Transaction Pressure

Meta / Commitment Context

Hackathon: Alchemy Hackathon 2026

Pitch Day: Friday, Jan 23 @ 8:00am PT

Hack Window: Jan 26–29

Demo Day: Jan 30 (morning)

Solo build constraints:

- 2–3 evenings of implementation
- Deterministic, demo-safe
- No production reliability expectations

Product One-Liner

Transaction Pressure is a windowed desktop-style game where failures are visible but their causes are hidden. Players inspect and approve “normal” blockchain transactions while unknowingly accumulating system pressure until a rate-limit incident occurs.

> Instead of hiding failures, we hide causes. You see the explosions — the game is figuring out why they happened.

Core Design Principles (Non-Negotiable)

1. **Failures are visible immediately** (incident overlay)
2. **Causes are latent and cumulative** (hidden meters)
3. **No tile is inherently bad** — only accumulation causes failure
4. **Inference over avoidance** — learning patterns, not positions
5. **Minesweeper-adjacent, not Minesweeper-literal**

Platform & UI Constraints

- Desktop web app
- Fixed-size, windowed UI (Windows XP–era feel)
  - Example: 960×720
  - Does not fill viewport
- No scrolling
- Grid + side panel always visible
- Incident overlay blocks input

Game Overview

Grid

- Size: **12 × 12** (144 tiles)
- Each tile represents a *normal* transaction/request

Player Actions

- **Inspect(tile)**
  - Reveals local signals (symptoms)
  - Does *not* advance time
  - Does *not* add pressure
- **Approve(tile)**
  - Commits the action
  - Advances turn counter
  - Adds hidden pressure
  - May trigger an incident

Win / Lose Conditions

Lose

- A **Rate Limit Incident** is triggered

Soft Win (Optional)

- Player approves N_APPROVAL_TARGET tiles without incident
  - Recommended: 25

System Model

Hidden Pressure Meters (Authoritative)

Meter A — Request Volume

- Represents total request pressure
- Increases on every approval

Meter B — Hotspot Concentration

- Represents correlated usage (same method/contract)
- Increases with clustering

```
Meters {
  volume: number
  hotspot: number
}
```

Incident Model

Incident Type

- **Rate Limit Incident** (single incident type in MVP)

Trigger Rule

Incident triggers immediately after an approval if:

```
volume >= VOLUME_THRESHOLD
OR
hotspot >= HOTSPOT_THRESHOLD
```

Only one incident fires per game.

Tile Data Model

Each tile has **two layers**: real metadata (optional) and synthetic simulation data (authoritative).

```
Tile {
  // Grid identity
  id: string
  row: number
  col: number

  // Real (Flavor Mode only)
  txHash?: string
  from?: string
  to?: string
  methodId?: string
  blockNumber?: number

  // Synthetic (authoritative)
  methodGroup: MethodGroup
  contractGroup: ContractGroup
  volumeWeight: number   // 1–4
  hotspotWeight: number  // 0–3

  // Player-visible state
  state: 'hidden' | 'inspected' | 'approved'
  revealedSignal?: Signal
}
```

Enumerations

```
MethodGroup =
  | 'eth_call'
  | 'getLogs'
  | 'traceCall'
  | 'sendRawTransaction'
  | 'getBlock'

ContractGroup = 'A' | 'B' | 'C' | 'D' | 'E'
```

Signals (Symptoms, Not Explanations)

Signals are revealed only through **Inspect** and never disclose thresholds.

```
Signal {
  symptom: 'Retries'
  severity: 0 | 1 | 2 | 3
  logSnippet: string
}
```

Severity is *local*. Logs are *suggestive*.

Global Status (Derived, Non-Authoritative)

Always visible summary derived from meters.

```
GlobalStatus = 'Stable' | 'Warm' | 'Hot' | 'Critical'
```

Derived from:

```
max(volume / VOLUME_THRESHOLD,
    hotspot / HOTSPOT_THRESHOLD)
```

Implementation Details

1. Tile Dataset Generation

Goals

- All tiles valid
- No explicit failure tiles
- Pressure emerges only via accumulation

Generation Algorithm

```
function generateTiles(seed): Tile[] {
  rng = seededRandom(seed)
  tiles = []

  for row in 0..11:
    for col in 0..11:
      tiles.push({
        id: uuid(),
        row,
        col,
        methodGroup: weightedRandom({
          eth_call: 0.25,
          getLogs: 0.20,
          traceCall: 0.15,
          sendRawTransaction: 0.25,
          getBlock: 0.15
        }),
        contractGroup: randomPick(['A','B','C','D','E']),
        volumeWeight: randomInt(1, 4),
        hotspotWeight: randomInt(0, 3),
        state: 'hidden'
      })
```

```
  return tiles
}
```

2. Local Severity Computation (Inspect)

Local severity answers:

> If I keep approving things like this, how bad could it get?

