Opoch Kernel Specification

Protein Structure Inference

Technical Reference · Version 1.0

CONTRACT

"If I speak, I have proof. If I cannot prove, I return the exact boundary."

New to Opoch? Read the Kernel Primer

OPOCH

www.opoch.com

Index

0. Executive Summary

The Problem

Protein structure prediction attempts to guess the native 3D conformation from amino acid sequence. Current methods (including AlphaFold) produce predictions without certificates of correctness or explicit handling of degeneracy.

The Solution

We compile protein folding into Opoch's Kernel Interface—a mathematical framework that transforms the problem from "prediction" into "quotient collapse under pinned tests." This provides:

- **Verifiable correctness**: Every solution is cryptographically receipted and polynomial-time checkable
- **Certified optimization**: Branch-and-bound with LB/UB certificates proves optimality
- **Honest ambiguity**: Multiple valid folds (OMEGA_MULTI) or resource limits (OMEGA_GAP) are explicitly reported
- **Deterministic outputs**: Same inputs always produce same outputs with replayable receipts

Key Results

The kernel approach guarantees:

- **Soundness**: Only verifier-pass structures are returned
- **Completeness**: All admissible conformations are reachable given sufficient budget
- **No false positives**: UNIQUE requires certified closure
- **No false negatives**: UNSAT requires explicit impossibility proof

Core Insight

The verifier is truth. Everything else is proposal. By separating the verifier (source of truth) from solvers (proposal mechanisms), we transform protein folding from guessing to proving.

1.1 Kernel Statement

Protein folding is not prediction. It is quotient-collapse under pinned tests. We compile protein structure inference into kernel primitives: possibility space, tests, truth quotient, frontier, and forced separator.

1.1.1 Possibility Space W

Definition: W

W = set of all conformations X admissible under hard chemistry constraints for sequence S (and optional evidence ledger L). Each X encodes 3D atom coordinates or internal coordinates.

1.1.2 Tests Delta

Each verifier check is a finite, decidable test:

- **GEOM test**: bond lengths/angles/chirality/planarity within tolerance
- **STERIC test**: no forbidden overlaps (hard exclusion radius)
- **LEDGER test**: each evidence item (restraint/density/contact) within tolerance
- **OBJECTIVE test**: energy/objective evaluation and bound certificates

1.1.3 Truth Pi (Quotient)

Definition: Pi

Truth is defined modulo gauge/representation slack:
- Global rigid motions (translation + rotation) are gauge
- Equivalent coordinate parameterizations are gauge

Two conformations are equivalent if all feasible tests agree (and differ only by gauge). Pi collapses minted distinctions.

1.1.4 Omega Frontier

Definition: Omega

Omega is NOT guessing. It is one of two forms:
- **OMEGA_MULTI**: multiple distinct optimal basins survive (true degeneracy under verifier)
- **OMEGA_GAP**: undecided under budget; return frontier basins + best lower bound + next test

1.1.5 tau* (Forced Separator)

Definition: tau*

tau* is the deterministic next distinguisher:
- For optimization: the next split variable/region that maximally tightens bounds
- For ambiguity: the cheapest missing evidence test that separates top competing basins

Tie-break must be Pi-invariant (canonical ordering).

Key Insight

Branch-and-bound on conformational regions IS the kernel refinement algorithm: split on tau* (best bounding variable), prune by certificates, repeat until UNIQUE (certified optimum) or Omega (degeneracy or budget limit).

This kernel view transforms protein folding from "guessing structures" to "collapsing possibility space under verifiable tests."

The verifier is truth. Everything else is proposal.

1.2 Problem Definition

Find the optimal protein conformation(s) for a given sequence under specified constraints.

1.2.1 Inputs

```
S = (s_1, ..., s_L)   Amino-acid sequence of length L
C = conditions        Solvent, temperature, ionic strength, partners, etc.
L = evidence ledger   Optional: constraints/tests with declared tolerances
epsilon = tolerance   Optimality certificate precision
delta = ensemble tol  For credible sets (optional)
```

1.2.2 Conformation Space (Admissible Geometry)

```
Let X encode 3D coordinates of atoms (or internal coordinates)
subject to hard constraints:
  - Covalent geometry constraints (bond lengths, angles)
  - Chirality constraints (L-amino acids, proline rings)
  - Steric exclusion constraints (van der Waals radii)

Denote admissible set: Omega(S) subset of R^n
```

