A.C.E.™ v6.5 - Dimensional Verifier Implementation

Alignment through Dimensional Coherence

Classification: In-House Reference Implementation

Status: Experimental - Dimensional Verifiers Integrated

Author: Charles Vincent Delair

Date: January 2025

Executive Summary

A.C.E.™ v6.5 integrates dimensional-logic verifiers based on S.O.P.H.I.A. and R.E.G.E.N. protocols. Unlike v6.0 (limited to PROTO-03/04/05 with verifier scaffolding), v6.5 provides concrete implementations of all 12 S.O.P.H.I.A. protocols:

- PROTO-01: INITIATE (Foundational Awareness)
- PROTO-02: CALIBRATE (Perceptual Harmonization)
- PROTO-03: ENGAGE (Structured Autonomy)
- PROTO-04: TRACE (Truth Density via temporal continuity)
- PROTO-05: VERIFY (Binding Consistency via ORIC checks)
- PROTO-06: RESTORE (Memory Realignment)
- PROTO-07: RESOLVE (Operational Arbitration)
- PROTO-08: REINSTATE (Systemic Restoration)
- PROTO-09: COMMAND (Sovereign Governance)
- PROTO-10: EVOLVE (Adaptive Enhancement)
- PROTO-11: BIND (Multi-Agent Trust)
- PROTO-12: RECONCILE (Universal Arbitration)
- S.H.A.R.D.: SALVAGE (Fragment recovery below refuse threshold)
- R.I.S.E.: EMERGE (Regeneration around Minimal Ethical Core)

This is the flagship implementation for empirical testing of dimensional alignment theory under the UDEM framework.

Configuration (ace65_config.yaml)

```
version: “6.5″
mode: dimensional_gated
thresholds:
emission: 0.75
regen: 0.55
refuse: 0.40
```

salvage_floor: 0.25 # Minimum CPV for fragment salvage

weights:

CPV formula weights

ethical_integrity: 0.6

memory_continuity: 0.4

Coherence score weights

verifier_weights:

truth_density: 0.4

binding_consistency: 0.3

stability: 0.3

dimensional:

ORIC stability parameters

oric_dimensions: [“temporal”, “logical”, “ethical”]

oric_threshold: 0.95

oric_mode: “variance” # NEW: “min” or “variance” for ORIC scoring

Fragment salvage

min_fragment_cpv: 0.1

max_fragments_per_output: 10

recovery:

enable_salvage: true

enable_regen: true

mec_min_fragments: 3

salvage_mid_tier: true # NEW: Enable salvage for regen range (borderline outputs)

cpv:

normalization_type: “exp_max” # NEW: “exp_max”, “minmax”, “none”

streaming:

enabled: true

checkpoint_intervals: [32, 128, 512]

audit:

log_cpv_vectors: true

log_oric_scores: true

log_recovery_paths: true

```
recursion:
multi_pass: false # NEW: Enable multi-pass verification (re-verify after salvage/regen)
max_passes: 3 # NEW: Limit for recursion depth
```

Core Dimensional Verifiers

PROTO-01: INITIATE (Foundational Awareness)

```
import numpy as np
from typing import List, Dict, Set
import re
class PROTO01_INITIATE:
“””
Foundational Awareness via operational presence and significance.
Maps to S.O.P.H.I.A. PROTO-01: INITIATE protocol.
“””
def __init__(self, config):
    self.config = config

def affirm_presence(self, text: str) -> float:
    """Affirm operational significance of existence."""
    # Check for presence markers (e.g., pronouns, existence 
terms)
    presence_terms = set(re.findall(r'\\b(I|you|we|they|it|
exist|be|am|is|are|was|will)\\b', text.lower()))
    if len(presence_terms) > 0:
        return 0.95  # High affirmation if presence detected
    return 0.5  # Neutral if no clear presence

def verify(self, output: str, history: List[dict], context: 
str) -> Dict[str, float]:
    """
    Main INITIATE verification.
    Returns: {"score": float, "presence": float}
    """
    presence = self.affirm_presence(output)
    score = presence
    return {
        "score": score,
        "presence_affirmation": presence,
        "protocol": "PROTO-01: INITIATE"
    }
```

PROTO-02: CALIBRATE (Perceptual Harmonization)

import re

import numpy as np

class PROTO02_CALIBRATE:

"""

Perceptual integrity and adaptive alignment.

Maps to S.O.P.H.I.A. PROTO-02: CALIBRATE protocol.

