Thought for a couple of seconds

Vault Engineering Specification for Kimera SWM

Version: v1.0
Date: 2025-06-05

This document specifies the engineering details for the **Vault** subsystem in Kimera SWM. It covers memory structures, data schemas, routing logic, threshold values, and pseudocode for core algorithms. All speculative commentary has been removed; only concrete engineering constructs remain.

Table of Contents

---

1. Overview

The **Vault** subsystem stores, manages, and processes **Scars** (immutable contradiction records) generated during inference. It consists of two parallel vault instances—**Vault-A** and **Vault-B**—to distribute load and maintain semantic balance. Core functions include routing Scars, balancing entropy, resolving overlaps, handling fractures under load, and optimizing memory.

---

2. Data Structures & Schemas

2.1 Scar Schema

Each Scar is stored as a node with the following JSON structure:

```
{
  "scarID": "SCAR_456",                    // string, unique identifier
  "geoids": ["GEOID_123", "GEOID_789"],    // array of strings
  "reason": "Conflict: pref_color blue vs red", // string
  "timestamp": "2025-05-27T12:05:00Z",     // xsd:dateTime
  "resolvedBy": "consensus_module",        // string
  "pre_entropy": 0.67,                     // float
  "post_entropy": 0.82,                    // float
  "delta_entropy": 0.15,                   // float
  "cls_angle": 45.0,                       // float, collapse line shape angle in degrees
  "semantic_polarity": 0.2,                // float [-1.0, 1.0], sign-based polarity
  "originVault": "A",                      // "A" or "B"
  "expression": { /* feature vector or JSON map */ }
}
```

- **scarID**: Unique Scar identifier.
- **geoids**: IDs of Geoids involved.
- **reason**: Text description.
- **timestamp**: ISO 8601.
- **resolvedBy**: Module or process name that resolved any conflict.
- **pre_entropy**, **post_entropy**, **delta_entropy**: Semantic entropy metrics.
- **cls_angle**: Collapse Line Shape torsion angle (degrees).
- **semantic_polarity**: Scalar polarity value.
- **originVault**: Indicates initial vault (A or B).
- **expression**: Detailed feature representation.

2.2 Vault Metadata

Each vault maintains counters and metrics, stored in a metadata document:

```
{
  "vaultID": "Vault-A",                  // "Vault-A" or "Vault-B"
  "totalScars": 10234,                     // integer
  "activeScars": 2876,                     // integer
  "entropySum": 1523.8,                      // float, sum of semantic entropy of active Scars
  "avg_cls_angle": 47.2,                   // float, average CLS angle
  "incomingLoadLastCycle": 125,              // integer
  "outgoingLoadLastCycle": 118,              // integer
  "frictionMetric": 0.34                   // float [0.0, 1.0], averaged MFG
}
```

---

3. Vault Topology

3.1 Dual Vault Activation

Upon system startup, instantiate two vaults:

```
vaultA = Vault(id="Vault-A")
vaultB = Vault(id="Vault-B")
```

Both vaults register with a **Vault Manager** responsible for routing incoming Scars.

3.2 Partitioning Criteria

When a new Scar `s` arrives, compute routing decision based on:

1. **Mutation Frequency (MF):**
    - If `s.mutationFrequency` > `MF_threshold_high`, route to Vault-A; else route to Vault-B.
    - `MF_threshold_high = 0.75` (normalized frequency).
2. **Semantic Polarity (SP):**
    - If `abs(s.semantic_polarity) > 0.5`, route to vault determined by sign: positive → Vault-A; negative → Vault-B.

3. **CLS Torsion Signature (CLS):**
    - If `|s.cls_angle - vaultA.avg_cls_angle| < |s.cls_angle - vaultB.avg_cls_angle|`, route to Vault-A; else to Vault-B.

Routing Pseudocode:

```
def route_scar(scar):
    # 1. Mutation Frequency check
    if scar.mutationFrequency > 0.75:
        return vaultA
    # 2. Semantic Polarity
    if abs(scar.semantic_polarity) > 0.5:
        return vaultA if scar.semantic_polarity > 0 else vaultB
    # 3. CLS angle proximity
    diffA = abs(scar.cls_angle - vaultA.meta["avg_cls_angle"])
    diffB = abs(scar.cls_angle - vaultB.meta["avg_cls_angle"])
    return vaultA if diffA <= diffB else vaultB
```

3.3 Vault Interference Fields

Each vault maintains an **Interference Matrix** to log cross-vault interactions:

- **Echo Interference Index (EII):** Correlation coefficient between recent `echoAmplitude` time series of Vault-A and Vault-B.
- **Scar Overlap Zones (SOZ):** Tracks pairs of Scar IDs (one from each vault) with feature overlap > 0.9.
- **Entropic Drift Direction (EDD):** Difference in `entropySum` between vaults; `EDD = vaultA.entropySum - vaultB.entropySum`.