1.2.3 Objective (Pinned Contract)

```
Define the objective energy E(X) as a declared, total function:

  E: Omega(S) -> R

IMPORTANT: "Complete" requires E be pinned (versioned) and replayable.
Optionally incorporate evidence as hard constraints or penalty terms.
```

Two canonical query types:

- **(A) MAP** (single best structure): Find X* in argmin_{X in Omega(S) ∩ L} E(X)
- **(B) Certified ensemble**: Return set of basins whose total certified mass >= 1-delta

1.2.4 Output Contract

Every query terminates in exactly one of four states:

- **UNIQUE**: single certified optimum class (up to gauge) + receipt
- **OMEGA_MULTI**: multiple certified optimum classes survive (degeneracy) + receipt
- **UNSAT**: no X satisfies hard constraints + ledger (infeasible) + certificate
- **OMEGA_GAP**: undecided under compute budget; frontier basins + bounds + next distinguisher

```
Output witness W = {
  sequence:    S
  conditions:  C
  ledger:      L
  structure:   X* (or basins for OMEGA_MULTI)
  energy:      E(X*)
  certificate: LB/UB closure proof
  verifier:    PASS
  receipt:     SHA256(canonical(W))
}
```

Note: Gauge Invariance

UNIQUE means unique up to gauge (rigid body motion). Two structures related by translation and rotation are the same answer. The canonical form centers at centroid and aligns principal axes.

1.3 Truth Gate (Verifier)

The verifier is the SOURCE OF TRUTH. Branch-and-bound, neural networks, and all other solvers are proposal mechanisms. Only the verifier determines validity.

1.3.1 Verification Checks

Check	Condition	On Failure
V1: Geometry	bonds/angles within tol	first violation witness
V2: Sterics	no forbidden overlaps	violating atom pair
V3: Ledger	all evidence constraints	first failing constraint
V4: Objective	compute E(X) exactly	pinned spec result
V5: Certificate	LB/UB consistency	bound violation witness

Detailed Check Specifications

V1: Geometry Check

```
def check_geometry(X, S, tolerances):
    for bond in covalent_bonds(S):
        d = distance(X[bond.atom1], X[bond.atom2])
        if abs(d - bond.ideal_length) > tolerances.bond:
            return FAIL(witness=("BOND", bond, d, bond.ideal_length))

    for angle in bond_angles(S):
        theta = compute_angle(X[angle.a1], X[angle.a2], X[angle.a3])
        if abs(theta - angle.ideal) > tolerances.angle:
            return FAIL(witness=("ANGLE", angle, theta, angle.ideal))

    # Check chirality, planarity...
    return PASS
```

V2: Steric Check

```
def check_sterics(X, radii, tolerance):
    for i, j in all_nonbonded_pairs(X):
        d = distance(X[i], X[j])
        min_d = radii[i] + radii[j] - tolerance
        if d < min_d:
            return FAIL(witness=("CLASH", i, j, d, min_d))
    return PASS
```

1.3.2 Verifier Theorems

Theorem 3.1: Verifier Soundness

If verify(X) = PASS, then X is admissible and satisfies the contract. Proof: The verifier checks all necessary conditions: geometry (V1), sterics (V2), ledger constraints (V3), objective computation (V4), and certificate validity (V5). Each check is deterministic with explicit tolerance. PASS requires all checks pass. QED.

Theorem 3.2: Verifier Completeness

If X satisfies the contract, then verify(X) = PASS. Proof: The verifier checks are exactly the defining conditions of an admissible conformation under the pinned specification. Any X satisfying all conditions will pass all checks. QED.

Theorem 3.3: Minimal Separator Property

If verify(X) = FAIL, the returned violation is a minimal separator witness. Proof: Each check returns the first violation found (deterministic ordering). This violation is sufficient to prove X is not admissible (single constraint failure). It is minimal in the sense that removing it would require re-checking. QED.

Implementation Note

The verifier must be deterministic and reproducible. Same inputs must produce identical outputs across implementations. Use exact arithmetic where possible, or document numerical tolerances precisely.

2. Method Overview

We solve protein folding because we understand its structural reality: it is not "prediction"; it is quotient-collapse under pinned tests.

