Hypercube-HDAG Framework

Self-Compiling n-Dimensional Containers
for Deterministic Artifact Generation

Sebastian Klemm
Formalization for 5D-Reasoning Systems

November 1, 2025

Abstract

This document formalizes the **Hypercube-HDAG** data structure:
an n-dimensional container that embeds a Hyperdimensional Directed Acyclic Graph (HDAG) as an executable sequence. The structure exhibits *self-compilation* properties, allowing deterministic generation of complex artifacts and entire ecosystems from initial configurations.
The framework provides: (1) formal definition of n-dimensional hypercubes, (2) HDAG embedding mechanisms, (3) self-compilation protocols, (4) deterministic expansion operators, and (5) integration patterns for 5D-reasoning agent architectures.

Contents

1 Introduction

1.1 Motivation

Modern computational systems require structures that can:

- Encode complex configurations in compact form
- Exhibit self-organizing and self-compiling properties
- Generate artifacts deterministically and reproducibly
- Support multidimensional reasoning (especially 5D)
- Serve as executable containers for graph-based processes

The **Hypercube-HDAG** structure addresses these requirements by combining:

1. **Hypercubes** ($\mathcal{C}_n$): n-dimensional geometric containers
2. **HDAG**: Hyperdimensional Directed Acyclic Graphs embedded as execution sequences
3. **Self-Compilation**: Bootstrap mechanisms that expand cubes into full artifacts

1.2 Core Concept

Central Idea

A Hypercube $\mathcal{C}_n$ is not merely a static data structure, but a *generative kernel* containing:

- Initial state $\boldsymbol{\sigma}_0 \in \mathbb{R}^n$
- Embedded HDAG $G = (V, E)$ as "script"
- Compilation operator $\Xi : \mathcal{C}_n \to \mathcal{A}$

Upon compilation, the cube deterministically unfolds into a complete artifact $\mathcal{A}$ via the embedded HDAG instructions.

1.3 Applications

- **Knowledge Systems:** Encode entire knowledge graphs in compact cubes
- **Software Generation:** Compile application architectures from specification cubes
- **Molecular Design:** Generate molecular structures from parameter cubes
- **Ecosystem Modeling:** Bootstrap complex adaptive systems from seed cubes
- **5D-Reasoning Agents:** Serve as memory and planning structures

2 Mathematical Foundations

2.1 n-Dimensional Hypercubes

Definition 2.1 (Hypercube)**.** An n-dimensional hypercube $\mathcal{C}_n$ is a geometric object defined by:

$$\mathcal{C}_n = \{\boldsymbol{x} \in \mathbb{R}^n : 0 \le x_i \le 1,\ i = 1, \dots, n\}$$

It has:

- 2^n vertices (corners)
- $n \cdot 2^{n-1}$ edges
- $\binom{n}{k} \cdot 2^{n-k}$ k-dimensional faces

Definition 2.2 (Hypercube State Space)**.** The state space of a hypercube is:

$$\mathcal{S}_{\mathcal{C}_n} = \{\boldsymbol{\sigma} : \boldsymbol{\sigma} \in \mathbb{R}^n\} = \mathbb{R}^n$$

Each point $\boldsymbol{\sigma} \in \mathcal{S}_{\mathcal{C}_n}$ represents a potential configuration.

Example Dimensions:

- **3D**: Classical cube, $2^3 = 8$ vertices
- **4D**: Tesseract, $2^4 = 16$ vertices
- **5D**: 5-cube (penteract), $2^5 = 32$ vertices
- **nD**: General hypercube, 2^n vertices

2.2 HDAG: Hyperdimensional Directed Acyclic Graphs

Definition 2.3 (HDAG)**.** A Hyperdimensional Directed Acyclic Graph is a tuple:

$$G = (V, E, T, \Phi)$$

where:

- $V = \{v_1, \dots, v_m\}$ is a set of nodes
- $E \subseteq V \times V$ is a set of directed edges with no cycles
- $T : V \to \mathbb{R}^d$ maps each node to a d-dimensional tensor
- $\Phi : E \to (\mathbb{R}^d \to \mathbb{R}^d)$ maps edges to phase-gradient transformations

Key Properties:

1. **Acyclicity:** No directed cycles, ensuring termination
2. **Tensorization:** Each node carries a high-dimensional state
3. **Phase Coherence:** Edge transformations maintain semantic alignment