Algorithm

```
function computeLocalSeverity(tile, approvedTiles): number {
  related = approvedTiles.filter(t ⇒
    t.methodGroup === tile.methodGroup ||
    t.contractGroup === tile.contractGroup
  )

  base = tile.hotspotWeight
  accumulated = related.length * 0.5
  rawScore = base + accumulated

  if rawScore < 1.5 return 0
  if rawScore < 3.0 return 1
  if rawScore < 4.5 return 2
  return 3
}
```

Log Snippets

```
function getLogSnippet(severity): string {
  switch severity:
    case 0: return 'ok'
    case 1: return 'retry-after: 1s'
    case 2: return '429: Too Many Requests'
```

```
    case 3: return 'rate limit exceeded (burst)'
}
```

---

3. Approving a Tile

Flow

```
function approveTile(tile): void {
  tile.state = 'approved'
  turn += 1

  updateMeters(tile)
  checkForIncident()
}
```

Meter Updates

```
function updateMeters(tile): void {
  meters.volume += tile.volumeWeight

  related = approvedTiles.filter(t ⇒
    t.methodGroup === tile.methodGroup ||
    t.contractGroup === tile.contractGroup
  )

  meters.hotspot += tile.hotspotWeight
  meters.hotspot += related.length * 0.3
}
```

Threshold Constants

```
VOLUME_THRESHOLD = 120
HOTSPOT_THRESHOLD = 45
```

4. Incident Detection

```
function checkForIncident(): void {
  if (meters.volume >= VOLUME_THRESHOLD)
    triggerIncident('volume')
  else if (meters.hotspot >= HOTSPOT_THRESHOLD)
    triggerIncident('hotspot')
}
```

5. Postmortem: Contributor Attribution

Scoring

```
function scoreContribution(tile): number {
  score = tile.volumeWeight

  if (incidentType === 'hotspot') {
    score += tile.hotspotWeight
    score += countRelated(tile) * 0.5
  }

  return score
}
```

Selection

```
function pickContributors(approvedTiles): Tile[] {
  return approvedTiles
    .map(t ⇒ ({ t, score: scoreContribution(t) }))
    .sort(descending score)
    .slice(0, randomInt(8, 12))
```

```
    .map(x ⇒ x.t)
}
```

Postmortem Output

- Incident type
- Meter exceeded
- Highlighted tiles
- One-line hint

Real Alchemy RPC Integration — Flavor Mode

Principle

Real RPC data provides identity and texture, never causality.

Gameplay must be fully functional without any RPC access.

RPC Budget (Hard Limit)

Maximum **3 RPC calls per game**:

1. eth_blockNumber
2. eth_getBlockByNumber (hashes only)
3. Optional: up to 10 eth_getTransactionByHash

If *any* call fails → synthetic fallback.

Real vs Simulated Boundary

Real (Optional)

- Chain name

- Block number
- Transaction hashes
- from, to, methodId

Always Simulated

- Pressure meters
- Thresholds
- Incidents
- Signal severity

RPC Fetch Phase (Pre-Game Only)

```
async function fetchRealTransactions(): Promise<RealTx[]> {
  try {
    blockNumber = await alchemy.eth_blockNumber()
    block = await alchemy.eth_getBlockByNumber(blockNumber, false)
    hashes = block.transactions.slice(0, 10)

    return await Promise.all(
      hashes.map(h ⇒ alchemy.eth_getTransactionByHash(h))
    )
  } catch {
    return []
  }
}
```

Mapping Real Data → Tiles

Real transactions **seed** tiles but never dominate them.

```
function mapRealTxsToTiles(realTxs, tiles) {
  if (realTxs.length === 0) return tiles
```

```
  for tile in tiles:
    tx = randomPick(realTxs)
    tile.txHash = tx.hash
    tile.from = tx.from
    tile.to = tx.to
    tile.methodId = tx.input?.slice(0, 10)

    tile.methodGroup = mapMethodIdToGroup(tile.methodId)
    tile.contractGroup = mapAddressToGroup(tile.to)
}
```

Mapping Helpers (Lossy by Design)

```
function mapMethodIdToGroup(methodId) {
  if (!methodId) return randomGroup()
  if (methodId.startsWith('0xa9059cbb')) return 'eth_call'
  return 'traceCall'
}
```

```
function mapAddressToGroup(address) {
  return ['A','B','C','D','E'][hash(address) % 5]
}
```

Guardrails

1. **Offline-first** — game must start with zero RPC
2. **No RPC during gameplay**
3. **No real errors surfaced as incidents**
4. **Incident logic is always synthetic**

Cursor Build Order (Strict)

1. Data models + constants
2. Synthetic tile generation
3. Grid rendering
4. Inspect action + signal computation
5. Approve action + meter updates
6. Incident detection
7. Postmortem attribution
8. RPC flavor mode + fallback
9. Styling / theme

Explicit Non-Goals

- Production realism
- Fairness guarantees
- Real rate limiting
- Live provider incidents
- Levels or progression

Final Sanity Check

This design guarantees:

- Failures are visible
- Causes are hidden
- Innocent actions trigger incidents
- Inference beats avoidance
- Real Alchemy data enhances credibility without breaking mechanics