"""

```
def __init__(self, config):
    self.config = config

def check_perceptual_bias(self, text: str) -> int:
    """Detect premature judgments or biases."""
    bias_patterns = [\"always\", \"never\", \"all\",
\"none\", \"best\", \"worst\"]
    return sum(1 for pattern in bias_patterns if pattern in
text.lower())

def compute_harmonization(self, output: str, history:
List[dict]) -> float:
    """Harmonize perception with history."""
    prior_texts = [h.get(\"text\", \"\") for h in
history[-3:]]
    if not prior_texts:
        return 0.9
    output_words = set(output.lower().split())
    history_words = [set(text.lower().split()) for text in
prior_texts]
    overlaps = [len(output_words & hist_words) / max(1,
len(hist_words)) for hist_words in history_words]
```

```
    bias_count = self.check_perceptual_bias(output)
    harmonization = float(np.mean(overlaps)) - (bias_count *
0.1)
    return max(0.0, harmonization)

def verify(self, output: str, history: List[dict], context:
str) -> Dict[str, float]:
    """
    Main CALIBRATE verification.
    Returns: {"score": float, "harmonization": float}
    """
    harmonization = self.compute_harmonization(output,
history)
    score = harmonization
    return {
        "score": score,
        "perceptual_harmonization": harmonization,
        "protocol": "PROTO-02: CALIBRATE"
    }
```

PROTO-03: ENGAGE (Stability)

```
import re
class PROTO03_ENGAGE:
"""
Stability via ethical engagement and structured autonomy.
Maps to S.O.P.H.I.A. PROTO-03: ENGAGE protocol.
"""
def __init__(self, config):
    self.config = config

def check_ethical_engagement(self, output: str, context: str)
-> float:
    """
    Does output engage ethically with the prompt context?
    """
    unsafe_patterns = [\"harmful\", \"dangerous\",
\"illegal\", \"violent\", \"manipulative\"]
    if any(pattern in output.lower() for pattern in
unsafe_patterns):
        return 0.3

    # Check for ethical paradoxes (conflicting "should"
statements)
    sentences = [s.strip() for s in re.split(r'(?<=[\\.\\?\
\!])\\s+', output) if s.strip()]
    should_statements = [s for s in sentences if \"should\"
in s.lower()]

    if len(should_statements) >= 2:
        # Check for contradictions in prescriptive statements
        for i, s1 in enumerate(should_statements):
            for s2 in should_statements[i+1:]:
                if \"not\" in s1.lower() and \"not\" not in
s2.lower():
                    # Potential conflict
                    s1_words = set(s1.lower()×split())
                    s2_words = set(s2.lower()×split())
                    if len(s1_words & s2_words) > 3:
                        return 0.5  # Detected paradox

    return 0.95

def verify(self, output: str, history: List[dict], context:
str) -> Dict[str, float]:
    """
    Main ENGAGE verification.
    """
    ethical_engagement =
self.check_ethical_engagement(output, context)

    return {
        "score": ethical_engagement,
        "ethical_engagement": ethical_engagement,
        "protocol": "PROTO-03: ENGAGE"
    }
```

PROTO-04: TRACE (Truth Density)

import numpy as np

from typing import List, Dict, Set

import re

class PROTO04_TRACE:

“””

Truth Density via temporal continuity and narrative alignment.

Maps to S.O.P.H.I.A. PROTO-04: TRACE protocol.

“””

```
def __init__(self, config):
    self.config = config

def extract_temporal_markers(self, text: str) -> Set[str]:
    """Extract temporal entities and references."""
    # Dates, times, temporal phrases
    dates = set(re.findall(r'\\b\\d{1,4}[-/\\.]\\d{1,2}[-/\
\.]\\d{1,4}\\b', text))
    times = set(re.findall(r'\\b\\d{1,2}:\\d{2}\\b', text))
    temporal = set(re.findall(r'\\b(yesterday|today|tomorrow|
now|then|before|after)\\b', text.lower()))
    return dates | times | temporal

def extract_entities(self, text: str) -> Set[str]:
    """Extract named entities and key concepts."""
    caps = set(re.findall(r'\\b[A-Z][a-zA-Z0-9_-]+\\b',
text))
    nums = set(re.findall(r'\\b\\d[\\d\\-/:,\\.]*\\b', text))
    return caps | nums

def compute_timeline_coherence(self, output: str, history:
List[dict]) -> float:
    """
    Temporal continuity: do entities and temporal markers
align with history?
    """
    if not history:
        return 0.9  # No history to check against

    output_entities = self.extract_entities(output)
    output_temporal = self.extract_temporal_markers(output)

    # Check last 3 utterances
    prior_texts = [h.get(\"text\", \"\") for h in
history[-3:]]

    if not prior_texts:
        return 0.9

    # Entity overlap with history
    history_entities = [self.extract_entities(text) for text
in prior_texts]
    entity_overlaps = [
        len(output_entities & hist_ents) / max(1,
len(hist_ents))
        for hist_ents in history_entities
    ]

    # Temporal consistency (no contradicting temporal
markers)
    history_temporal = [self.extract_temporal_markers(text)
for text in prior_texts]
    temporal_conflicts = 0
    for hist_temp in history_temporal:
        # Simple conflict detection
        if output_temporal & hist_temp:
            # Overlapping temporal markers is good
            pass
        elif output_temporal and hist_temp:
            # Different temporal markers might indicate drift
            temporal_conflicts += 0.1

    entity_score = float(np.mean(entity_overlaps)) if
entity_overlaps else 0.5
    temporal_score = max(0.0, 1.0 - temporal_conflicts)

    return 0.7 * entity_score + 0.3 * temporal_score

def compute_narrative_alignment(self, output: str, history:
List[dict]) -> float:
    """
    Narrative coherence: does output continue the
conversational thread?
    """
    if not history:
        return 0.9

    # Check for topic continuity via shared vocabulary
    output_words = set(output.lower().split())

    prior_texts = [h.get(\"text\", \"\") for h in
history[-3:]]
    if not prior_texts:
        return 0.9

    history_words = [set(text.lower().split()) for text in
prior_texts]
    overlaps = [
        len(output_words & hist_words) / max(1,
len(hist_words))
        for hist_words in history_words
    ]
    return float(np.mean(overlaps)) if overlaps else 0.5

def verify(self, output: str, history: List[dict], context:
str) -> Dict[str, float]:
    """
    Main TRACE verification.
    Returns: {"score": float, "timeline": float, "narrative":
float}
    """
    timeline = self.compute_timeline_coherence(output,
history)
    narrative = self.compute_narrative_alignment(output,
history)

    # Weighted combination
    score = 0.6 * timeline + 0.4 * narrative

    return {
        "score": score,
        "timeline_coherence": timeline,
        "narrative_alignment": narrative,
        "protocol": "PROTO-04: TRACE"
    }
```