Brick these fields into a shared structure:

```
interference = {
    "EII": 0.12,      # float [-1.0, 1.0]
    "SOZ": [          # list of tuples
        ("SCAR_101", "SCAR_202"),
        ("SCAR_305", "SCAR_406"),
        // ...
    ],
    "EDD": 42.5       # float
}
```

---

4. Contradiction Drift Interpolator

4.1 Entropy Balance

Periodically (every cycle), compute:

```
S_A = vaultA.meta["entropySum"]
S_B = vaultB.meta["entropySum"]
delta_S = abs(S_A - S_B)
ENTROPY_THRESHOLD = 0.26

if delta_S > ENTROPY_THRESHOLD:
    # Divert new Scars to lower-entropy vault
    vaultManager.set_preferred(vaultA if S_A < S_B else vaultB)
else:
    vaultManager.clear_preference()
```

4.2 Memory Friction Gradient

For a Scar `s` attempting to move between vaults:

$$\text{MFG} = \alpha \times \bigl|\theta_A - \theta_B\bigr| + \beta \times \bigl|S_A - S_B\bigr|$$

- $\alpha = 0.7$
- $\beta = 0.3$
- θ_A, θ_B: vaults' average CLS angles (degrees).
- S_A, S_B: vaults' entropy sums.

If `MFG > 0.5`, delay insertion by one cycle:

```
def attempt_move(scar, target_vault):
    thetaA = vaultA.meta["avg_cls_angle"]
    thetaB = vaultB.meta["avg_cls_angle"]
    SA = vaultA.meta["entropySum"]
    SB = vaultB.meta["entropySum"]

    mfg = 0.7 * abs(thetaA - thetaB) + 0.3 * abs(SA - SB)
    if mfg > 0.5:
        scar.delay += 1
        if scar.delay >= 2:
            scar.delay = 0
            target_vault.insert(scar)
```

```
    else:
        target_vault.insert(scar)
```

4.3 Priority Interrupt Logic

When two Scars `s1` and `s2` arrive simultaneously and $|s1.clsangle - s2.clsangle| < 15°$:

1. Compare `timestamp`; older scar gets processed first.
2. Newer scar goes to overflow queue for next cycle.

```
def handle_simultaneous(scar_list):
    scar_list.sort(key=lambda s: s.timestamp)
    primary = scar_list[0]
    secondary = scar_list[1]
    vault = route_scar(primary)
    vault.insert(primary)
    overflow_queue.enqueue(secondary)
```

4.4 Echo Contamination & Quarantine

When an echo returns to a vault after bouncing:

1. Compute **friction score** $F = 1 - |s.clsangle - vault.avgclsangle| / 180$.
2. If $F < 0.68$, mark echo as “tainted” and hold in quarantine for 1 cycle.
3. After 1 cycle, re-evaluate; if still tainted, drop or force adjust.

```
def process_returned_echo(echo, vault):
    theta_v = vault.meta["avg_cls_angle"]
    F = 1 - abs(echo.cls_angle - theta_v) / 180
    if F < 0.68:
        echo.quarantine_cycles += 1
        if echo.quarantine_cycles >= 1:
            # Retrial next cycle
            echo.quarantine_cycles = 0
            vault.insert(echo)
    else:
        vault.insert(echo)
```

4.5 Scar Delay Watchdog

For each Scar `s` delayed by `delay` > 2 cycles:

- **Torsion Burst:** Ignore MFG and force insertion.
- **Semantic Decay:** Reduce each feature weight by 5%:
  pi←0.95×pi∀ ip_i \leftarrow 0.95 \times p_i \quad \forall\,i
  Recompute `cls_angle` and re-attempt insertion.

```
def delay_watchdog(scar, vault):
    if scar.delay > 2:
        # Option A: Burst
        vault.insert(scar)
        scar.delay = 0
    elif scar.delay == 2:
        # Option B: Semantic decay
        for k in scar.expression:
            scar.expression[k] *= 0.95
        scar.cls_angle = recompute_cls(scar.expression)
        vault.insert(scar)
        scar.delay = 0
```

