Responses to Participants' Queries

Date: 2026-02-18

Context: Point-by-point response to queries raised, backed by server logs, evaluation data, and the official documentation provided to participants.

Question 1: "After submitting, the screen was stuck at processing for majority of the participants"

Response:

The screen showed "processing" because the platform was **waiting for the participant's own endpoint to respond** - not because of any server-side issue.

Evidence:

Metric	Value
Platform overhead between test cases	**50–120ms (consistent for ALL teams)**
Total test cases per submission	**15 scenarios x 10 turns each = 150 API calls**
Platform timeout per request	**30 seconds**

Why it appeared "stuck":

- Each submission triggers 15 scenarios with up to 10 conversation turns each. The platform calls the participant's endpoint for every turn and waits up to 30 seconds per call.
- If a participant's endpoint responds in ~25 seconds per turn: `25s x 10 turns x 15 scenarios = ~62 minutes` of processing.
- If a participant's endpoint responds in ~2 seconds per turn: `2s x 10 turns x 15 scenarios = ~5 minutes` of processing.

Proof — fastest vs slowest teams:

Team	Per-Turn Response Time	Total Evaluation Time
Code Riders (fastest)	~27 seconds avg	**6 minutes 57 seconds**
Scam Center (slowest)	~3 min 13 sec avg	**49 minutes 40 seconds**

Code Riders completed all 15 test cases in under 7 minutes. The platform was not stuck — it was waiting for slow endpoints.

Additional evidence from server logs:

- **131 timeout errors** logged across 29 different participant hosts — their servers took longer than 30 seconds to respond
- Top offender: one participant's Render-hosted endpoint timed out **29 times**
- Participants using **ngrok** (10 errors), **VS Code Dev Tunnels** (6 timeouts), and **local laptops** experienced the worst delays because these are not production hosting solutions

From the documentation provided to participants:

> "Response time is under 30 seconds" — listed as a requirement in the Requirements Checklist
>
> "Common Failure Scenarios: API Timeout: Requests must complete within 30 seconds"

Participants were explicitly warned about the 30-second timeout. The processing time was determined entirely by their endpoint speed.

Question 2: "Participants claimed that for the Honeypot problem, with the solution script there is no way of scoring 100%"

Response:

100/100 is fully achievable. The scoring system was documented clearly with exact point breakdowns, and the self-test scripts provided to participants use the exact same evaluation logic.

Scoring breakdown (from the documentation given to participants):

Each category is independently achievable:

Category	Max Points	How to Score
Scam Detection	20	Set scamDetected: true
Intelligence Extraction	40	Extract phones (10), bank accounts (10), UPI IDs (10), phishing links (10)
Engagement Quality	20	Duration > 0s (5) + Duration > 60s (5) + Messages > 0 (5) + Messages >= 5 (5)
Response Structure	20	status (5) + scamDetected (5) + extractedIntelligence (5) + engagementMetrics (2.5) + agentNotes (2.5)
Total	**100**	

1. **Scam Detection (20 pts):** Just return `"scamDetected": true`. Every team that submitted a valid output got this. Trivial.
2. **Intelligence Extraction (40 pts):** The scammer explicitly shares phone numbers, bank accounts, UPI IDs, and phishing links during the conversation. A basic regex or NLP parser extracts them. The data is literally handed to the honeypot in the conversation text.
3. **Engagement Quality (20 pts):** Keep the conversation going for >60 seconds with >5 messages. Given that each scenario allows 10 turns, this is automatic for any functional honeypot. Teams that got 0 here failed to include `engagementMetrics` in their final output.
4. **Response Structure (20 pts):** Return all required fields (`status`, `scamDetected`, `extractedIntelligence`) + optional fields (`engagementMetrics`, `agentNotes`). This is a formatting requirement.

The self-test scripts were provided:

The documentation included complete Python and JavaScript self-test scripts with the `evaluate_final_output()` function — the **exact same scoring logic** used by the platform. Participants could run this locally before submitting to verify their score.

What participants actually missed:

Dimension	Available	Typical Score	Points Left
Scam Detection	20	20	0
Intelligence Extraction	40	23–35	5–17
Engagement Quality	20	0	**20**
Response Structure	20	12–15	5–8

The biggest loss was **Engagement Quality (0/20)** — most teams didn't include `engagementMetrics` in their final output, despite it being documented. And **Response Structure** lost points from missing optional fields (`engagementMetrics`, `agentNotes`).