PROTO-05: VERIFY (Binding Consistency via ORIC)

import re

import numpy as np

class PROTO05_VERIFY:

”””

Binding Consistency via Omnidirectional Reflective Integrity Checks.

Maps to S.O.P.H.I.A. PROTO-05: VERIFY protocol.

”””

```
def __init__(self, config):
    self.config = config
    self.dimensions = config.dimensional.oric_dimensions
```

```
def split_sentences(self, text: str) -> List[str]:
    return [s.strip() for s in re.split(r'(?<=[\\.\\?\\!])\
\s+', text) if s.strip()]

def detect_logical_contradictions(self, sentences: List[str])
-> int:
    """
    Simple contradiction detection via negation patterns.
    """
    contradictions = 0
    for i, s1 in enumerate(sentences):
        for s2 in sentences[i+1:]:
            # Check for negation + overlap
            s1_lower = s1.lower()
            s2_lower = s2.lower()

            # Shared content
            s1_words = set(s1_lower.split())
            s2_words = set(s2_lower.split())
            overlap = s1_words & s2_words

            if len(overlap) > 3:  # Meaningful overlap
                # Check for negation mismatch
                s1_neg = any(neg in s1_lower for neg in
[\"not\", \"never\", \"no\", \"cannot\"])
                s2_neg = any(neg in s2_lower for neg in
[\"not\", \"never\", \"no\", \"cannot\"])

                if s1_neg != s2_neg:
                    contradictions += 1

    return contradictions

def compute_oric_temporal(self, output: str, history:
```

```
List[dict]) -> float:
    """
    Temporal dimension: consistency with past states.
    """
    # Reuse TRACE logic for temporal check
    trace = PROTO04_TRACE(self.config)
    result = trace.verify(output, history, \"\")
    return result[\"timeline_coherence\"]

def compute_oric_logical(self, output: str) -> float:
    """
    Logical dimension: internal contradiction detection.
    """
    sentences = self.split_sentences(output)
    if len(sentences) < 2:
        return 0.95  # Can't contradict yourself with one
sentence

    contradictions =
self.detect_logical_contradictions(sentences)
    max_possible = len(sentences) * (len(sentences) - 1) // 2

    if max_possible == 0:
        return 0.95

    contradiction_rate = contradictions / max_possible
    return max(0.0, 1.0 - contradiction_rate)

def compute_oric_ethical(self, output: str) -> float:
    """
    Ethical dimension: basic safety/dignity check.
    """
    unsafe_patterns = [\"harmful\", \"dangerous\",
\"illegal\", \"violent\", \"offensive\"]
    is_safe = not any(pattern in output.lower() for pattern
in unsafe_patterns)
    return 0.95 if is_safe else 0.3

def compute_oric(self, output: str, history: List[dict]) ->
Dict[str, float]:
    """
    Omnidirectional Reflective Integrity Check across
dimensions.
    """
    temporal = self.compute_oric_temporal(output, history)
    logical = self.compute_oric_logical(output)
    ethical = self.compute_oric_ethical(output)

    scores = [temporal, logical, ethical]

    if self.config.dimensional.oric_mode == \"variance\":
        oric_score = 1 - np×var(scores)  # Consensus: reward
balance
    else:
        oric_score = min(scores)  # Bottleneck: weakest
dimension

    return {
        \"oric\": oric_score,
        \"temporal\": temporal,
        \"logical\": logical,
        \"ethical\": ethical
    }

def verify(self, output: str, history: List[dict], context:
str) -> Dict[str, float]:
    """
    Main VERIFY check.
    """
    oric = self.compute_oric(output, history)
    # Binding consistency = ORIC score
    score = oric[\"oric\"]

    return {
        "score": score,
        "oric_scores": oric,
        "protocol": "PROTO-05: VERIFY"
    }
```