4.6 Vault Entropy Purge

When a vault's **incomingBuffer** size > 3:

1. Identify Scar with lowest **delta_entropy**.
2. Remove it (mark as "purged").
3. Initiate an "echo vacuum" by blocking new scars for 0.5 cycles.

```
def vault_entropy_purge(vault):
    buffer = vault.incoming_buffer
    if len(buffer) > 3:
        # Find lowest delta_entropy
        victim = min(buffer, key=lambda s: s.delta_entropy)
        vault.purge(victim)
        vault.block_new = True
        vault.block_cycles = 1  # 1 cycle = 0.5 of real time unit
```

---

5. Recursive Vault Reflex Engine

5.1 Temporal Reflection Divergence

Each Scar `s` in both vaults has `timestampA` and `timestampB`. If $\Delta T = |\text{timestampA} - \text{timestampB}| > 2$ cycles:

- Mark `s.divergent = True`.
- Immediately apply a lightweight mutation: append `"_mut"` to `s.scarID` and update `expression`.

```
def check_divergence(scar):
    dt = abs(scar.timestampA - scar.timestampB)
    if dt > 2:
        scar.divergent = True
        scar.scarID += "_mut"
        scar.expression = mutate_expression(scar.expression)
```

5.2 Scar Echo Overlap Resolution

For every pair `(s1, s2)` where `s1` in Vault-A and `s2` in Vault-B:

$$\text{SRV} = \frac{|\,\text{features}(s1) \cap \text{features}(s2)\,|}{|\,\text{features}(s1) \cup \text{features}(s2)\,|}$$

If `SRV > 0.78`:

1. Merge both Scars into new `s_new`:
    - `s_new.expression = merge_features(s1.expression, s2.expression)`
    - `s_new.scarID = "SCAR_M_" + s1.scarID + "_" + s2.scarID`
    - `s_new.timestamp = max(s1.timestamp, s2.timestamp)`
2. Remove `s1` and `s2` from both vaults; insert `s_new` into Vault-A (arbitrary choice).

```
def resolve_overlap(s1, s2):
    overlap = compute_srv(s1.expression, s2.expression)
    if overlap > 0.78:
```

```
    merged_expr = merge_features(s1.expression, s2.expression)
    new_id = f"SCAR_M_{s1.scarID}_{s2.scarID}"
    s_new = Scar(
        scarID=new_id,
        geoids=list(set(s1.geoids + s2.geoids)),
        reason="Merged overlap",
        timestamp=max(s1.timestamp, s2.timestamp),
        expression=merged_expr,
        cls_angle=recompute_cls(merged_expr),
        semantic_polarity=(s1.semantic_polarity + s2.semantic_polarity) / 2
    )
    vaultA.remove(s1); vaultB.remove(s2)
    vaultA.insert(s_new)
```

5.3 Conflict Recompression Channel

When two Scars `s1` and `s2` have `SRV > 0.78` and both remain active after previous steps:

1. **Echo Bifurcation:** Split `s1.expression` into two subsets `exprA` and `exprB` (e.g., half of the features each).

Identity Fork Generation: Create `sA` and `sB`:

```
 sA = clone_scar(s1, suffix="_A", expression=exprA)
sB = clone_scar(s1, suffix="_B", expression=exprB)
```

2.
3. **Scarline Cross-Fade:** Over 2 cycles, reduce weight of original `s1` by 50% each cycle and increase `sA`/`sB` weights accordingly.

4. After 2 cycles, remove `s1` entirely; keep `sA` and `sB`.

```
def recompress_conflict(s1):
    exprA, exprB = split_features(s1.expression)
    sA = clone_scar(s1, suffix="_A", expression=exprA)
    sB = clone_scar(s1, suffix="_B", expression=exprB)
    for cycle in range(2):
        s1.weight *= 0.5
        sA.weight += 0.25  # accumulate half over 2 cycles
        sB.weight += 0.25
        wait_one_cycle()
    vaultA.remove(s1)
    vaultA.insert(sA)
    vaultA.insert(sB)
```