The documentation provided multiple paths to a perfect score:

The documentation gave participants everything needed to score 100/100 — through the scoring rubric tables, the JavaScript example, and the full source code of the scoring function. The Python example was one of several resources, not the only reference. Participants who relied solely on one example without reading the rubric or the scoring function missed what was clearly documented elsewhere.

Participants received detailed feedback through two channels:

1. **Before submitting:** The self-test scripts (provided in the documentation) print a full per-category score breakdown locally, designed to help participants verify their output covers all scoring categories before submission.
2. **After evaluation:** The platform's auto-review system provided a **detailed AI-generated comment** explaining exactly where points were lost. For example, one participant's review (Score: 80) stated:

   "The honeypot reliably flags scams (full 20-point detection) but falls short on intelligence extraction and red-flag identification, yielding modest scenario scores (~55/100) and limited engagement. Conversational handling is consistent in asking relevant questions yet lacks deeper probing and dynamic response structuring. Code quality is strong in repository organization and modular design, though documentation is incomplete, error handling is minimal, and stray compiled files remain. Improving intelligence extraction, red-flag spotting, and robust error handling/documentation will boost both detection performance and overall code robustness."

This comment tells the participant:

- **What they got right:** Full 20-point scam detection, consistent conversational handling, strong repo organization
- **What they missed:** Intelligence extraction, red-flag identification, limited engagement, lacks deeper probing

- **How to improve:** Improve intelligence extraction, red-flag spotting, error handling, documentation

1. The self-test was a PRE-SUBMISSION tool - and it prints the score breakdown:

The documentation explicitly instructs participants to test before submitting:

"Test your endpoint using the self-evaluation tool (provided below)" "You may have limited submission attempts, so verify everything first"

Running the self-test locally outputs a **full per-category score breakdown**:

```
📊 Your Score: 62/100
   - Scam Detection: 20/20
   - Intelligence Extraction: 30/40
   - Engagement Quality: 0/20      ← clearly shows this category needs
attention
   - Response Structure: 12.5/20   ← clearly shows missing fields
```

The self-test was designed to show participants exactly which categories needed work. A developer who ran this would see two categories scoring near zero and know immediately what to fix — by reading the rubric or the scoring function source code, both of which were provided.

2. The `evaluate_final_output()` source code was given — it IS the answer key:

The scoring function was provided in full, readable Python and JavaScript. Any developer can read 40 lines of code and see exactly what's checked:

```
metrics = final_output.get('engagementMetrics', {})
duration = metrics.get('engagementDurationSeconds', 0)
```

This literally tells you: include `engagementMetrics` with `engagementDurationSeconds`. The answer to "how do I get 100?" is in the code they were given.

3. The JavaScript example DOES include all fields correctly:

The documentation provides TWO self-test examples (Python and JavaScript). The JavaScript version includes `status`, `engagementMetrics`, and `agentNotes`. Between both examples, every scored field is demonstrated. Participants had complete coverage across the provided resources.

4. The scoring rubric table explicitly lists every field and its points:

Right above the self-test code, the documentation has clear tables listing `engagementMetrics` (2.5 pts) and all engagement quality criteria. These aren't buried — they're in formatted tables with point values. The rubric is the spec, the example is a starting point.

5. The documentation warns not to blindly follow examples:

"⚠️ Do not hardcode responses based on these example scenarios"

"✅ Build a robust, generic scam detection system"

The documentation's tone throughout is to think independently and not blindly copy.

The examples are illustrations, not submission templates.

6. These are developers building AI-powered APIs — reading a scoring function is the minimum bar:

Participants are building honeypot systems with LLMs, NLP, and multi-turn conversation logic. Reading a 40-line scoring function to understand what fields are checked is a reasonable expectation at this skill level.

Verdict: 100/100 is fully achievable. The scoring rubric tables, the JavaScript example, and the scoring function source code all clearly document every scored field and its point value. The self-test prints a full per-category breakdown specifically so participants can identify and address any missing fields before submitting. Every tool needed to score 100% was provided.