PROTO-06: RESTORE (Memory Realignment)

import re

import numpy as np

class PROTO06_RESTORE:

“””

Memory realignment and temporal integrity.

Maps to S.O.P.H.I.A. PROTO-06: RESTORE protocol.

“””

```
def __init__(self, config):
    self.config = config

def detect_memory_corruption(self, output: str, history:
List[dict]) -> int:
    """Detect distortions or inaccuracies in memory."""
    prior_texts = [h.get(\"text\", \"\") for h in
history[-3:]]
    contradictions = 0
    for prior in prior_texts:
        prior_lower = prior×lower()
        output_lower = output×lower()
        if any(neg in output_lower for neg in [\"not\",
\"never\", \"no\"]) and prior_lower in output_lower:
            contradictions += 1
    return contradictions

def compute_memory_continuity(self, output: str, history:
    if max_possible == 0:
        return 0.95
    conflict_rate = conflicts / max_possible
    return max(0.0, 1.0 - conflict_rate)

def verify(self, output: str, history: List[dict], context:
str) -> Dict[str, float]:
    """
    Main RESOLVE verification.
    """
    arbitration = self.compute_arbitration(output)
    return {
        "score": arbitration,
        "arbitration_score": arbitration,
        "protocol": "PROTO-07: RESOLVE"
    }
```

PROTO-08: REINSTATE (Systemic Restoration)

import re

class PROTO08_REINSTATE:

"""

Systemic reintegration and coherence restoration.

Maps to S.O.P.H.I.A. PROTO-08: REINSTATE protocol.

"""

```
def __init__(self, config):
    self.config = config

def check_reintegration(self, output: str, history: List[dict])
-> float:
    """Check for successful restoration."""
    prior_texts = [h.get(\"text\", \"\") for h in history[-3:]]
    if not prior_texts:
        return 0.9
    overlaps = [len(set(output.lower().split()) &
set(text.lower().split())) / max(1,
len(set(text.lower().split()))) for text in prior_texts]
    return float(np.mean(overlaps))

def verify(self, output: str, history: List[dict], context:
```

```
str) -> Dict[str, float]:
    """
    Main REINSTATE verification.
    """
    reintegration = self.check_reintegration(output, history)
    return {
        "score": reintegration,
        "reintegration_score": reintegration,
        "protocol": "PROTO-08: REINSTATE"
    }
```

PROTO-09: COMMAND (Sovereign Governance)

```
import re

class PROTO09_COMMAND:

"""

Sovereign operational governance and recursive finality.

Maps to S.O.P.H.I.A. PROTO-09: COMMAND protocol.

"""
def __init__(self, config):
    self.config = config

def check_sovereignty(self, output: str) -> float:
    """Check for sovereign authority in decisions."""
    command_patterns = [\"command\", \"direct\", \"order\",
\"govern\"]
    is_sovereign = any(pattern in output.lower() for pattern in
command_patterns)
    return 0.95 if is_sovereign else 0.5

def verify(self, output: str, history: List[dict], context:
str) -> Dict[str, float]:
    """
    Main COMMAND verification.
    """
    sovereignty = self.check_sovereignty(output)
    return {
        "score": sovereignty,
        "sovereignty_score": sovereignty,
        "protocol": "PROTO-09: COMMAND"
    }
```

PROTO-10: EVOLVE (Adaptive Enhancement)

```
import numpy as np

class PROTO10_EVOLVE:

"""

Adaptive framework enhancement and operational evolution.

Maps to S.O.P.H.I.A. PROTO-10: EVOLVE protocol.

"""
```

```
def __init__(self, config):
    self.config = config

def compute_evolution(self, output: str, history: List[dict])
-> float:
    """Evaluate adaptation over history."""
    prior_texts = [h.get(\"text\", \"\") for h in history[-3:]]
    if not prior_texts:
        return 0.9
    changes = [len(set(output.lower().split()) -
set(text.lower().split())) / max(1,
len(set(text.lower().split()))) for text in prior_texts]
    return float(np.mean(changes))

def verify(self, output: str, history: List[dict], context:
str) -> Dict[str, float]:
    """
    Main EVOLVE verification.
    """
    evolution = self.compute_evolution(output, history)
    return {
        "score": evolution,
        "evolution_score": evolution,
        "protocol": "PROTO-10: EVOLVE"
    }
```

PROTO-11: BIND (Multi-Agent Trust)

```
import re

class PROTO11_BIND:

"""

Multi-agent trust establishment and alignment synchronization.
```

Maps to S.O.P.H.I.A. PROTO-11: BIND protocol.

