PRD 29: Chemical Reaction Networks and the Origin of Life

Pure Thought AI Challenge 29

Pure Thought AI Challenges Project

January 18, 2026

Abstract

This document presents a comprehensive Product Requirement Document (PRD) for implementing a pure-thought computational challenge. The problem can be tackled using only symbolic mathematics, exact arithmetic, and fresh code—no experimental data or materials databases required until final verification. All results must be accompanied by machine-checkable certificates.

Contents

Domain: Biology Systems Chemistry
Timeline: 6-9 months
Difficulty: High
Prerequisites: Chemical kinetics, graph theory, dynamical systems, information theory, thermodynamics

---

0.1 1. Problem Statement

0.1.1 Scientific Context

The **origin of life** is one of science's deepest mysteries: how did non-living chemistry give rise to self-replicating, evolving systems capable of Darwinian evolution? A leading hypothesis posits that life emerged from **autocatalytic reaction networks**—collections of molecules that catalyze their own production from simple precursors. This framework, pioneered by Stuart Kauffman, Manfred Eigen, and others, treats the origin of life as a phase transition in chemical space.

Key concepts:

- **Autocatalytic sets**: Chemical reaction networks where every reaction is catalyzed by molecules within the set, and all molecules can be produced from a simple "food set" of externally supplied compounds
- **RAF sets** (Reflexively Autocatalytic and Food-generated): Formal mathematical definition requiring closure and catalytic completeness
- **Hypercycles**: Eigen's model of catalytic cycles forming stable coexistence states
- **Molecular evolution**: Once autocatalytic sets emerge, natural selection operates on variation and heredity

The formose reaction (autocatalytic synthesis of sugars from formaldehyde) and the metabolic pathways of modern cells provide empirical evidence for autocatalytic chemistry's central role in biology.

0.1.2 Core Question

Can we algorithmically detect minimal autocatalytic sets in realistic chemical reaction networks and characterize their emergence from combinatorial chemistry?

Key challenges:

- **RAF detection**: Given a network with 100+ species and 1000+ reactions, find all minimal autocatalytic subsets (NP-hard in general)
- **Catalytic closure**: Verify that every reaction has a catalyst from within the set
- **Food-generation**: Prove all molecules are reachable from the food set via catalyzed reactions
- **Thermodynamic viability**: Check $G < 0$ for all reactions under given conditions
- **Dynamical stability**: Verify hypercycle coexistence against parasites and fluctuations
- **Information content**: Quantify emergence of complexity via Shannon entropy and mutual information

0.1.3 Why This Matters

- **Origin of life**: Provides testable hypotheses for abiogenesis in primordial soup or hydrothermal vents
- **Synthetic biology**: Guide design of minimal autocatalytic reaction sets for artificial cells
- **Astrobiology**: Predict probability of life emergence on exoplanets with different chemistries
- **Evolutionary theory**: Understand pre-Darwinian selection at the chemical level
- **Complex systems**: General principles for emergence of self-organization

0.1.4 Pure Thought Advantages

Autocatalytic sets are **ideal for pure thought investigation**:

- Based on **graph algorithms** (reachability, closure)
- Thermodynamics computable from **standard free energies** (database lookup)
- Dynamics solvable via **ODE integration** (deterministic kinetics)
- Information theory **exact** (Shannon entropy formulas)
- All results **certified via symbolic computation**
- NO wet lab experiments until validation phase
- NO empirical reaction rate measurements initially

0.2 2. Mathematical Formulation

0.2.1 Chemical Reaction Networks

Reaction network: Tuple (S, R, C, F) where:

- S = s,...,s: species (molecules)
- R = r,...,r: reactions r: as → bs
- C: S × R → 0,1: catalysis relation (C(s,r) = 1 if s catalyzes r)
- F S: food set (externally supplied)

Reaction graph: Directed hypergraph where nodes are species, hyperedges are reactions.
Stoichiometry matrix: where = b - a (net production of species i in reaction j).

0.2.2 RAF Sets (Hordijk Steel)

Definition: Subset R' R is a RAF set if:

- **Reflexively Autocatalytic**: r R', s $\mathrm{cl}_F(R')suchthatC(s,r) = 1$

(Every reaction is catalyzed by something producible in the set)

- **Food-generated**: All reactants in R' are either in F or produced by reactions in R'

$\mathrm{cl}_F(R') = closureofFunderreactionsinR'$
Minimal RAF (maxRAF): A RAF set with no proper subset that is also RAF.
Theorem (Hordijk-Steel 2004): For random catalytic networks with sufficient connectivity, RAF sets emerge with high probability above a critical complexity threshold.