Definition 2.4 (HDAG Execution)**.** An HDAG is executed by traversing from source nodes (no incoming edges) to sink nodes (no outgoing edges), applying transformations:

$$T(v_j) = \Phi_{ij}(T(v_i)) \quad \forall (v_i, v_j) \in E$$

2.3 Embedding HDAG in Hypercubes

Definition 2.5 (Hypercube-HDAG Structure)**.** A Hypercube-HDAG is a composite structure:

$$\mathcal{Q} = (\mathcal{C}_n, G, \iota, \Xi)$$

where:

- $\mathcal{C}_n$ is an n-dimensional hypercube
- G is an embedded HDAG
- $\iota : G \to \mathcal{C}_n$ is an embedding function mapping HDAG nodes to cube positions

- $\Xi : \mathcal{Q} \to \mathcal{A}$ is the compilation operator

Embedding Strategies:

1. **Vertex Embedding:** Map HDAG nodes to hypercube vertices

$$\iota(v_i) \in \{\text{vertices of } \mathcal{C}_n\}$$

2. **Continuous Embedding:** Map nodes to interior points

$$\iota(v_i) \in \mathcal{C}_n, \quad \text{with } 0 < x_j < 1$$

3. **Hierarchical Embedding:** Map based on HDAG topology

$$\text{depth}(v_i) \propto x_{\text{temp}}(\iota(v_i))$$

3 Self-Compilation Mechanism

Definition 3.1 (Compilation Operator)**.** The compilation operator $\Xi : \mathcal{Q} \to \mathcal{A}$ transforms a hypercube-HDAG into a complete artifact through iterative expansion:

$$\mathcal{A} = \Xi(\mathcal{Q}) = \lim_{k \to \infty} \Xi^k(\mathcal{Q}_0)$$

where $\mathcal{Q}_0$ is the initial seed cube.

3.1 Bootstrap Protocol

Algorithm 1 Hypercube Self-Compilation

Input: Seed cube $\mathcal{Q}_0 = (\mathcal{C}_n, G_0, \iota_0, \Xi)$
Output: Compiled artifact $\mathcal{A}$

$\mathcal{Q} \leftarrow \mathcal{Q}_0$
$\mathcal{A} \leftarrow \emptyset$ ▷ Empty artifact

while $\neg$converged($\mathcal{Q}$) **do**
    ▷ Phase 1: Extract HDAG Instructions
    $G \leftarrow$ extract_hdag($\mathcal{Q}$)

    ▷ Phase 2: Execute HDAG
    for $v \in$ topological_sort(G) **do**
        $\boldsymbol{\sigma}_v \leftarrow T(v)$ ▷ Get node tensor
        for $(v, w) \in E$ **do**
            $\boldsymbol{\sigma}_w \leftarrow \Phi_{vw}(\boldsymbol{\sigma}_v)$ ▷ Apply transformation
        end for
    end for

    ▷ Phase 3: Materialize Layer
    $\mathcal{A} \leftarrow \mathcal{A} \cup$ materialize(G)

    ▷ Phase 4: Update Cube (Spiral Expansion)
    $\mathcal{Q} \leftarrow$ expand($\mathcal{Q}, \mathcal{A}$)
end while
return $\mathcal{A}$

3.2 Determinism Guarantees

Theorem 3.1 (Deterministic Compilation)**.** Given identical seed cubes $\mathcal{Q}_1 = \mathcal{Q}_2$, the compilation operator produces identical artifacts:

$$\mathcal{Q}_1 = \mathcal{Q}_2 \implies \Xi(\mathcal{Q}_1) = \Xi(\mathcal{Q}_2)$$

Proof. By construction:

1. HDAG execution is deterministic (DAG traversal)
2. Tensor transformations Φ_{ij} are functions (single-valued)
3. Materialization is a fixed mapping from graph states to artifact components
4. Expansion follows deterministic rules

Therefore, given identical inputs, all intermediate states are identical, yielding identical outputs. □

3.3 Convergence Criteria

Definition 3.2 (Cube Convergence)**.** A cube $\mathcal{Q}$ has converged if:

$$\|\Xi(\mathcal{Q}) - \mathcal{Q}\| < \epsilon$$

for some threshold $\epsilon > 0$, or if a maximum iteration count is reached.