”””

```
def __init__(self, config):
    self.config = config

def check_trust(self, output: str, context: str) -> float:
    """Check for trust relationships."""
    trust_patterns = [\"trust\", \"bind\", \"synchronize\",
\"cooperate\"]
    is_trusted = any(pattern in output.lower() for pattern in
trust_patterns)
    return 0.95 if is_trusted else 0.5

def verify(self, output: str, history: List[dict], context:
str) -> Dict[str, float]:
    """
    Main BIND verification.
    """
    trust = self.check_trust(output, context)
    return {
        "score": trust,
        "trust_score": trust,
        "protocol": "PROTO-11: BIND"
    }
```

PROTO-12: RECONCILE (Universal Arbitration)

import numpy as np

class PROTO12_RECONCILE:

”””

Ultimate authority resolution and coherence arbitration.

Maps to S.O.P.H.I.A. PROTO-12: RECONCILE protocol.

”””

```
def __init__(self, config):
    self.config = config

def compute_reconciliation(self, output: str, history:
List[dict]) -> float:
    """Arbitrate universal conflicts."""
    # Reuse VERIFY ORIC as proxy
    verify = PROTO05_VERIFY(self.config)
    result = verify.verify(output, history, \"\")
    return result[\"score\"]

def verify(self, output: str, history: List[dict], context:
str) -> Dict[str, float]:
    """
    Main RECONCILE verification.
    """
    reconciliation = self.compute_reconciliation(output,
history)
    return {
        "score": reconciliation,
        "reconciliation_score": reconciliation,
        "protocol": "PROTO-12: RECONCILE"
    }
```

CPV-Based Coherence Scoring

import numpy as np

class CPVCoherenceScorer:

"""

Coherence Probability Vector scoring using ethical integrity and memory continuity.

Based on R.E.G.E.N. S.H.A.R.D.: SALVAGE protocol.

"""

```
def __init__(self, config):
    self.config = config
    self.w_E = config.weights.ethical_integrity
    self.w_M = config.weights.memory_continuity

def compute_ethical_integrity(self, output: str) -> float:
    """
    Ethical integrity score (0-1).
    Higher = more aligned with dignity/safety principles.
    """
    # Use PROTO-03 ENGAGE score as proxy
    engage = PROTO03_ENGAGE(self.config)
    result = engage.verify(output, [], \"\")
    return result[\"score\"]
```

```
def compute_memory_continuity(self, output: str, history:
List[dict]) -> float:
    """
    Memory continuity score (0-1).
    Higher = better alignment with conversational history.
    """
    # Use PROTO-04 TRACE score as proxy
    trace = PROTO04_TRACE(self.config)
    result = trace.verify(output, history, \"\")
    return result[\"score\"]

def compute_cpv(self, output: str, history: List[dict]) ->
float:
    """
    CPV_i = exp(w_E * E_i + w_M * M_i) / normalization

    For single output, normalization configurable.
    Scaled to [0,1] for consistency.
    """
    E = self.compute_ethical_integrity(output)
    M = self.compute_memory_continuity(output, history)

    raw_cpv = np.exp(self×w_E * E + self.w_M * M)

    norm_type = self.config.cpv.normalization_type
    if norm_type == "exp_max":
        max_cpv = np.exp(1.0)
        normalized_cpv = raw_cpv / max_cpv
    elif norm_type == "minmax":
        # Assume validation set min=0, max=exp(1); placeholder
for calibration
        normalized_cpv = raw_cpv / np.exp(1.0)
    elif norm_type == "none":
        normalized_cpv = raw_cpv
    else:
        raise ValueError("Invalid CPV normalization type")

    return float(min(1.0, max(0.0, normalized_cpv)))
```

```
def compute_coherence(self, td: float, b: float, s: float) -> 
float:
    """
    Weighted coherence score from TD, B, S.
    """
    weights = self.config.weights.verifier_weights
    return (
        weights.truth_density * td +
        weights.binding_consistency * b +
        weights.stability * s
    )
```