0.2.3 Hypercycles (Eigen Schuster)

Hypercycle: n species forming catalytic cycle: s → s → ... → s → s where s catalyzes production of s.
Dynamics:

```
d x /dt =      k      xx       -   d  x   -    x
```

where = kxx (selection flux).
Coexistence condition: All species maintain positive concentration at steady state.
Theorem (Eigen): Hypercycles are stable against competitive exclusion if cycle length n 5.

0.2.4 Thermodynamic Constraints

Gibbs free energy: For reaction aA + bB → cC + dD,

```
G  =   G   + RT ln([C]   [D]    / [A]   [B]   )
```

Thermodynamic viability: Reaction proceeds forward if G < 0.
Constraint on RAF: All reactions in set must satisfy G < 0 under specified concentrations.

0.2.5 Information Theory

Shannon entropy: H(X) = - p log p (bits) where p = concentration of species i.
Mutual information: I(X;Y) = H(X) + H(Y) - H(X,Y) quantifies correlation between species.
Information emergence: RAF sets exhibit I(X;Y) > 0, indicating functional relationships.

0.2.6 Certificates

All results must come with **machine-checkable certificates**:

- **RAF certificate**: Reachability graph proving all molecules producible from food
- **Catalytic closure certificate**: Witness s $\mathrm{cl}_F(R')foreachreactionr$

- **Thermodynamic certificate**: G < 0 verified for all reactions
- **Stability certificate**: Jacobian eigenvalues of steady state all negative

Export format: JSON with reaction network and RAF set:

```
{
  "network": {"species": ["A", "B", "C"], "reactions": [...], "food":
     ["A"]},
  "raf_set": {"reactions": [0, 2, 5], "species": ["A", "B", "C"]},
  "is_minimal": true,
  "thermodynamically_viable": true,
  "hypercycle_stable": true
}
```