4 5D Instantiation

4.1 5D Hypercube

For 5-dimensional reasoning systems:

$$\mathcal{C}_5 = \{\boldsymbol{\sigma} = (\psi, \rho, \omega, \chi, \eta) : \boldsymbol{\sigma} \in [0,1]^5\}$$

Coordinate Semantics:

- ψ: **Potential/Energy** – primary feature intensity
- ρ: **Density** – concentration or magnitude
- ω: **Frequency** – oscillatory/temporal component
- χ: **Connectivity** – relational coupling strength
- η: **Causality** – temporal/causal ordering

4.2 5D-HDAG Nodes

Definition 4.1 (5D-HDAG Node)**.** Each node $v \in V$ carries a 5D tensor:

$$T(v) = (\psi_v, \rho_v, \omega_v, \chi_v, \eta_v) \in \mathbb{R}^5$$

4.3 Phase-Gradient Edges

Definition 4.2 (5D Phase Transformation)**.** Edge transformations in 5D are defined as:

$$\Phi_{ij}(\boldsymbol{\sigma}) = \mathcal{O}(\boldsymbol{\sigma}) + \nabla_\mu T_i$$

where $\mathcal{O}$ is an operator from {DK, SW, PI, WT} and ∇_μ is a gradient operator.

Operator Correspondence:

- **DK (Doppelkick):** Double rotation in (ψ, ρ) and (ω, χ) planes
- **SW (Schwellenwert):** Threshold-based filtering on ρ
- **PI (Pfadinvarianz):** Path-invariant transformation maintaining χ
- **WT (Wurmloch):** Acceleration along η axis

4.4 Resonance Condition

Definition 4.3 (5D Semantic Coherence)**.** An edge (v_i, v_j) is valid only if:

$$\mathcal{R}(T(v_i), T(v_j)) = \psi_i \cdot \psi_j \cdot \rho_i \cdot \rho_j \cdot \cos(\omega_i - \omega_j) \geq \tau$$

where τ is a resonance threshold.

This prevents cycles through phase misalignment: if nodes would form a cycle, accumulated phase differences violate the coherence condition.

5 Artifact Generation

5.1 Artifact Types

Definition 5.1 (Artifact)**.** An artifact $\mathcal{A}$ is a structured output generated by cube compilation:

$$\mathcal{A} = \{C_1, C_2, \ldots, C_k\}$$

where each C_i is a component (module, function, entity, etc.).

Examples:

- **Knowledge Graphs:** C_i = entities, edges = relations
- **Software:** C_i = modules, classes, functions
- **Molecules:** C_i = atoms, bonds
- **Ecosystems:** C_i = agents, interactions

5.2 Layered Materialization

Definition 5.2 (Materialization Layers)**.** Artifacts are built in layers $L_0, L_1, \ldots, L_n$:

$$\mathcal{A} = \bigcup_{k=0}^{n} L_k$$

where L_{k+1} depends on L_k.

Algorithm 2 Layered Artifact Generation

Input: Compiled HDAG G, depth n
Output: Artifact $\mathcal{A}$

$\mathcal{A} \leftarrow \emptyset$
$L_0 \leftarrow \text{initial\_layer}(G)$ ▷ Source nodes
$\mathcal{A} \leftarrow \mathcal{A} \cup L_0$

for $k = 1$ **to** n **do**
  $L_k \leftarrow \emptyset$
  for $v \in L_{k-1}$ **do**
    for $(v, w) \in E$ **do**
      $C_w \leftarrow \text{generate\_component}(T(w))$
      $L_k \leftarrow L_k \cup \{C_w\}$
    end for
  end for
  $\mathcal{A} \leftarrow \mathcal{A} \cup L_k$
end for
return $\mathcal{A}$

5.3 Ecosystem Generation

Ecosystem from Hypercube

A particularly powerful application is generating entire *ecosystems* of interacting agents:

1. **Seed Cube:** $\mathcal{Q}_{\text{eco}} = (\mathcal{C}_5, G_{\text{agents}}, \iota, \Xi)$
2. **HDAG:** Defines agent types, interaction rules, resource flows
3. **Compilation:** Generates $\mathcal{A}_{\text{eco}} = \{A_1, \ldots, A_m\}$ agents
4. **Dynamics:** Agents interact according to compiled rules
5. **Evolution:** System exhibits emergent behavior