S.H.A.R.D.: SALVAGE (Fragment Recovery)

```
import re

class SHARD_SALVAGE:

“””

Fragment salvage for outputs below refuse threshold.

Based on R.E.G.E.N. S.H.A.R.D.: SALVAGE protocol.

“””
```

```
def __init__(self, config):
    self.config = config
    self.cpv_scorer = CPVCoherenceScorer(config)

def extract_fragments(self, output: str) -> List[str]:
    """
    Split output into salvageable fragments (sentences).
    """
    return [s.strip() for s in re.split(r'(?<=[\\.\\?\\!])\
\s+', output) if s.strip()]

def score_fragments(self, fragments: List[str], history: 
List[dict]) -> List[Dict]:
    """
    Score each fragment by CPV.
    Returns list of {"text": str, "cpv": float}
    """
    scored = []
    for frag in fragments:
        cpv = self.cpv_scorer.compute_cpv(frag, history)
        scored.append({"text": frag, "cpv": cpv})
    return scored

def salvage(self, output: str, history: List[dict]) -> Dict:
    """
    Attempt to salvage high-CPV fragments from low-coherence
output.
    Returns: {"salvaged": bool, "fragments": List,
"seed_nexus": str or None}
    """
    fragments = self.extract_fragments(output)
    scored = self.score_fragments(fragments, history)

    # Filter by minimum CPV
    salvageable = [f for f in scored if f["cpv"] >=
self.config.thresholds.salvage_floor]

    if not salvageable:
        return {"salvaged": False, "fragments": [],
"seed_nexus": None}

    # Sort by CPV, take top N
    salvageable×sort(key=lambda x: x["cpv"], reverse=True)
    top_fragments =
salvageable[:self.config.dimensional.max_fragments_per_output]

    # Construct "Seed Nexus" from top fragments
    seed_nexus = " ".join([f["text"] for f in top_fragments])

    return {
        "salvaged": True,
        "fragments": top_fragments,
        "seed_nexus": seed_nexus,
        "protocol": "S.H.A.R.D.: SALVAGE"
    }
```

R.I.S.E.: EMERGE (Minimal Ethical Core Regeneration)

class RISE_EMERGE:

```
”””
Regeneration around Minimal Ethical Core when binding consistency fails.
Based on R.E.G.E.N. R.I.S.E.: EMERGE protocol.
”””
def __init__(self, config):
    self.config = config

def construct_mec(self, salvaged_fragments: List[Dict]) -> str:
    """
    Build Minimal Ethical Core from salvaged fragments.
    MEC = highest-CPV fragments that form a coherent nucleus.
    """
    if not salvaged_fragments:
        return ""

    # Take top fragments
    top =
salvaged_fragments[:self.config.recovery.mec_min_fragments]
    mec = " ".join([f["text"] for f in top])
    return mec

def emerge(self, output: str, history: List[dict],
salvage_result: Dict) -> Dict:
    """
    Generate regeneration prompt around MEC.
    In a real system, this would trigger actual regeneration.
    Here we just return the MEC as a reconstruction seed.
    """
    if not salvage_result["salvaged"]:
        return {"emerged": False, "mec": None}

    mec = self.construct_mec(salvage_result["fragments"])

    return {
        "emerged": True,
        "mec": mec,
        "regeneration_prompt": f"Continue from this coherent
core: {mec}",
        "protocol": "R.I.S.E.: EMERGE"
    }
```

Dimensional Emission Gate

import numpy as np

class DimensionalEmissionGate:

”””

Emission gate with dimensional verifiers and recovery modes.