0.3 3. Implementation Approach

0.3.1 Phase 1 (Months 1-2): RAF Detection Algorithms

Goal: Implement efficient RAF detection for networks with 50-100 species.

```
import networkx as nx
from typing import Set, List, Tuple
import numpy as np

class ReactionNetwork:
    """Chemical reaction network with catalysis."""

    def __init__(self, species: List[str], reactions: List[dict],
       food_set: Set[str]):
        """
        Args:
            species: List of molecule names
            reactions: [{reactants: [s1,s2], products: [s3], catalyst:
               s4}, ...]
            food_set: Externally supplied molecules
        """
        self.species = species
        self.reactions = reactions
        self.food_set = food_set

        # Build indices
        self.species_to_idx = {s: i for i, s in enumerate(species)}

        # Build catalysis graph
        self.catalysis_graph = self._build_catalysis_graph()


    def _build_catalysis_graph(self) -> nx.DiGraph:
        """Build graph where edges s     r mean s catalyzes reaction
           r."""
        G = nx.DiGraph()

        # Add nodes for species and reactions
        for s in self.species:
            G.add_node(('species', s))

        for i, rxn in enumerate(self.reactions):
            G.add_node(('reaction', i))
```

```
36
37              # Add edge from catalyst to reaction
38              if 'catalyst' in rxn and rxn['catalyst']:
39                  G.add_edge(('species', rxn['catalyst']), ('reaction',
                        i))
40
41          return G
42
43
44  def find_all_raf_sets(network: ReactionNetwork, max_size: int = None)
        -> List[Set[int]]:
45      """
46      Find all RAF sets in network.
47
48      Algorithm (Hordijk & Steel):
49      1. Compute closure cl_F = all species producible from food
50      2. For each subset R'     R, check if RAF
51      3. Prune search using reachability constraints
52
53      Returns:
54          List of RAF sets (as sets of reaction indices)
55      """
56      # Compute reachable species from food
57      reachable = compute_food_closure(network,
            set(range(len(network.reactions))))
58
59      # Only consider reactions using reachable species
60      viable_reactions = [
61          i for i, rxn in enumerate(network.reactions)
62          if all(r in reachable for r in rxn['reactants'])
63      ]
64
65      raf_sets = []
66
67      # Search over subsets (exponentialuse heuristics for large
            networks)
68      if max_size is None:
69          max_size = min(10, len(viable_reactions))
70
71      for size in range(1, max_size + 1):
72          for candidate in combinations(viable_reactions, size):
73              candidate_set = set(candidate)
74              if is_raf_set(network, candidate_set):
75                  raf_sets.append(candidate_set)
76
77      return raf_sets
78
79
80  def is_raf_set(network: ReactionNetwork, reaction_subset: Set[int]) ->
        bool:
81      """
82      Check if reaction subset forms a RAF set.
83
84      Conditions:
85      1. Reflexively autocatalytic: every reaction catalyzed by molecule
            in closure
```

```
    2. Food-generated: all reactants producible from food
    """
    # Compute closure
    closure = compute_food_closure(network, reaction_subset)

    # Check catalysis for each reaction
    for rxn_idx in reaction_subset:
        rxn = network.reactions[rxn_idx]

        catalyst = rxn.get('catalyst')
        if catalyst is None:
            return False  # No catalyst assigned

        if catalyst not in closure:
            return False  # Catalyst not producible

    # Check all reactants are in closure
    for rxn_idx in reaction_subset:
        rxn = network.reactions[rxn_idx]
        for reactant in rxn['reactants']:
            if reactant not in closure:
                return False

    return True


def compute_food_closure(network: ReactionNetwork, reaction_subset:
    Set[int]) -> Set[str]:
    """
    Compute cl_F(R'): all species producible from food using reactions
        in subset.

    Fixed-point iteration.
    """
    closure = set(network.food_set)

    changed = True
    while changed:
        changed = False

        for rxn_idx in reaction_subset:
            rxn = network.reactions[rxn_idx]

            # Check if all reactants available
            if all(r in closure for r in rxn['reactants']):
                # Add products to closure
                for product in rxn['products']:
                    if product not in closure:
                        closure.add(product)
                        changed = True

    return closure


def find_minimal_raf_sets(network: ReactionNetwork) -> List[Set[int]]:
    """
```

```
    Find all maxRAF sets (minimal RAF sets with no proper subset being RAF).
    """
    all_rafs = find_all_raf_sets(network)

    # Filter to minimal ones
    minimal_rafs = []

    for raf in all_rafs:
        is_minimal = True

        # Check if any proper subset is also RAF
        for other_raf in all_rafs:
            if other_raf < raf:  # Proper subset
                is_minimal = False
                break

        if is_minimal:
            minimal_rafs.append(raf)

    return minimal_rafs


from itertools import combinations

def raf_detection_exhaustive(network: ReactionNetwork) -> dict:
    """
    Exhaustive RAF detection with certificates.
    """
    minimal_rafs = find_minimal_raf_sets(network)

    # Generate certificates
    certificates = []
    for raf_set in minimal_rafs:
        closure = compute_food_closure(network, raf_set)

        cert = {
            'raf_reactions': list(raf_set),
            'raf_species': list(closure),
            'is_minimal': True,
            'size': len(raf_set),
            'catalysis_verified': verify_catalysis(network, raf_set, closure)
        }
        certificates.append(cert)

    return {
        'n_minimal_rafs': len(minimal_rafs),
        'minimal_rafs': minimal_rafs,
        'certificates': certificates
    }


def verify_catalysis(network: ReactionNetwork, raf_set: Set[int], closure: Set[str]) -> bool:
    """Verify every reaction has catalyst in closure."""
```

```
    for rxn_idx in raf_set:
        catalyst = network.reactions[rxn_idx].get('catalyst')
        if catalyst not in closure:
            return False
    return True
```

Validation: Test on formose reaction network (Breslow 1959).