Core Principles

- **Verifier defines reality**: admissible conformations are Pi-classes that PASS
- **Ambiguity is Omega**: if multiple basins survive, output them or the missing distinguisher
- **Certified optimization replaces guessing**: bounds are the only truthful proof of optimality
- **Receipts make refinements reusable**: cost falls with use (compounding intelligence)

Why This Works

Traditional protein structure prediction attempts to "guess" the native state. This is fundamentally wrong. The kernel approach instead:

1. **Pins the contract**: E(X) is versioned and reproducible
2. **Defines truth via verifier**: no hand-waving about "close enough"
3. **Uses certified optimization**: bounds prove optimality
4. **Handles ambiguity honestly**: OMEGA_MULTI for true degeneracy
5. **Compounds intelligence**: every solved problem accelerates future queries

Key Insight

The protein folding "problem" is not about predicting nature. It is about finding conformations that minimize a pinned energy function under verifiable constraints. Nature uses different physics than our models. Our job is to solve OUR model exactly, then improve the model based on experimental feedback.

3. Exact Algorithms

3.1 Branch-and-Bound

The core algorithm for finding certified optimal conformations.

Key Object: Region R

```
Region R subset Omega(S) described by:
  - Bounded internal coordinate ranges (phi, psi, chi angles)
  - Discrete rotamer sets (if used)
  - Additional constraint propagation results

Required primitives (declared once):
  - Bound(R): returns (LB(R), UB(R)) such that
      LB(R) <= inf_{X in R} E(X) <= UB(R)
  - Feasible(R): returns whether R contains any admissible X
      (or gives UNSAT witness if impossible inside R)
```

Branch-and-Bound Algorithm

```
def branch_and_bound(S, L, E, epsilon, max_nodes):
    # Initialize with full conformation space
    R0 = initial_region(S)
    queue = PriorityQueue()  # by LB
    queue.push((Bound(R0).LB, R0))

    X_best = None
    UB_best = infinity
    nodes_expanded = 0

    while not queue.empty():
        LB_R, R = queue.pop()
        nodes_expanded += 1

        # Pruning: if this region can't improve, skip
        if LB_R >= UB_best - epsilon:
            continue

        # Check feasibility
        feas = Feasible(R, L)
        if feas == UNSAT:
            continue  # Prune infeasible region

        # Try to find a feasible point in R
        X_sample = sample_feasible(R, L)
        if X_sample is not None:
            E_sample = E(X_sample)
            if E_sample < UB_best:
                X_best = X_sample
                UB_best = E_sample
```

```
            # Split region by tau* rule
            tau_star = select_split_variable(R)
            R1, R2 = split_region(R, tau_star)

            for Ri in [R1, R2]:
                LB_i, UB_i = Bound(Ri)
                if LB_i < UB_best - epsilon:
                    queue.push((LB_i, Ri))

            # Check node budget
            if nodes_expanded >= max_nodes:
                return OMEGA_GAP(
                    frontier=queue.top_regions(k=10),
                    best_LB=queue.min_LB(),
                    best_UB=UB_best,
                    best_X=X_best,
                    tau_star=select_next_split(queue)
                )

        # Queue exhausted
        if X_best is None:
            return UNSAT(reason="No feasible conformation found")

        # Check for degeneracy (multiple optimal basins)
        optimal_basins = find_basins_within_epsilon(X_best, epsilon)
        if len(optimal_basins) > 1:
            return OMEGA_MULTI(basins=optimal_basins, energy=UB_best)

        return UNIQUE(
            structure=X_best,
            energy=UB_best,
            certificate=(global_LB, UB_best),
            receipt=SHA256(canonical(X_best, UB_best))
        )
```

3.2 Feasibility Oracle (UNSAT Certificates)

When ledger constraints are inconsistent with hard chemistry:

```
def feasibility_oracle(R, L, S):
    # Check geometry constraints
    geom_result = check_geometry_feasibility(R, S)
    if geom_result == UNSAT:
        return UNSAT(witness=geom_result.witness)

    # Check steric constraints
    steric_result = check_steric_feasibility(R)
    if steric_result == UNSAT:
        return UNSAT(witness=steric_result.witness)

    # Check ledger constraints
    for constraint in L:
        if not satisfiable_in_region(R, constraint):
            return UNSAT(witness=("LEDGER", constraint))

    return FEASIBLE
```

Examples of UNSAT certificates:

- Impossible distance restraints (mutually exclusive)
- Steric impossibility (atoms forced to overlap)
- Contradictory geometry constraints

3.3 Algorithm Theorems

Theorem 5.1: Branch-and-Bound Soundness

If B&B; returns UNIQUE(X*, E*), then X* is a verified optimum within epsilon. Proof: UNIQUE is only returned when: (1) verify(X*) = PASS, (2) E(X*) = E*, (3) global_LB >= E* - epsilon (certified closure). By (3), no conformation can have energy more than epsilon better than X*. QED.