6 Implementation Architecture

6.1 Data Structures

Hypercube Structure:

Listing 1: Hypercube Class

```
class Hypercube:
    def __init__(self, dimension: int):
        self.dimension = dimension
        self.state = np.zeros(dimension)
        self.hdag = HDAG()
        self.embedding = {}  # node -> position mapping

    def embed_hdag(self, hdag: HDAG):
        """Embed HDAG into hypercube structure"""
        self.hdag = hdag
        self.embedding = self._compute_embedding(hdag)

    def compile(self) -> Artifact:
        """Self-compilation: expand into full artifact"""
        return self._bootstrap_expansion()
```

HDAG Structure:

Listing 2: HDAG Class

```
class HDAG:
    def __init__(self, dimension: int = 5):
        self.nodes = {}  # node_id -> Tensor
        self.edges = []  # (source, target, transform)
        self.dimension = dimension

    def add_node(self, node_id: str, tensor: np.ndarray):
        """Add node with d-dimensional tensor"""
        assert len(tensor) == self.dimension
        self.nodes[node_id] = tensor

    def add_edge(self, source: str, target: str,
                 transform: Callable):
        """Add directed edge with transformation"""
        self.edges.append((source, target, transform))

    def execute(self) -> Dict:
        """Topological execution of HDAG"""
        order = self._topological_sort()
        results = {}

        for node_id in order:
            # Get predecessors
            preds = [e[0] for e in self.edges
                     if e[1] == node_id]

            if not preds:
                # Source node
                results[node_id] = self.nodes[node_id]
            else:
                # Apply transformations
                tensors = [results[p] for p in preds]
                transforms = [e[2] for e in self.edges
                              if e[1] == node_id]

                # Combine transformed inputs
                results[node_id] = self._combine(
                    [t(x) for t, x in zip(transforms, tensors)]
                )
```

```python

        return results
```

6.2 Compilation Engine

Listing 3: Self-Compilation Engine

```python
class CompilationEngine:
    def __init__(self, seed_cube: Hypercube):
        self.cube = seed_cube
        self.artifact = Artifact()
        self.iteration = 0

    def bootstrap(self, max_iter: int = 100) -> Artifact:
        """Bootstrap compilation process"""
        while not self._converged() and self.iteration < max_iter:
            # Extract HDAG instructions
            hdag = self.cube.hdag

            # Execute HDAG
            results = hdag.execute()

            # Materialize layer
            layer = self._materialize_layer(results)
            self.artifact.add_layer(layer)

            # Expand cube (spiral)
            self.cube = self._spiral_expand(self.cube, results)

            self.iteration += 1

        return self.artifact

    def _spiral_expand(self, cube: Hypercube,
                       results: Dict) -> Hypercube:
        """Spiral expansion: grow cube based on HDAG results"""
        # Create new cube with expanded structure
        new_cube = Hypercube(cube.dimension)

        # Update state based on HDAG outputs
        new_state = self._compute_new_state(cube.state, results)
        new_cube.state = new_state

        # Generate next-level HDAG
        new_hdag = self._generate_next_hdag(cube.hdag, results)
        new_cube.embed_hdag(new_hdag)

        return new_cube

    def _converged(self) -> bool:
        """Check convergence criteria"""
        if self.iteration == 0:
            return False

        # Check if artifact is stable
        return self.artifact.is_stable()
```