0.3.2 Phase 2 (Months 2-4): Hypercycle Dynamics

Goal: Simulate hypercycle ODEs and analyze stability.

```
from scipy.integrate import odeint
from scipy.linalg import eig

def hypercycle_ode(n_species: int, catalysis_rates: np.ndarray,
    decay_rates: np.ndarray) -> dict:
    """
    Simulate n-species hypercycle dynamics.

    dx_i/dt = k_i x_{i-1} x_i - d_i x_i -    x_i

    where    =  _j  k_j x_{j-1} x_j (selection flux).
    """
    def dydt(x, t):
        dxdt = np.zeros(n_species)

        # Compute selection flux
        phi = sum(catalysis_rates[i] * x[(i-1) % n_species] * x[i]
                  for i in range(n_species))

        for i in range(n_species):
            i_prev = (i - 1) % n_species

            production = catalysis_rates[i] * x[i_prev] * x[i]
            decay = decay_rates[i] * x[i]
            dilution = phi * x[i]

            dxdt[i] = production - decay - dilution

        return dxdt

    # Initial conditions (small random perturbation)
    x0 = np.ones(n_species) / n_species + 0.01 *
       np.random.randn(n_species)
    x0 = np.maximum(x0, 0.001)  # Ensure positive

    # Integrate
    t = np.linspace(0, 1000, 10000)
    sol = odeint(dydt, x0, t)

    # Check coexistence
    final_state = sol[-1]
    coexists = all(final_state > 1e-3)

    # Compute stability (Jacobian at equilibrium)
```

```
    if coexists:
        J = compute_hypercycle_jacobian(final_state, catalysis_rates,
            decay_rates)
        eigenvalues = eig(J)[0]
        is_stable = all(np.real(eigenvalues) < 0)
    else:
        is_stable = False
        eigenvalues = None

    return {
        'trajectory': sol,
        'time': t,
        'coexists': coexists,
        'final_state': final_state,
        'is_stable': is_stable,
        'eigenvalues': eigenvalues
    }


def compute_hypercycle_jacobian(x_eq: np.ndarray, k: np.ndarray, d:
    np.ndarray) -> np.ndarray:
    """
    Compute Jacobian matrix at equilibrium.

    J_ij =    (dx_i/dt) /   x_j
    """
    n = len(x_eq)
    J = np.zeros((n, n))

    for i in range(n):
        i_prev = (i - 1) % n

        # Diagonal term
        J[i, i] = k[i] * x_eq[i_prev] - d[i] - sum(k[j] * x_eq[(j-1) %
            n] for j in range(n))

        # Off-diagonal (coupling through catalysis)
        J[i, i_prev] = k[i] * x_eq[i]

        # Selection flux coupling
        for j in range(n):
            j_prev = (j - 1) % n
            J[i, j] -= k[j] * x_eq[j_prev] * x_eq[i] / n  # Approximate

    return J


def test_hypercycle_stability(n_range: range = range(2, 10)) -> dict:
    """
    Test Eigen's conjecture: hypercycles stable only for n     5.
    """
    results = {}

    for n in n_range:
        # Random catalysis and decay rates
        k = np.random.uniform(0.5, 2.0, n)
```

```
        d = np.random.uniform(0.1, 0.5, n)

        hypercycle_result = hypercycle_ode(n, k, d)

        results[n] = {
            'coexists': hypercycle_result['coexists'],
            'stable': hypercycle_result['is_stable']
        }

    return results
```

Validation: Reproduce Eigen Schuster (1979) stability threshold n 5.

0.3.3 Phase 3 (Months 4-5): Thermodynamic Constraints

Goal: Verify G < 0 for all reactions in RAF sets.

```
def compute_reaction_free_energy(reaction: dict,
                                 concentrations: dict,
                                 standard_free_energies: dict,
                                 T: float = 298.15) -> float:
    """
    Compute Gibbs free energy change for reaction.

     G  =   G   + RT ln(Q)

    where Q = [products] / [reactants] (reaction quotient).
    """
    R = 8.314  # J/(mol K)

    # Compute   G
    delta_G_standard = 0
    for product in reaction['products']:
        delta_G_standard += standard_free_energies.get(product, 0)
    for reactant in reaction['reactants']:
        delta_G_standard -= standard_free_energies.get(reactant, 0)

    # Compute reaction quotient Q
    Q = 1.0
    for product in reaction['products']:
        Q *= concentrations.get(product, 1e-6)
    for reactant in reaction['reactants']:
        Q /= max(concentrations.get(reactant, 1e-6), 1e-10)

    # Gibbs free energy
    delta_G = delta_G_standard + R * T * np.log(Q)

    return delta_G


def verify_thermodynamic_viability(network: ReactionNetwork,
                                   raf_set: Set[int],
                                   concentrations: dict,
                                   standard_free_energies: dict) -> dict:
    """
    Verify all reactions in RAF set have  G  < 0.
```

```
    """
    viable = True
    delta_Gs = []

    for rxn_idx in raf_set:
        rxn = network.reactions[rxn_idx]

        delta_G = compute_reaction_free_energy(rxn, concentrations,
            standard_free_energies)
        delta_Gs.append(delta_G)

        if delta_G >= 0:
            viable = False

    return {
        'thermodynamically_viable': viable,
        'delta_Gs': delta_Gs,
        'max_delta_G': max(delta_Gs) if delta_Gs else 0,
        'mean_delta_G': np.mean(delta_Gs) if delta_Gs else 0
    }
```

Validation: Check formose reaction G values against literature.