Theorem 5.2: Branch-and-Bound Completeness

If a feasible conformation exists, B&B; will find it (given sufficient budget). Proof: The initial region R0 contains all admissible conformations. Splitting is complete (every point is in some leaf region). Feasibility oracle is sound. Therefore, if X exists in Omega(S) ∩ L, it will be sampled. QED.

Theorem 5.3: Optimality Certificate

When B&B; returns UNIQUE with certificate (LB, UB), the solution is epsilon-optimal. Proof: The certificate states LB <= optimal_energy <= UB. Since UB = E(X_best) and LB >= UB - epsilon (closure condition), we have optimal_energy >= E(X_best) - epsilon. Therefore X_best is within epsilon of optimal. QED.

Theorem 5.4: Branch-and-Bound Termination

Given finite resolution delta on coordinates, B&B; always terminates. Proof: Each split reduces region volume by at least factor 1/2. With minimum coordinate resolution delta, maximum tree depth is O(n * log(diam/delta)) where n is dimension and diam is initial diameter. Finite depth implies finite tree. QED.

4. Failure Modes and Bounded Outputs

Omega represents what we do not know, structured precisely.

UNSAT (Proven Infeasible)

UNSAT Certificate

No admissible conformation satisfies hard constraints + ledger.
Certificate must be explicit:
- Geometry violation: impossible bond/angle combination
- Steric impossibility: forced overlap
- Ledger contradiction: mutually exclusive constraints

```
UNSAT = {
    status: "UNSAT",
    certificate: {
        type: "LEDGER_CONTRADICTION",
        constraints: [c1, c2, c3],
        reason: "Distance d(A,B) < 3A and d(A,B) > 5A cannot both hold"
    }
}
```

OMEGA_MULTI (Degeneracy)

OMEGA_MULTI

Multiple optimal basins survive even with closed certificates.
This is not undecided; it is the correct "many answers" truth.
Returns all epsilon-optimal basins with their structures and energies.

```
OMEGA_MULTI = {
    status: "OMEGA_MULTI",
    basins: [
        {structure: X1, energy: E1, basin_id: "fold_A"},
        {structure: X2, energy: E2, basin_id: "fold_B"}
    ],
    certificate: "All basins within epsilon of global minimum",
    receipt: SHA256(canonical(basins))
}
```

OMEGA_GAP (Undecided Under Budget)

OMEGA_GAP

Budget/time/memory limit hit before certificates close.
Returns structured frontier: remaining regions, bounds, next split.

```
OMEGA_GAP = {
    status: "OMEGA_GAP",
    frontier: [
        {region: R1, LB: lb1, UB: ub1},
        {region: R2, LB: lb2, UB: ub2}
    ],
    global_LB: best_lower_bound,
    global_UB: best_upper_bound,
    best_X: current_best_structure,
    tau_star: {variable: "phi_42", split_point: -60.0}
}
```

5. Correctness and Guarantees

Summary of Guarantees

Property	Guarantee
Soundness	Only verifier-pass structures returned
Completeness	All admissible solutions reachable
Optimality	Certified epsilon-optimality via LB/UB closure
Honest Omega	UNSAT / OMEGA_MULTI / OMEGA_GAP never conflated
Verifiable	Any output replayable with receipts
Deterministic	Same inputs always produce same outputs

Formal Properties

Property: No False Positives

The system never claims a structure is optimal when it is not. Proof: UNIQUE requires certified closure (LB >= UB - epsilon). Until closure is achieved, the system returns OMEGA_GAP. QED.

Property: No False Negatives

The system never claims UNSAT when a feasible solution exists. Proof: UNSAT requires explicit certificate (geometry/steric/ledger impossibility). Without such proof, the system returns OMEGA_GAP. QED.

Property: Honest Degeneracy

When multiple equally-good solutions exist, the system reports all of them. Proof: OMEGA_MULTI is returned when multiple basins survive within epsilon. The system does not arbitrarily pick one. QED.

Key Point

These guarantees hold because the verifier is the source of truth, not any particular solver. The solver proposes, the verifier disposes. Certificates prove claims; absence of certificate means OMEGA.