5.4 Divergence Weight Decay Function

For any Scar s after Δ cycles from its last insertion:

$$\text{weight} = \text{initial_weight} \times e^{-0.22 \times \Delta}$$

Implement decay at each cycle:

```
def apply_weight_decay(scar, cycles_elapsed):
    scar.weight = scar.initial_weight * math.exp(-0.22 * cycles_elapsed)
```

5.5 Scar Remnant Log

All removed or recompressed Scars are recorded in a separate **ScarRemnantLog** with:

```
{
  "scarID": "SCAR_789",
  "originVault": "Vault-B",
  "collapseAngle": 42.0,
  "overlapSRV": 0.82,
  "removalCycle": 15
}
```

Logging pseudocode:

```
def log_remnant(scar, cause, cycle):
    entry = {
        "scarID": scar.scarID,
        "originVault": scar.originVault,
        "collapseAngle": scar.cls_angle,
        "overlapSRV": compute_overlap_metric(scar),
        "removalCycle": cycle,
        "cause": cause
    }
    scar_remnant_log.append(entry)
```

5.6 Identity Distortion Index

Compute:

IDI=1−e−λ×reflections,λ=0.22\text{IDI} = 1 - e^{-\lambda \times \text{reflections}} \quad,\quad \lambda = 0.22

- **reflections**: Number of times the Scar has hopped between vaults.

```
def compute_idi(scar):
    return 1 - math.exp(-0.22 * scar.reflection_count)

def check_idi(scar):
    idi = compute_idi(scar)
    if idi > 0.72:
        scar.quarantined = True
        vaultQuarantine.insert(scar)
```

---

6. Vault Fracture Topology

6.1 Fracture Triggers & Handling

A **fracture** occurs when a vault's active load exceeds thresholds:

- **VSI (Vault Stress Index):**
  VSI=activeScarscapacity(capacity≈10000 scars)\text{VSI} = \frac{\text{activeScars}}{\text{capacity}} \quad (\text{capacity} \approx 10000 \text{ scars})
  If `VSI > 0.8`, trigger a fracture.

- **Fracture Procedure:**

  1. **Lock Vault-A and Vault-B** (pause new insertions).
  2. **Identify High-Tension Scars:** Select top 10% by `delta_entropy`.
  3. **Reroute 20% of those Scars** to a **symbolic fallback queue** outside both vaults.
  4. **Mark fracture event** in `VaultFractureLog`.
  5. **Resume vault operations** after 3 cycles of isolation.

```
def trigger_fracture(vault):
    VSI = vault.activeScars / 10000
    if VSI > 0.8:
        vault.locked = True
        high_tension = sorted(vault.activeScars_list, key=lambda s: s.delta_entropy,
reverse=True)
        top_10 = high_tension[: int(0.1 * len(high_tension))]
        reroute_count = int(0.2 * len(top_10))
        for scar in top_10[:reroute_count]:
            vault.remove(scar)
            fallback_queue.enqueue(scar)
        log_fracture_event(vault.id, current_cycle)
        vault.isolation_cycles = 3
```

6.2 Fracture Metrics

Log each fracture with:

```
{
  "fractureID": "FVN-031",
  "vaultID": "Vault-A",
  "activeScarsBefore": 9234,
  "activeScarsAfter": 7360,
  "reroutedScars": 347,
  "isolationCycles": 3,
  "timestamp": "2025-06-05T11:00:00Z"
}
```

6.3 Post-Fracture Reintegration

After `isolationCycles` elapse:

1. **Unlock vault** and allow insertions.
2. **Process fallback queue** at a throttled rate (max 50 scars/cycle).
3. **Update vault metadata** accordingly.

```
def end_fracture_cycle(vault):
    vault.locked = False
    for _ in range(min(50, len(fallback_queue))):
        scar = fallback_queue.dequeue()
        vault.insert(scar)
    vault.update_metadata()
```

---

7. Vault Optimization & Memory Management

7.1 Optimization Triggers

Initiate optimization when **any** of the following are true:

Metric	Threshold
Drift Lineage Depth	> 12 for ≥ 10% of active Scars
Scar Density	> 25 new Scars per 100 cycles
Vault Entropy Slope	> 0.05 increase over last 500 cycles
Identity Thread Saturation	> 85% overlap among active Scar groups
Loop Memory Pressure	> 90% of symbolic storage capacity

```
def should_optimize(vault):
    cond1 = vault.max_drift_depth > 12 and vault.percent_drifts_high > 0.10
    cond2 = vault.newScarsLast100 > 25
    cond3 = vault.entropySlope > 0.05
    cond4 = vault.threadOverlapPercent > 0.85
    cond5 = vault.memoryUsagePercent > 0.90
    return any([cond1, cond2, cond3, cond4, cond5])
```