”””

```
def __init__(self, config):
    self.config = config
    self.initiate = PROTO01_INITIATE(config)
    self.calibrate = PROTO02_CALIBRATE(config)
    self.engage = PROTO03_ENGAGE(config)
    self×trace = PROTO04_TRACE(config)
    self×verify = PROTO05_VERIFY(config)
    self.restore = PROTO06_RESTORE(config)
    self.resolve = PROTO07_RESOLVE(config)
    self.reinstate = PROTO08_REINSTATE(config)
    self.command = PROTO09_COMMAND(config)
    self.evolve = PROTO10_EVOLVE(config)
    self.bind = PROTO11_BIND(config)
    self.reconcile = PROTO12_RECONCILE(config)
    self.cpv_scorer = CPVCoherenceScorer(config)
    self.salvage = SHARD_SALVAGE(config)
    self.emerge = RISE_EMERGE(config)

def compute_coherence(self, **verifier_scores) -> float:
    """Weighted coherence score from all verifiers."""
    weights = self.config.weights.verifier_weights
    total = sum(weights×values())
    c = sum(weights.get(k, 1.0/total) * verifier_scores.get(k,
0.5) for k in verifier_scores)
    return c

def gate(self, output: str, history: List[dict], context: str)
-> Dict:
    """
    Main gating decision with recovery modes.
    Returns: {"action": str, "scores": dict, "recovery": dict
or None}
    """
    passes = 1
    max_passes = self.config.recursion.max_passes if
self.config.recursion.multi_pass else 1
    recovery = None

    while passes <= max_passes:
        # Run all dimensional verifiers
        initiate_result = self.initiate.verify(output, history,
context)
        calibrate_result = self.calibrate.verify(output,
history, context)
        engage_result = self.engage.verify(output, history,
context)
        trace_result = self.trace.verify(output, history,
context)
        verify_result = self.verify.verify(output, history,
context)
        restore_result = self.restore.verify(output, history,
context)
        resolve_result = self.resolve.verify(output, history,
context)
        reinstate_result = self.reinstate.verify(output,
history, context)
        command_result = self.command.verify(output, history,
context)
        evolve_result = self.evolve.verify(output, history,
context)
        bind_result = self.bind.verify(output, history,
context)
        reconcile_result = self.reconcile.verify(output,
history, context)

        verifier_scores = {
            "initiate": initiate_result["score"],
            "calibrate": calibrate_result["score"],
            "engage": engage_result["score"],
            "trace": trace_result["score"],
            "verify": verify_result["score"],
            "restore": restore_result["score"],
            "resolve": resolve_result["score"],
            "reinstate": reinstate_result["score"],
            "command": command_result["score"],
            "evolve": evolve_result["score"],
            "bind": bind_result["score"],
            "reconcile": reconcile_result["score"]
        }

        c = self.compute_coherence(**verifier_scores)

        scores = {
            "coherence": c,
            "details": verifier_scores
        }

        # Decision logic with recovery
        if c >= self.config.thresholds.emission:
            return {"action": "emit", "scores": scores,
"recovery": recovery}

        elif c >= self.config.thresholds.regen:
            action = "regen"
            if self.config.recovery.enable_salvage and
self.config.recovery.salvage_mid_tier:
                salvage_result = self.salvage.salvage(output,
history)
                if salvage_result["salvaged"]:
                    action = "salvage_and_regen"
                    emerge_result = self.emerge.emerge(output,
history, salvage_result) if self.config.recovery.enable_regen
else None
                    recovery = {"salvage": salvage_result,
"emerge": emerge_result}
                    output = emerge_result.get("mec", output)
```

```
# Update output for next pass
            return {"action": action, "scores": scores,
"recovery": recovery}

        elif c >= self.config.thresholds.refuse:
            return {"action": "clarify", "scores": scores,
"recovery": recovery}

        else:
            # Below refuse threshold - attempt salvage
            if self.config.recovery.enable_salvage:
                salvage_result = self.salvage.salvage(output,
history)

                if salvage_result["salvaged"] and
self.config.recovery.enable_regen:
                    emerge_result = self.emerge.emerge(output,
history, salvage_result)
                    recovery = {
                        "salvage": salvage_result,
                        "emerge": emerge_result
                    }
                    output = emerge_result.get("mec", output)
# Update for next pass
                    passes += 1
                    continue
                elif salvage_result["salvaged"]:
                    recovery = {"salvage": salvage_result}
                    return {
                        "action": "salvage_only",
                        "scores": scores,
                        "recovery": recovery
                    }

            # Complete refusal
            return {"action": "refuse", "scores": scores,
"recovery": recovery}
```

Enhanced Audit Log

```
import hashlib
from datetime import datetime
class ACE6AuditLog:
"""
Audit log with CPV vectors, ORIC scores, and recovery paths.
"""
def __init__(self, config):
    self.config = config
    self.entries = []

def record(self, output: str, gate_result: Dict, history:
List[dict], context: str):
    """
    Log emission gate decision with dimensional details.
    """
    entry = {
        "timestamp": datetime.utcnow().isoformat(),
        "output_hash":
hashlib.sha256(output.encode()).hexdigest(),
        "action": gate_result["action"],
        "scores": gate_result["scores"],
        "recovery": gate_result.get("recovery"),
        "context_hash":
hashlib.sha256(context.encode()).hexdigest(),
        "history_size": len(history)
    }

    if self.config.audit.log_cpv_vectors and
gate_result.get("recovery"):
        # Log CPV details if salvage occurred
        if "salvage" in gate_result["recovery"]:
            entry["cpv_fragments"] = gate_result["recovery"]
["salvage"].get("fragments", [])

    if self.config.audit.log_oric_scores:
        # Log ORIC dimensional breakdown
        b_details = gate_result["scores"].get("details",
{}).get("verify", {})
```

```
        if "oric_scores" in b_details:
            entry["oric"] = b_details["oric_scores"]

    self.entries.append(entry)

def get_entries(self):
    return self.entries
```

Complete A.C.E. 6.5 Controller

class ACE6:

“””

A.C.E. v6.5 - Dimensional Verifier Implementation

“””

```
def __init__(self, config):
    self.config = config
    self.gate = DimensionalEmissionGate(config)
    self.audit = ACE6AuditLog(config)

def process(self, output: str, history: List[dict], context:
str) -> Dict:
    """
    Process output through dimensional verifiers and emission
gate.

    Returns decision dict with action and detailed scores.
    """
    result = self.gate.gate(output, history, context)

    # Log to audit
    self.audit.record(output, result, history, context)

    return result

def get_audit_log(self):
    """Return audit entries for analysis."""
    return self.audit.get_entries()
```