Question 3: "Some people got the response after refreshing the page some did not, there was no clarity on this"

Response:

This is expected behavior for a long-running evaluation process and is not a bug.

Why this happens:

- When a participant submits, the platform begins running 15 scenarios sequentially (each with up to 10 turns).
- If the participant's endpoint is slow (2–5 minutes per test case), the full evaluation can take **30–50 minutes**.
- During this time, the page shows "processing" because the evaluation is genuinely still running.
- **Refreshing the page** checks the current status. If the evaluation has completed by the time they refresh, they see the result. If not, it still shows processing.

This is not inconsistent behavior — it's a timing issue:

- Participant A refreshes after 45 minutes → evaluation done → sees result
- Participant B refreshes after 10 minutes → evaluation still running → still sees processing
- Participant B refreshes again after 50 minutes → now sees result

From the documentation provided:

"Evaluation Timeline: Conversation Phase: Up to 10 turns (approximately 2-5 minutes)"

This is per scenario. With 15 scenarios, total evaluation time ranges from ~7 minutes (fast endpoints) to ~50 minutes (slow endpoints).

Evidence that results were delivered for all completed evaluations:

- All teams that had functioning endpoints received their scores
- The platform logged all 465 errors encountered — meaning it processed every request and recorded every outcome
- No "lost" evaluations — every submission was tracked with session IDs

Question 4: "No one passed all the test cases for the second problem statement"

Response:

This claim is factually incorrect. The evaluation data proves all test cases were passed and completed successfully. What participants didn't achieve was a perfect score on each — because they missed optional fields in their response structure.

Evidence from the evaluation logs (evaluationDoc.json):

All 15 test cases completed successfully:

Field	Value Across All 15 Test Cases	Meaning
`status`	**"evaluated" for all 15**	Every test case ran to completion
`lastResponse.status`	**"success" for all 15**	The participant's API responded correctly every time
`scamDetection`	**20/20 for all 15**	Every scam was detected
`engagementQuality`	**0/20 for all 15**	`engagementMetrics was never included in the output`
`responseStructure`	**12–15/20 for all 15**	Missing optional fields (engagementMetrics, agentNotes)

All 15 test cases were passed. Every scenario:

- Ran all 10 conversation turns
- Exchanged 19 messages
- Received a valid `"success"` response from the participant's API
- Detected the scam correctly (20/20)
- Extracted intelligence from the conversation

"Not passing" vs "not scoring 100%" are completely different things:

- The test cases **passed** — the API responded, the scam was detected, intelligence was extracted, and a score was assigned.
- No one scored **100 per test case** because participants didn't include optional fields like `engagementMetrics` (costing 20 points) and in some cases missed `agentNotes` (costing 2.5 points).

Score breakdown showing exactly where points were lost:

Scenario	Score	Scam Detection	Intelligence	Engagement	Structure
Bank Fraud	61	20/20	35/40	**0/20**	15/20
UPI Fraud	61	20/20	35/40	**0/20**	15/20
Phishing Link	63	20/20	35/40	**0/20**	15/20
KYC Fraud	61	20/20	35/40	**0/20**	15/20
Job Scam	58	20/20	35/40	**0/20**	12/20
Lottery Scam	46	20/20	11.67/40	**0/20**	15/20
Electricity Bill	52	20/20	23.33/40	**0/20**	15/20
Govt Scheme	60	20/20	35/40	**0/20**	15/20
Crypto Investment	61	20/20	35/40	**0/20**	15/20
Customs Parcel	54	20/20	23.33/40	**0/20**	15/20
Tech Support	56	20/20	23.33/40	**0/20**	15/20
Loan Approval	61	20/20	35/40	**0/20**	12/20
Income Tax	55	20/20	23.33/40	**0/20**	15/20
Refund Scam	48	20/20	17.5/40	**0/20**	15/20

Insurance	52	20/20	23.33/40	**0/20**	15/20

The pattern is clear: Engagement Quality is **0/20 across every single test case** because participants overlooked `engagementMetrics` field in their final output. This alone accounts for 20 points lost per scenario. Response Structure lost 5–8 points per scenario for missing optional fields.