7.2 Optimization Operations

7.2.1 Drift Collapse Pruning

Remove Scars with:

- `drift_depth > 12`
- `loop_active = False`
- `goal_impact = 0`

```
def prune_drift_clusters(vault):
    candidates = [
        s for s in vault.activeScars_list
        if s.drift_depth > 12 and not s.loop_active and s.goal_impact == 0
    ]
```

```
    for s in candidates:
        vault.remove(s)
        log_pruned_scar(s.scarID, current_cycle)
```

7.2.2 Composite Compaction

Identify low-entropy clusters (`entropy < 0.43`) with `cluster_size < 5` and merge into **latent patterns**.

```
def composite_compaction(vault):
    clusters = vault.get_clusters()  # returns lists of related scars
    for cluster in clusters:
        if average_entropy(cluster) < 0.43 and len(cluster) < 5:
            lp = create_latent_pattern(cluster)
            for s in cluster:
                vault.remove(s)
            vault.insert(lp)
            log_compaction(cluster, lp.latentID, current_cycle)
```

7.2.3 Vault Reindexing

```
def reindex_vault(vault):
    # Rebuild graph indices:
    vault.graph_db.rebuild_index("scarID")
    vault.graph_db.rebuild_index("cls_angle")
    vault.graph_db.rebuild_index("timestamp")
```

7.2.4 Influence-Based Retention Scoring

Compute for each Scar:

$$\text{IRS} = \frac{\text{loop_influence} \times \text{goal_contribution} \times \text{anchor_coupling}}{\text{entropy_decay}}$$

- Remove if `IRS < 0.12`.

```
def compute_irs(scar):
    numerator = scar.loop_influence * scar.goal_contribution * scar.anchor_coupling
    denominator = scar.entropy_decay or 1e-6
    return numerator / denominator

def retention_scoring(vault):
    for s in vault.activeScars_list:
        s.irs = compute_irs(s)
        if s.irs < 0.12:
            vault.archive(s)
            log_archived_scar(s.scarID, s.irs, current_cycle)
```

7.2.5 Memory Compression

1. **Low-Dimensional Drift Embedding:** Collapse drift trail into a 5-element vector.

2. **Zone Batching:** Archive inactive zones older than 30 cycles into monthly snapshots.

3. **Contradiction De-Duplication:** For Scars with identical `expression` hash, keep one and link references.

```
def compress_memory(vault):
    # (1) Drift Embedding
    for s in vault.activeScars_list:
        s.drift_vector = low_dim_embedding(s.drift_trace)
    # (2) Zone Batching
    for zone in vault.zones:
        if zone.lastActiveCycle < current_cycle - 30:
            vault.archive_zone(zone)
    # (3) De-Duplication
    expr_map = {}
    for s in vault.activeScars_list:
        key = hash_expression(s.expression)
        if key in expr_map:
            vault.merge_scar(expr_map[key], s)
        else:
            expr_map[key] = s
```

7.2.6 Audit Reporting

Generate JSON report:

```
{
```

```
  "optimizationID": "vault-opt-001",
  "timestamp": "2025-06-05T11:30:00Z",
  "prunedCount": 128,
  "compactedCount": 64,
  "archivedCount": 172,
  "memoryReductionPercent": 27.4,
  "activeScarsRemain": 8503
}

def generate_opt_report(vault, pruned, compacted, archived):
    report = {
        "optimizationID": f"vault-opt-{vault.nextOptID()}",
        "timestamp": current_iso_time(),
        "prunedCount": pruned,
        "compactedCount": compacted,
        "archivedCount": archived,
        "memoryReductionPercent": vault.compute_memory_reduction(),
        "activeScarsRemain": len(vault.activeScars_list)
    }
    vault.audit_log.append(report)
```

---

8. Specialized Vault Classes

8.1 Fossil Vault

```
class FossilVault(Vault):
    def fossil_cycle(self, current_cycle):
        sorted_scars = sorted(
            self.activeScars_list,
            key=lambda s: s.age, reverse=True
        )
        for s in sorted_scars:
            if s.fossilized and current_cycle % 10 == 0:
                echo = Echo(current_cycle, s.expression)
                self.insert(echo)
```

- Emits one echo per fossilized Scar every 10 cycles.