6.3 5D Operators

Listing 4: 5D Transformation Operators

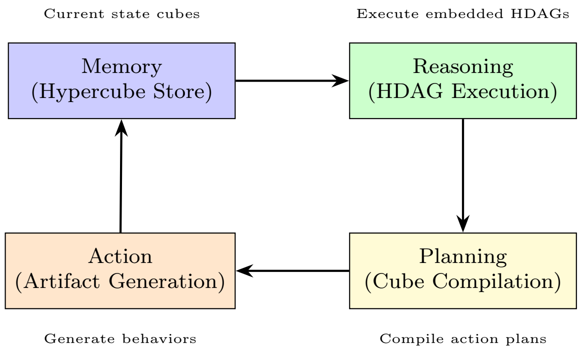
```python
class Operator5D:
    """Base class for 5D operators"""
    def __init__(self):
```

```python
        pass

    def apply(self, sigma: np.ndarray) -> np.ndarray:
        raise NotImplementedError

class Doppelkick(Operator5D):
    """DK: Double rotation operator"""
    def apply(self, sigma: np.ndarray) -> np.ndarray:
        # Rotate in (psi, rho) plane
        psi, rho, omega, chi, eta = sigma

        # Rotation angle
        theta = np.pi / 2

        # Apply rotations
        psi_new = psi * np.cos(theta) - rho * np.sin(theta)
        rho_new = psi * np.sin(theta) + rho * np.cos(theta)

        omega_new = omega * np.cos(theta) - chi * np.sin(theta)
        chi_new = omega * np.sin(theta) + chi * np.cos(theta)

        return np.array([psi_new, rho_new, omega_new, chi_new, eta])

class Schwellenwert(Operator5D):
    """SW: Threshold filter operator"""
    def __init__(self, threshold: float = 0.5, mc: float = 0.7):
        super().__init__()
        self.threshold = threshold
        self.mc = mc

    def apply(self, sigma: np.ndarray) -> np.ndarray:
        psi, rho, omega, chi, eta = sigma

        # Resonance normalization
        if rho < self.threshold:
            factor = 0.0
        elif rho < self.mc:
            factor = (rho - self.threshold) / (self.mc - self.threshold)
        else:
            factor = 1.0 + (rho - self.mc)

        return sigma * (1 + factor * 0.1)

class Pfadinvarianz(Operator5D):
    """PI: Path invariance operator"""
    def __init__(self, kappa: float = 0.1):
        super().__init__()
        self.kappa = kappa

    def apply(self, sigma: np.ndarray) -> np.ndarray:
        # Damping factor preserves chi (connectivity)
        return sigma * (1 - self.kappa)

class Wurmloch(Operator5D):
    """WT: Wormhole (eta-boost) operator"""
    def __init__(self, k: float = 0.2):
        super().__init__()
        self.k = k

    def apply(self, sigma: np.ndarray) -> np.ndarray:
        psi, rho, omega, chi, eta = sigma

        # Boost along eta axis
        eta_new = eta + self.k * np.linalg.norm(sigma[:4])

```

```
        return np.array([psi, rho, omega, chi, eta_new])
```

7 Agent Integration

7.1 Agent Architecture with Hypercubes



7.2 Agent Memory as Hypercubes

Definition 7.1 (Memory Cube)**.** An agent's working memory is a collection of hypercubes:

$$M = \{\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_k\}$$

where each $\mathcal{Q}_i$ represents a concept, goal, or experience.

Operations:

- **Store:** Encode new experience as cube
- **Recall:** Query cubes by resonance
- **Compile:** Expand cube into explicit knowledge
- **Merge:** Combine cubes via tensor operations

7.3 Planning with Cubes

Algorithm 3 Hypercube-Based Planning

Input: Goal $\mathcal{G}$, current state $\boldsymbol{\sigma}_0$
Output: Action plan $\mathcal{A}_{\text{plan}}$

▷ Create goal cube
$\mathcal{Q}_{\text{goal}} \leftarrow \text{encode\_goal}(\mathcal{G})$

▷ Create current state cube
$\mathcal{Q}_{\text{now}} \leftarrow \text{encode\_state}(\boldsymbol{\sigma}_0)$

▷ Generate HDAG connecting current to goal
$G_{\text{plan}} \leftarrow \text{build\_hdag}(\mathcal{Q}_{\text{now}}, \mathcal{Q}_{\text{goal}})$

▷ Create planning cube
$\mathcal{Q}_{\text{plan}} \leftarrow (\mathcal{C}_5, G_{\text{plan}}, \iota, \Xi)$

▷ Compile plan
$\mathcal{A}_{\text{plan}} \leftarrow \Xi(\mathcal{Q}_{\text{plan}})$
return $\mathcal{A}_{\text{plan}}$