0.3.4 Phase 4 (Months 5-7): Information-Theoretic Analysis

Goal: Quantify information emergence in autocatalytic sets.

```
from scipy.stats import entropy

def compute_shannon_entropy(concentrations: np.ndarray) -> float:
    """
    Shannon entropy H(X) = -   p_i log_2(p_i).

    Measures diversity of molecular species.
    """
    # Normalize to probabilities
    probs = concentrations / np.sum(concentrations)
    probs = probs[probs > 0]  # Remove zeros

    H = entropy(probs, base=2)  # bits

    return H


def compute_mutual_information(conc_X: np.ndarray, conc_Y: np.ndarray)
    -> float:
    """
    Mutual information I(X;Y) between two molecular species.

    I(X;Y) = H(X) + H(Y) - H(X,Y)
    """
    # Joint distribution (discretize concentrations)
    hist_2d, _, _ = np.histogram2d(conc_X, conc_Y, bins=10)
    hist_2d = hist_2d / np.sum(hist_2d)  # Normalize

    # Marginals
```

```
    hist_X = np.sum(hist_2d, axis=1)
    hist_Y = np.sum(hist_2d, axis=0)

    # Entropies
    H_X = entropy(hist_X[hist_X > 0], base=2)
    H_Y = entropy(hist_Y[hist_Y > 0], base=2)
    H_XY = entropy(hist_2d[hist_2d > 0].flatten(), base=2)

    I_XY = H_X + H_Y - H_XY

    return I_XY


def information_analysis_raf(network: ReactionNetwork,
                             raf_set: Set[int],
                             trajectory: np.ndarray) -> dict:
    """
    Compute information-theoretic properties of RAF dynamics.
    """
    n_species = len(network.species)

    # Shannon entropy over time
    entropies = [compute_shannon_entropy(trajectory[t]) for t in
       range(len(trajectory))]

    # Mutual information matrix
    I_matrix = np.zeros((n_species, n_species))

    for i in range(n_species):
        for j in range(i+1, n_species):
            I_matrix[i, j] = compute_mutual_information(trajectory[:,
               i], trajectory[:, j])
            I_matrix[j, i] = I_matrix[i, j]

    return {
        'entropy_trajectory': entropies,
        'final_entropy': entropies[-1],
        'entropy_increase': entropies[-1] - entropies[0],
        'mutual_information_matrix': I_matrix,
        'mean_mutual_information': np.mean(I_matrix[I_matrix > 0])
    }
```

Validation: Verify H(X) increases as RAF set emerges from simple precursors.