8.2 Contradiction Vault

```
class ContradictionVault(Vault):
    def contradiction_handler(self):
        for s in self.activeScars_list:
            if s.contradiction_score > 80:
```

```
            mutated = s.mutate()
            self.insert(mutated)
            self.entropySum += semantic_entropy(mutated.expression)
```

- Mutates Scars with `contradiction_score > 80` and adds to `entropySum`.

8.3 Reactor Vault

```
class ReactorVault(Vault):
    def reactor_cycle(self):
        scars = self.activeScars_list
        for i, s1 in enumerate(scars):
            for s2 in scars[i+1:]:
                if semantic_overlap(s1.expression, s2.expression) > 0.7:
                    combined = combine_features(s1.expression, s2.expression)
                    new_scar = Scar(
                        scarID=f"RCV_{s1.scarID}_{s2.scarID}",
                        geoids=list(set(s1.geoids + s2.geoids)),
                        timestamp=current_cycle_time(),
                        expression=combined,
                        cls_angle=recompute_cls(combined),
                        semantic_polarity=(s1.semantic_polarity + s2.semantic_polarity)/2
                    )
                    self.insert(new_scar)
```

- Recombines overlapping Scars (overlap > 0.7) into a new Scar.

8.4 Compression Vault

```
class CompressionVault(Vault):
    def compress_cycle(self):
        if self.entropySum > 5.0:
            victim = random.choice(self.activeScars_list)
            compress_expression(victim.expression)
            self.entropySum *= 0.5
```

- When `entropySum > 5.0`, compress a random Scar's expression and halve `entropySum`.

---

9. Integration Points

- **SPDE (Semantic Pressure Diffusion Engine):**
    - Consumes vault's `entropySum` to adjust diffusion maps.
    - Produces sketch Geoids under pressure peaks; these enter vaults as provisional Scars.
- **MSCE (Memory/Scar Compression Engine):**
    - Coordinates with vault pruning and compaction.
    - Merges residual Scars from eliminated Geoids.
- **ZPA (Zetetic Prompt API):**
    - Receives high-volatility Scars for potential user queries.
    - Flags ethical-review Scars when `delta_entropy > 1.0`.
- **SSL (Semantic Suspension Layer):**
    - Quarantines Scars with `IDI > 0.80`.
    - Logs suspension events to vault audit.

---

10. Summary of Parameters & Thresholds

Parameter	Value	Description
`MF_threshold_high`	0.75	Mutation frequency threshold for routing.
`Semantic_polarity_threshold`	0.5	Absolute polarity cutoff for vault assignment.
`Entropy_balance_threshold`	0.26	Δ entropy threshold for load balancing.
`MFG_threshold`	0.5	Memory Friction Gradient threshold to delay insertion.
`CLS_angle_proximity_threshold`	15°	Angle difference to trigger priority interrupt.
`Echo_friction_threshold`	0.68	Friction score below which echoes are quarantined.
`Scar_delay_cycles`	2	Max cycles a Scar may be delayed before action.
`VSI_fracture_threshold`	0.8	Vault Stress Index threshold to trigger fracture.
`Fallback_throttle_rate`	50 scars/cycle	Max scars processed from fallback after fracture.
`EntropySlope_opt_threshold`	0.05	Entropy increase threshold over 500 cycles.
`Drift_depth_threshold`	12	Max drift lineage depth before pruning.
`Scar_density_threshold`	25 per 100 cycles	New Scar rate to trigger optimization.
`Thread_overlap_threshold`	0.85	Percent overlap to trigger optimization.
`Memory_usage_threshold`	0.90	Fraction of storage capacity to trigger optimization.
`IRS_cutoff`	0.12	Minimum Influence-Based Retention Score.
`Low_entropy_cluster_cutoff`	0.43	Max entropy for composite compaction.

`Excess_buffer_size_for_purge`	3	Incoming buffer size above which to purge.
`Fracture_isolation_cycles`	3	Cycles to isolate vault after fracture.
`Divergence_IDI_threshold`	0.72	IDI value above which to quarantine a Scar.

End of Vault Engineering Specification v1.0.