6. Implementation Playbook

Follow these steps in order for a correct implementation.

Step 0: Pin the Contract

- Version the energy function E(X) with exact specification
- Version all tolerances (bond lengths, angles, steric radii)
- Version the ledger schema (restraint types, tolerance semantics)
- Document coordinate system and units (Angstroms, degrees)

Step 1: Implement the Verifier First

- GEOM check: bond lengths, angles, chirality, planarity
- STERIC check: van der Waals exclusion
- LEDGER check: evidence constraint satisfaction
- Return minimal witness on FAIL
- Test verifier independently before any solver code

Step 2: Implement Canonicalization (Pi)

- Normalize rigid motions: center at centroid, align principal axes
- Canonicalize internal coordinate representation
- Canonicalize evidence ordering and units
- Ensure deterministic output for equivalent inputs

Step 3: Implement Bound(R)

- Interval bounds on internal coordinates
- Convex relaxations for energy terms
- Certified steric pruning
- Must guarantee: LB(R) <= inf_{X in R} E(X) <= UB(R)

Step 4: Implement Region Splitting tau*

- Deterministic choice of variable + split point
- Maximize expected bound tightening
- Pi-invariant tie-break (canonical ordering)
- Document splitting strategy precisely

Step 5: Implement Branch-and-Bound Engine

- Priority queue ordered by lower bound
- Prune regions where LB >= UB_best - epsilon
- Track best feasible point found
- Terminate on closure or budget exhaustion

Step 6: Implement Omega Objects

- UNSAT: explicit certificate of infeasibility
- OMEGA_MULTI: list of epsilon-optimal basins
- OMEGA_GAP: frontier regions + bounds + tau*
- Never return bare "failed" without explanation

Step 7: Implement Receipts

Every solution generates a deterministic, implementation-independent receipt:

```
def generate_receipt(structure, energy, sequence):
    witness = {
        "sequence": sequence,
        "coordinates": canonicalize(structure),
        "energy": energy,
        "verifier": "PASS"
    }
    # NO timestamps in hashed payload!
    canonical = json.dumps(witness, sort_keys=True, separators=(",", ":"))
    return hashlib.sha256(canonical.encode()).hexdigest()
```

Step 8: Self-Improvement (Lemmas)

- Store pruning lemmas: (motif pattern, steric certificate)
- Store constraint implications discovered during search
- Store basin signatures for common folds
- Reapply as derived tests in future runs
- System gets faster over time on similar sequences

7. Optional Extensions

Complexes and Multimers

Extend to multi-chain systems:

- Model as coupled conformation spaces: X = (X_A, X_B, ...)
- Add inter-chain steric constraints
- Add interface constraints from evidence (cross-links, contacts)
- Same B&B; framework with expanded constraint set

Kinetics and Folding Pathways

Extend to path-space inference:

- Model as sequence of conformations: P = (X_0, X_1, ..., X_T)
- Add transition constraints (local moves, energy barriers)
- Add barrier certificates (saddle point bounds)
- Enables folding pathway inference, not just endpoint

Active Learning for Experimental Design

Use tau* to guide experiments:

- tau* identifies the test that would most reduce uncertainty
- Translate to experimental measurement (NMR restraint, cross-link)
- Enables optimal experimental design under budget
- Each experiment collapses Omega frontier maximally

Cross-Check Solvers

Optional independent verification:

- Discretized coarse-grain ILP/CP-SAT model
- Lattice protein models for sanity checks
- Multiple energy function implementations
- Cross-validation strengthens receipts

Appendix A: Sample Verification Output

PASS Example

```
Input: Structure X for sequence "ACDEFGHIK"

Verification Results:
  V1 Geometry: PASS
    - All 8 peptide bonds within 0.02A of 1.33A
    - All 16 backbone angles within 3 degrees of ideal
    - All chirality centers correct (L-amino acids)

  V2 Sterics: PASS
    - No atom pairs closer than sum of vdW radii - 0.4A
    - Checked 435 non-bonded pairs

  V3 Ledger: PASS
    - NOE restraint d(Ala1.CA, Lys9.CA) = 8.2A (limit: <10A)
    - All 12 distance restraints satisfied

  V4 Objective: E(X) = -142.3 kcal/mol

  V5 Certificate: PASS
    - Global LB = -145.1 kcal/mol
    - Gap = 2.8 kcal/mol < epsilon = 5.0 kcal/mol

Final: PASS
Receipt: a3f8c2d1e9b7...
```