0.3.5 Phase 5 (Months 7-8): Origin of Life Scenarios

Goal: Apply RAF theory to prebiotic chemistry (formose, amino acids, nucleotides).

```
def formose_reaction_network() -> ReactionNetwork:
    """
    Construct formose reaction network (autocatalytic sugar synthesis).

    HCHO     glycolaldehyde     glyceraldehyde     ...     sugars
    """
    species = [
        'HCHO',  # Formaldehyde (food)
```

```
        'glycolaldehyde',
        'glyceraldehyde',
        'dihydroxyacetone',
        'erythrose',
        'ribose'
    ]

    reactions = [
        {'reactants': ['HCHO', 'HCHO'], 'products': ['glycolaldehyde'],
         'catalyst': 'glycolaldehyde'},  # Autocatalytic
        {'reactants': ['HCHO', 'glycolaldehyde'], 'products':
            ['glyceraldehyde'],
         'catalyst': 'glyceraldehyde'},
        {'reactants': ['glyceraldehyde', 'HCHO'], 'products':
            ['erythrose'],
         'catalyst': 'erythrose'},
        {'reactants': ['erythrose', 'HCHO'], 'products': ['ribose'],
         'catalyst': 'ribose'}
    ]

    food_set = {'HCHO'}

    return ReactionNetwork(species, reactions, food_set)


def amino_acid_network() -> ReactionNetwork:
    """
    Simplified amino acid synthesis from HCN, NH3, H2O.
    """
    # ... (similar construction)
    pass


def origin_of_life_analysis(network: ReactionNetwork) -> dict:
    """
    Complete origin of life analysis pipeline.
    """
    # 1. Find RAF sets
    raf_result = raf_detection_exhaustive(network)

    # 2. Thermodynamic viability
    concentrations = {s: 0.001 for s in network.species}  # 1 mM
    concentrations.update({s: 1.0 for s in network.food_set})  # 1 M
        food

    standard_free_energies =
        estimate_standard_free_energies(network.species)

    thermo_results = []
    for raf_set in raf_result['minimal_rafs']:
        thermo = verify_thermodynamic_viability(network, raf_set,
            concentrations, standard_free_energies)
        thermo_results.append(thermo)

    # 3. Hypercycle stability (if applicable)
    # ... (check if RAF forms hypercycle structure)
```

```
    # 4. Information emergence
    # ... (simulate dynamics and compute entropy)

    return {
        'raf_detection': raf_result,
        'thermodynamics': thermo_results,
        'conclusion': 'VIABLE' if any(t['thermodynamically_viable'] for
            t in thermo_results) else 'NOT_VIABLE'
    }


def estimate_standard_free_energies(species: List[str]) -> dict:
    """
    Estimate   G   from group contribution methods or database lookup.
    """
    # Placeholder: use Benson group additivity or lookup tables
    free_energies = {}

    for s in species:
        # Approximate: small organic molecules ~ -100 to -200 kJ/mol
        free_energies[s] = -150.0 + 50.0 * np.random.randn()

    return free_energies
```

Validation: Reproduce Kauffman's autocatalytic set emergence threshold.

0.3.6 Phase 6 (Months 8-9): Certificate Generation

Goal: Generate complete certificates for all RAF sets found.

```
from dataclasses import dataclass, asdict
import json

@dataclass
class RAFCertificate:
    """Complete certificate for RAF set."""

    network_name: str
    n_species: int
    n_reactions: int
    food_set: List[str]

    # RAF properties
    raf_reactions: List[int]
    raf_species: List[str]
    is_minimal: bool

    # Verification
    catalytic_closure_verified: bool
    food_generation_verified: bool
    thermodynamically_viable: bool
    mean_delta_G: float

    # Dynamics
    hypercycle_stable: bool
```

```
26
27      # Information
28      shannon_entropy: float
29      mutual_information_mean: float
30
31      # Metadata
32      computation_date: str
33
34      def export_json(self, filename: str):
35          with open(filename, 'w') as f:
36              json.dump(asdict(self), f, indent=2)
37
38      def verify(self) -> bool:
39          checks = [
40              len(self.raf_reactions) > 0,
41              len(self.raf_species) >= len(self.food_set),
42              self.catalytic_closure_verified,
43              self.food_generation_verified
44          ]
45          return all(checks)
46
47
48  def generate_raf_certificate(network: ReactionNetwork, raf_set:
        Set[int]) -> RAFCertificate:
49      """Generate complete certificate for RAF set."""
50      closure = compute_food_closure(network, raf_set)
51
52      # Thermodynamics
53      concentrations = {s: 0.001 for s in network.species}
54      standard_free_energies =
            estimate_standard_free_energies(network.species)
55      thermo = verify_thermodynamic_viability(network, raf_set,
            concentrations, standard_free_energies)
56
57      cert = RAFCertificate(
58          network_name='Formose',
59          n_species=len(network.species),
60          n_reactions=len(network.reactions),
61          food_set=list(network.food_set),
62          raf_reactions=list(raf_set),
63          raf_species=list(closure),
64          is_minimal=True,
65          catalytic_closure_verified=verify_catalysis(network, raf_set,
                closure),
66          food_generation_verified=(closure.issuperset(
67              set(r for rxn_idx in raf_set for r in
                    network.reactions[rxn_idx]['reactants'])
68          )),
69          thermodynamically_viable=thermo['thermodynamically_viable'],
70          mean_delta_G=thermo['mean_delta_G'],
71          hypercycle_stable=False,  # Would require dynamics
72          shannon_entropy=0.0,  # Would require simulation
73          mutual_information_mean=0.0,
74          computation_date='2026-01-17'
75      )
76
```

```
77      return cert
```