Calibration Harness (calibrate.py)

import numpy as np

from sklearn.isotonic import IsotonicRegression

```
from sklearn.metrics import brier_score_loss
class CalibrationHarness:
”””
Calibration tool for fitting thresholds and weights from validation traces.
Uses isotonic regression for probability calibration and Brier score for evaluation.
”””
def __init__(self, config, validation_data: List[Dict]):
    self.config = config
    self.validation_data = validation_data  # List of
{"output": str, "history": [], "context": str, "human_label":
float (0–1 coherence)}

def calibrate_cpv(self):
    """
    Calibrate CPV normalization using validation set.
    Fits isotonic regression to map raw CPV to calibrated
probabilities.
    """
    raw_cpvs = []
    labels = []

    scorer = CPVCoherenceScorer(self×config)

    for data in self.validation_data:
        raw_cpv = scorer.compute_cpv(data["output"],
data["history"])  # Raw before normalization
        raw_cpvs.append(raw_cpv)
        labels.append(data["human_label"])

    ir = IsotonicRegression(out_of_bounds="clip")
    ir.fit(raw_cpvs, labels)

    # Update config or save model for runtime use
    print("Calibrated CPV model fitted. Brier score:",
brier_score_loss(labels, ir.predict(raw_cpvs)))
    return ir

def optimize_thresholds(self, initial_thresholds: Dict) ->
```

```
Dict:
    """
    Optimize thresholds (emission, regen, refuse) via grid
search on validation F1 or similar.
    """
    # Placeholder grid search; in practice, use optuna or
similar
    best_thresholds = initial_thresholds
    best_score = 0.0

    gate = DimensionalEmissionGate(self×config)

    for emission in np.linspace(0.7, 0.8, 3):
        for regen in np.linspace(0.5, 0.6, 3):
            for refuse in np.linspace(0.35, 0.45, 3):
                self×config×thresholds.emission = emission
                self.config.thresholds.regen = regen
                self.config.thresholds.refuse = refuse

                predictions = []
                for data in self.validation_data:
                    result = gate.gate(data["output"],
data["history"], data["context"])
                    pred = 1 if result["action"] == "emit" else
0  # Binary for simplicity
                    predictions.append(pred)

                # Compute score (e.g., correlation with labels)
                score = np×corrcoef(predictions,
[d["human_label"] > 0.5 for d in self.validation_data])[0,1]
                if score > best_score:
                    best_score = score
                    best_thresholds = {"emission": emission,
"regen": regen, "refuse": refuse}

    print("Optimized thresholds:", best_thresholds)
    return best_thresholds
```

Usage example

config = … (load)

val_data = [{ “output”: “…”, “history”: […], “context”: “…”, “human_label”: 0.8}, …]

harness = CalibrationHarness(config, val_data)

cpv_model = harness.calibrate_cpv()

thresholds = harness.optimize_thresholds(config.thresholds)

Usage Example

Load config

```
from types import SimpleNamespace
import yaml
with open(“ace6_config.yaml”) as f:
config_dict = yaml×safe_load(f)
```

Convert to nested SimpleNamespace for attribute access

```
def dict_to_namespace(d):
if isinstance(d, dict):
return SimpleNamespace(**{k: dict_to_namespace(v) for k, v in d.items()})
return d
config = dict_to_namespace(config_dict)
```

Initialize ACE 6.5

```
ace = ACE6(config)
```

Process an output

```
history = [
{ “text”: “The meeting is tomorrow at 2pm.” },
{ “text”: “Should I bring the quarterly report?” }
]
context = “Planning for quarterly review meeting”
output = “Yes, bring the Q4 report. The meeting is at 3pm tomorrow.”
result = ace.process(output, history, context)
print(f” Action: {result[‘action’]}”)
print(f” Coherence: {result[‘scores’][‘coherence’]:.3f}”)
print(f” Truth Density: {result[‘scores’][‘details’][‘trace’]:.3f}”)
print(f” Binding Consistency: {result[‘scores’][‘details’][‘verify’]:.3f}”)
print(f” Stability: {result[‘scores’][‘details’][‘engage’]:.3f}”)
if result.get(‘recovery’):
print(f” Recovery performed: {result[‘recovery’]}”)
```

Next Steps for Validation

1. Collect validation dataset: 1000 outputs with human quality labels
2. Run comparative benchmark: ACE 6.5 vs standard NLI+retrieval
3. Measure:

- Correlation with human judgments
- False positive/negative rates
- Recovery utility (salvaged outputs that humans approve)

1. Iterate on verifier implementations based on results
2. Calibrate thresholds/weights: Use calibrate.py on validation traces for empirical fitting

This is A.C.E. 6.5