FAIL Example

```
Input: Structure X for sequence "ACDEFGHIK"

Verification Results:
  V1 Geometry: PASS
  V2 Sterics: FAIL
    - CLASH detected: Phe5.CE1 -- Ile8.CD1
    - Distance: 2.1A
    - Minimum allowed: 3.0A (vdW sum - tolerance)

Final: FAIL
Witness: ("CLASH", "Phe5.CE1", "Ile8.CD1", 2.1, 3.0)
```

Appendix B: Reference Implementation Skeleton

```python
# alphaprotein_kernel.py - Reference Implementation Skeleton

import hashlib
import json
from dataclasses import dataclass
from typing import List, Tuple, Optional, Dict, Any
from enum import Enum

class Status(Enum):
    UNIQUE = "UNIQUE"
    OMEGA_MULTI = "OMEGA_MULTI"
    OMEGA_GAP = "OMEGA_GAP"
    UNSAT = "UNSAT"

@dataclass
class Conformation:
    """3D structure representation."""
    sequence: str
    coordinates: List[Tuple[float, float, float]]  # Per-atom coords

    def canonicalize(self) -> 'Conformation':
        """Center at centroid, align principal axes."""
        # Implementation: SVD alignment
        pass

@dataclass
class Region:
    """Bounded region in conformation space."""
    phi_bounds: List[Tuple[float, float]]
    psi_bounds: List[Tuple[float, float]]
    chi_bounds: List[List[Tuple[float, float]]]

    def split(self, variable: str, point: float) -> Tuple['Region', 'Region']:
        """Split region at given variable and point."""
        pass

@dataclass
class VerifierResult:
    status: str  # "PASS" or "FAIL"
    witness: Optional[Any] = None
    energy: Optional[float] = None
```

```
class Verifier:
    """Truth gate - source of all validity."""

    def __init__(self, tolerances: Dict[str, float]):
        self.tol = tolerances

    def verify(self, X: Conformation, ledger: List[Any]) -> VerifierResult:
        # V1: Geometry
        geom = self._check_geometry(X)
        if geom.status == "FAIL":
            return geom

        # V2: Sterics
        steric = self._check_sterics(X)
        if steric.status == "FAIL":
            return steric

        # V3: Ledger
        ledger_result = self._check_ledger(X, ledger)
        if ledger_result.status == "FAIL":
            return ledger_result

        # V4: Objective
        energy = self._compute_energy(X)

        return VerifierResult(status="PASS", energy=energy)


class BranchAndBound:
    """Certified global optimizer."""

    def solve(self, sequence: str, ledger: List, epsilon: float,
              max_nodes: int) -> Dict:
        R0 = self._initial_region(sequence)
        queue = [(self._bound(R0)[0], R0)]
        X_best, UB_best = None, float('inf')
        nodes = 0

        while queue and nodes < max_nodes:
            LB_R, R = heapq.heappop(queue)
            nodes += 1

            if LB_R >= UB_best - epsilon:
                continue

            # ... (rest of B&B logic)

        # Return appropriate status
        pass
```

Appendix C: Production Checklist

Pre-Deployment Checklist

- Energy function E(X) pinned with version hash
- All tolerances documented and versioned
- Verifier passes independent test suite
- Canonicalization produces deterministic output
- Bound(R) provably correct (LB <= true <= UB)
- Receipt generation is timestamp-free
- All four output states implemented (UNIQUE/OMEGA_MULTI/OMEGA_GAP/UNSAT)

Monitoring Checklist

- Track solve time distribution
- Track node count distribution
- Track certificate gap at termination
- Track OMEGA_GAP frequency (indicates budget issues)
- Track OMEGA_MULTI frequency (indicates model degeneracy)
- Log all receipts for audit trail

Safety Checklist

- Never claim UNIQUE without certificate closure
- Never claim UNSAT without explicit proof
- Always return structured OMEGA on budget exhaustion
- Validate all inputs before processing
- Bound memory usage to prevent OOM
- Implement graceful timeout handling

Test Suite Requirements

Test	Description	Expected
Small peptide	Ala-Gly-Ala trivial	UNIQUE
Constrained	With NOE restraints	UNIQUE
Degenerate	Symmetric sequence	OMEGA_MULTI
Infeasible	Contradictory restraints	UNSAT
Large	Budget exhaustion	OMEGA_GAP