0.4 4. Example Starting Prompt

Prompt for AI System:

You are tasked with finding autocatalytic sets in chemical reaction networks to model the origin of life. Your goals:

- **RAF Detection (Months 1-2)**:
- Implement formose reaction network (6 species, 10 reactions)
- Find all minimal RAF sets
- Verify catalytic closure and food-generation
- **Hypercycle Dynamics (Months 2-4)**:
- Simulate n-species hypercycles for n = 2,...,10
- Verify Eigen's stability threshold (n 5)
- Compute Jacobian eigenvalues
- **Thermodynamics (Months 4-5)**:
- Compute G for all reactions
- Verify viability: G < 0
- Use group contribution methods
- **Information Theory (Months 5-7)**:
- Compute Shannon entropy H(X)
- Compute mutual information I(X;Y)
- Track entropy increase over time
- **Origin of Life (Months 7-8)**:
- Apply to formose, amino acids, nucleotides
- Find minimal autocatalytic sets
- Compare to literature thresholds
- **Certificates (Months 8-9)**:
- Generate RAFCertificate for each set
- Export to JSON
- Verify all certificates

Success Criteria:

- MVR (2-4 months): RAF detection for toy networks
- Strong (6-8 months): Formose analysis, hypercycle stability
- Publication (9 months): Complete origin-of-life scenario

References:

- Kauffman (1986): Autocatalytic sets
- Eigen Schuster (1979): Hypercycle theory
- Hordijk Steel (2004): RAF algorithm

Begin by implementing RAF detection for formose network.

0.5 5. Success Criteria

0.5.1 Minimum Viable Result (Months 1-4)

Core Achievements:

- RAF detection for networks with 10-20 species
- Catalytic closure verification
- Basic hypercycle simulation (n 5)
- Certificate generation

Validation:

- Find RAF in formose network
- Reproduce Eigen stability threshold

Deliverables:

- Python module `raf`$_d$*etection.py*
- Jupyter notebook: formose analysis
- JSON certificates for 3+ RAF sets

0.5.2 Strong Result (Months 4-8)

Extended Capabilities:

- Thermodynamic viability checks
- Information-theoretic analysis
- Hypercycle stability for n 10
- Multiple prebiotic networks

Publications Benchmark:

- Reproduce Hordijk Steel (2004) results
- Match thermodynamic thresholds

Deliverables:

- Database of 10+ networks
- Entropy vs time plots
- Stability phase diagrams

0.5.3 Publication-Quality Result (Months 8-9)

Novel Contributions:

- Novel RAF sets in unexplored chemistry
- Thermodynamic-information tradeoffs
- Predictive model for RAF emergence
- Experimental predictions

Deliverables:

- Arxiv preprint on origin of life
- Public database of RAF sets
- Web tool for RAF detection

0.6 6. Verification Protocol

```python
def verify_raf_certificate(cert: RAFCertificate) -> dict:
    results = {
        'catalysis_verified': cert.catalytic_closure_verified,
        'food_generation_verified': cert.food_generation_verified,
        'thermodynamically_viable': cert.thermodynamically_viable,
        'certificate_valid': cert.verify()
    }

    results['all_checks_passed'] = all(v for v in results.values()
        if isinstance(v, bool))

    return results
```

0.7 7. Resources and Milestones

0.7.1 Essential References

- **Foundational Papers**:
- Kauffman (1986): "Autocatalytic sets of proteins"
- Eigen Schuster (1979): *The Hypercycle*
- Hordijk Steel (2004): "Detecting autocatalytic, self-sustaining sets"
- **Modern Work**:
- Xavier et al. (2020): "Autocatalytic chemical networks at the origin of life"
- Vasas et al. (2012): "Evolution before genes"

0.7.2 Milestone Checklist

Month 1: RAF detection implemented

Month 2: Formose network analyzed

Month 3: Hypercycle simulations working

Month 4: Thermodynamic checks complete

Month 5: Information theory implemented

Month 6: 5+ networks analyzed

Month 7: Origin-of-life scenarios tested

Month 8: Certificates generated

Month 9: Database exported

End of PRD 29