g. {`"D": "scripts/analyze_diffusion.py"`}).
- `notes`: a short free-text description of the computational method.

Do not register new entries in `omai/representation/units.py` for display-only units. The `units.py` registry is for the executor and comparison layers, which need `conversion_factor` between registered units. Adding `cm^2/s` or `eV` to the registry is only necessary when a test exercises `conversion_factor` or `dimension_si_scale` for those units.

Example: `mat_diffusion_analysis.py` defines `DIFFUSION_DIFFUSIVITY` (for `DIFFUSIVITY_STATE`) and `DIFFUSION_ACTIVATION` (for `ACTIVATION_ENERGY`), both with `representation_name="mat-diffusion-ana` No new entries were added to `omai/representation/units.py`.

C.5 Step 5: Instance records

For each value in the catalog record's `example_instances` list, call `omai.map_data.record_instance` with `domains=map_data.DOMAINS`. Pass only real numbers from the catalog: never invent or interpolate values. If the record has no numeric `example_instances`, skip this step.

The `variable` argument must match the `name` of a node in the operator graph (the `id` field as returned by `build_graph_dict`). The `source_ref` string becomes part of the output filename; keep it short and descriptive.

Call `record_instance` from a script or a one-off Python session; the call writes a JSON file under `docs/data/instances/`.

Example: the LGPS activation energy (0.152 eV) from the catalog was written as `docs/data/instances/lgps-activationenergy-atomisticskills-mat-diffusion-analysis-lgps.json` with `variable="Activat` `source_kind="simulation"`, and `conditions={"T_range": "600-1000K", "potential": "MatPES-r2SCA` The room-temperature conductivity (91.44 mS/cm) was not recorded because `RoomTemperatureConductivity` is a catalog-only quantity with no operator node.

C.6 Step 6: SYMBOLS entry

Add the LaTeX symbol for each new operator-graph node to the `SYMBOLS` dict in `omai/materials/domain.py`. The key is the node's `name` string; the value is the LaTeX symbol used in graph visualizations and the site deck.

Example: after adding `DIFFUSIVITY_STATE` and `ACTIVATION_ENERGY`, the dict received `"Diffusivity": r"D"` and `"ActivationEnergy": r"E_a"`. Add a node's symbol only once the node actually enters the graph (i.e. an edge in `EDGES` produces or consumes it); `STRUCTURE` is defined in `shared_primitives` but stays out of `SYMBOLS` until the first skill wires an edge to it.

C.7 Step 7: Regenerate and test

After completing steps 1 through 6, regenerate the site data files and run the test suite:

```
PYTHONPATH=. python -m omai.map_data
python -m pytest tests/test_materials_data.py
```

`python -m omai.map_data` rewrites the generated data files (`graph.json`, `instances.json`, `codes.json`, `catalog.json`, `version.json`). Commit the regenerated files alongside any content change; the site map renders them directly.

The test suite checks that all domain imports succeed, that the new nodes and edges appear in the unified graph, that the representation spec is discoverable via `build_codes`, and that `record_instance` validates the variable name against the live node set.