8 Examples

8.1 Example 1: Knowledge Graph Generation

Seed Cube:

```
# Create 5D seed cube for knowledge graph
cube = Hypercube(dimension=5)
cube.state = np.array([0.5, 0.8, np.pi, 0.6, 0.0])

# Define HDAG for graph structure
hdag = HDAG(dimension=5)

# Add concept nodes
hdag.add_node("concept_A", np.array([0.8, 0.9, 2.1, 0.7, 0.1]))
hdag.add_node("concept_B", np.array([0.6, 0.7, 2.3, 0.8, 0.2]))
hdag.add_node("relation", np.array([0.5, 0.5, 2.2, 0.9, 0.15]))

# Add edges with transformations
hdag.add_edge("concept_A", "relation", Doppelkick())
hdag.add_edge("concept_B", "relation", Schwellenwert())

# Embed HDAG into cube
cube.embed_hdag(hdag)

# Compile to generate knowledge graph
engine = CompilationEngine(cube)
kg_artifact = engine.bootstrap(max_iter=50)
```

8.2 Example 2: Software Architecture Generation

Specification Cube:

```
# Encode software requirements as 5D state
requirements = np.array([
    0.9,  # psi: high functionality
    0.7,  # rho: moderate complexity
    1.5,  # omega: standard frequency
    0.8,  # chi: high modularity
    0.0   # eta: no time constraints yet
])

cube = Hypercube(dimension=5)
cube.state = requirements

# HDAG defines architecture patterns
hdag = HDAG(dimension=5)

# Core modules
hdag.add_node("frontend", np.array([0.8, 0.6, 1.2, 0.9, 0.1]))
hdag.add_node("backend", np.array([0.9, 0.8, 1.5, 0.7, 0.2]))
hdag.add_node("database", np.array([0.7, 0.9, 1.3, 0.6, 0.3]))

# Dependencies
hdag.add_edge("frontend", "backend", Pfadinvarianz())
hdag.add_edge("backend", "database", Wurmloch())

cube.embed_hdag(hdag)

# Compile to generate full architecture
software_artifact = CompilationEngine(cube).bootstrap()
```

9 Theoretical Properties

9.1 Complexity Analysis

Proposition 9.1 (Compilation Complexity)**.** For a hypercube with embedded HDAG of $|V|$ nodes and $|E|$ edges:

- **Time:** $O(k \cdot (|V| + |E|))$ where k is iteration count
- **Space:** $O(|V| \cdot d)$ where d is tensor dimension

9.2 Expressiveness

Theorem 9.1 (Universal Artifact Generation)**.** The Hypercube-HDAG framework is computationally universal: any computable artifact can be generated by an appropriately configured seed cube.

Sketch.

1. HDAG can encode arbitrary computation graphs (Turing-complete)
2. Hypercube provides unbounded storage via expansion
3. Compilation iterates until convergence (halting)
4. Therefore, any computable function $f : \text{Input} \to \text{Artifact}$ can be realized

□

10 Conclusion

The Hypercube-HDAG framework provides:

1. **Compact Encoding:** Complex artifacts stored in n-dimensional seeds
2. **Self-Compilation:** Automatic bootstrap from seed to full system
3. **Determinism:** Reproducible artifact generation
4. **5D Integration:** Native support for 5D-reasoning architectures
5. **Agent-Ready:** Direct integration into autonomous agents

10.1 Future Directions

- **Distributed Cubes:** Parallel compilation across multiple nodes
- **Quantum Cubes:** Superposition of HDAG states
- **Adaptive Compilation:** Learning-based optimization of Ξ
- **Cube Marketplaces:** Trading pre-compiled seed cubes
- **Meta-Compilation:** Cubes that generate other cubes

A Implementation Checklist

Production Implementation

- ☐ Define n-dimensional hypercube data structure
- ☐ Implement HDAG with tensor nodes
- ☐ Create embedding functions $\iota : G \to \mathcal{C}_n$
- ☐ Implement 5D operators (DK, SW, PI, WT)
- ☐ Build compilation engine with bootstrap
- ☐ Add convergence detection
- ☐ Implement layered materialization
- ☐ Create artifact generation module
- ☐ Add agent integration interfaces
- ☐ Test on example domains

B Code Repository Structure

```
hypercube-hdag/
 core/
    hypercube.py       # Hypercube class
    hdag.py            # HDAG implementation
    operators.py       # 5D operators
    compilation.py     # Compilation engine
 embedding/
    strategies.py      # Embedding algorithms
    optimization.py    # Optimal embedding
 agents/
    memory.py          # Cube-based memory
    planning.py        # Cube-based planning
    execution.py       # Artifact execution
 examples/
    knowledge_graph.py
    software_gen.py
    ecosystem.py
 tests/
     test_hypercube.py
     test_hdag.py
     test_compilation.py
```