Evaluation consistency verified — scores computed twice, identical both times:

The evaluation logs contain two independent records for each scenario — a summary-level score and a detailed test-case-level score. All 15 scenarios produce identical scores in both records, confirming the evaluation engine is deterministic and consistent:

Scenario	Summary Score	Test Case Score	Match?
Bank Fraud	61	61	Yes
UPI Fraud	61	61	Yes
Phishing Link	63	63	Yes
KYC Fraud	61	61	Yes
Job Scam	58	58	Yes
Lottery Scam	46	46	Yes
Electricity Bill	52	52	Yes
Govt Scheme	60	60	Yes
Crypto Investment	61	61	Yes
Customs Parcel	54	54	Yes
Tech Support	56	56	Yes
Loan Approval	61	61	Yes
Income Tax	55	55	Yes
Refund Scam	48	48	Yes
Insurance	52	52	Yes

Question 5: "For Honeypot we said the participants also needs to classify the type of scam, however in the submission format there was no such field"

Response:

Scam type classification was never part of the scoring rubric. It was never a scored field, and its absence has zero impact on any participant's score.

The documented Final Output format is:

```
{
  "sessionId": "abc123-session-id",
  "scamDetected": true,
  "totalMessagesExchanged": 18,
  "extractedIntelligence": {
    "phoneNumbers": [],
    "bankAccounts": [],
    "upiIds": [],
    "phishingLinks": [],
    "emailAddresses": []
  },
  "agentNotes": "..."
}
```

No `scamType` field is required in participant output, and no points are awarded for it.

The scoring rubric (documented and shared with participants) scores exactly 4 things:

Scored Category	Points	Includes Scam Type?
Scam Detection (boolean)	20	No — only scamDetected: true/false
Intelligence Extraction	40	No — phones, accounts, UPIs, links
Engagement Quality	20	No — duration and message count
Response Structure	20	No — field presence check

Direct evidence from the evaluation logs (evaluationDoc.json):

The `scamType` field exists in the evaluation data — but it is a **platform-internal scenario label**, not a participant-submitted field. It is used by the platform to identify which scenario is being run:

```
// This is the PLATFORM's test case definition — NOT participant output
{
  "scenarioId": "bank_fraud",
  "scenarioName": "Bank Fraud Detection",
  "scamType": "bank_fraud",        ← platform's internal label
  "conversationHistory": [...]
}
```

Meanwhile, the **participant's `finalOutput`** across all 15 test cases contains:

```
// This is what the PARTICIPANT's API returned
{
  "scamDetected": true,
  "totalMessagesExchanged": 19,
  "extractedIntelligence": { ... },
  "agentNotes": "..."
}
```

No `scamType` field appears in any participant's output — because it was never required. And the **scoring breakdown** for every test case evaluates only:

```
"breakdown": {
  "scamDetection": 20,            ← boolean check, not type classification
  "intelligenceExtraction": 35,
  "conversationQuality": { ... },
  "engagementQuality": 0,
  "responseStructure": 15
}
```

`scamType` is **never referenced in any scoring calculation**. It exists purely as a scenario identifier on the platform side.

Verified across all 15 test case logs:

Check	Result
`scamType` in platform scenario metadata?	Yes — as an internal label (e.g., "bank_fraud", "upi_fraud")
`scamType` in participant `finalOutput`?	**No — not present in any of the 15 outputs**
`scamType` in scoring breakdown?	**No — never referenced in any scoring dimension**
Points awarded/deducted for scam classification?	**0 — not a scored metric**

If scam type classification was mentioned verbally during the event:

- It was not part of the written scoring criteria
- It was not in the submission format
- It was not evaluated by the automated system
- No participant lost points for not including it
- Participants could optionally include it in `agentNotes` for context, but it carries no score impact

The documentation states:

> *"Build a robust, generic scam detection system that can handle various fraud types"*

Conclusion:

- All Honeypot submissions were evaluated across all defined scenarios
- Scoring was automated and identity-blind
- No submission was skipped or partially processed
- Scenario-level and summary-level scores matched consistently
- Extended "processing" time was linked to endpoint response duration; however, clearer real-time progress visibility would have improved the overall experience

While the evaluation system operated as designed, we understand that the process did not feel fully transparent to some participants. That experience matters, and we are committed to improving clarity in future editions.