Gruesøme’s Arcade — Game Integration Guide (Economy + Metrics + Web3 Locks)

Version: v1.3

This guide is the “one document” you follow when building ANY new arcade game that must:

- run standalone for dev/demo
- run embedded in the Arcade (iframe)
- use parent-authoritative Credits + payouts
- support multi-metric leaderboards fairly (per-game, per-genre)

1) Golden Rules (Do Not Break These)

1. Parent-authoritative economy

- The Arcade (parent) decides when a run is allowed, the runId, and the cost.
- Games never deduct Credits locally.
- Games never “submit score directly to a leaderboard” when embedded.
- The game sends a RUN_RESULT; the Arcade verifies + records.

2. Run gating (Web3 lock compatibility)

- When embedded, gameplay cannot start until RUN_GRANTED.
- If not embedded, game starts normally (standalone).

3. Multi-metric support

- Every run can report multiple metrics (score, time, waves, etc).
- The Arcade chooses which metric is the “ranked” metric for each board.
- Metric direction (higher is better vs lower is better) must be declared.

4. Fairness guardrails for in-run spend (if your game spends Credits during play)

- Use a ranked spend cap (rankedSpendCapAC) so whales cannot buy rank.
- Report inRunSpendAC so efficiency-based leaderboards can exist.
- If spend can increase power, your skill metric MUST be paired with an efficiency metric.

2) Bridge Protocol (postMessage)

All embedded games must speak a minimal protocol over window.postMessage.

2.1 Channel key

Every message must include: channel: ""

Example: "MOONSHOT_ARCADE_BRIDGE_V1" or "GA_BRIDGE_V1" Use a constant per game. The Arcade will reject missing/wrong channel.

2.2 Message envelope

```
{
  channel: "GA_BRIDGE_V1",
  type: "ARCADE:READY",
  payload: { ... }
}
```

2.3 Required message types

A) Game -> Arcade

Arcade:Ready

- sent once on load (only when embedded)

payload: { gameId: "", version?: "1.0.0", metricsVersion?: "v3.1" }

Arcade:Request_Run

- sent when the player attempts to start

payload: { gameId: "", desiredRunType?: "ranked|casual|tournament" }

Arcade:Run_Result

- sent once at game over

payload: { gameId: "", runId: "", durationMs: 12345, metricId: "", // optional but recommended metricValue: 999, // optional but recommended metrics: { // multi-metric payload (recommended) score: 1234, durationMs: 45678, waves: 19, kills: 88, accuracyBp: 7425, inRunSpendAC: 12, efficiency: 102.5 }, spent: { paidAC: 1, promoAC: 0 }, // optional (if your game spends in-run) flags: { suspectedCheat?: false } // optional }

B) Arcade -> Game

Arcade:Sync

- sent any time balances/membership change

payload example: { address: "0x…", credits: { paid: 120, promo: 15 }, membership: { tier: "free|pro1|pro2|pro3", active: true, expiresAt: 0 }, avatar: { png: "https://…/avatar.png", nickname: "GRUES" } }

Arcade:Run_Granted

- authorizes gameplay start and assigns a runId

payload: { runId: "", runType: "ranked", cost: { paidAC: 1, promoAC: 0 } }

Arcade:Run_Denied

payload: { reason: "not_connected|no_funds|poh_required|rate_limited", message: "..." }

2.4 Start-input queueing (required feel)

When embedded:

- On first start input (click/tap/space):

1) do NOT start gameplay yet

2) send ARCADE:REQUEST_RUN

3) queue the initiating input (ex: flap/thrust/fire)

- When RUN_GRANTED arrives:

1) start game immediately

2) apply queued input immediately (so controls feel identical)

2.5 Timeout UX (required)

If no RUN_GRANTED / RUN_DENIED within 2500ms:

- show center message:
- “Connect wallet to play” OR “Not enough credits — open Wallet”
- provide hint:
- “Open Wallet in the sidebar”

3) Metrics: Generic Schema + Best Practices

3.1 Metric object fields (catalog side)

Each game declares allowed metrics in arcade-games.json:

- id: metric id (string)
- label: short UI name
- direction: "desc" or "asc"
- unit: "points|ms|bp|count|ac|ratio"
- clamp: optional { min, max } to prevent outliers
- notes: optional short meaning / anti-cheat note

3.2 Common metric IDs (recommended)

- score (desc) — classic points
- durationMs (desc for survival, asc for speedruns)
- timeMs (asc) — speedrun time
- waves (desc) — wave/round reached (tower defense / survival)
- kills (desc)
- accuracyBp (desc) — accuracy in basis points (0..10000)
- inRunSpendAC (asc or desc depending on board) — Credits spent during run
- efficiency (desc) — “primary performance per spent AC” (example: waves / AC)

3.3 Normalization (how to keep it fair)

Different game types reward different skills. Instead of forcing one global score:

- Each game has a defaultMetric for skill payouts
- Games can expose multiple metrics so players can compete on different axes

3.4 Anti-cheat boundary

- The game is untrusted.
- The Arcade server should verify:
- durationMs is plausible
- metrics are within clamps
- spend does not exceed rankedSpendCapAC
- runId exists and was granted

4) In-Run Spend Games (Economy-dynamic games)

If your game allows spending Credits inside a run (upgrades, ammo, retries):

- set usesCreditsInRun=true in arcade-games.json
- set rankedSpendCapAC (recommended)
- always report inRunSpendAC
- include an efficiency metric so “spend-heavy” doesn’t dominate

Example efficiency formulas:

- efficiency = waves / max(1, inRunSpendAC)
- efficiency = score / max(1, inRunSpendAC)
- efficiency = kills / max(1, inRunSpendAC)

5) Quick Implementation Checklist (copy/paste)

[ ] Detect embedded mode: window.parent !== window [ ] Send ARCADE:READY on load (embedded only) [ ] On first start input:

- if embedded: ARCADE:REQUEST_RUN + start timeout + queue input

- else: start normally

[ ] On RUN_GRANTED: start + apply queued input [ ] On RUN_DENIED: show message, do not start [ ] Track runStartMs with performance.now() [ ] On game over: send ARCADE:RUN_RESULT with runId + durationMs + metrics

6) Debugging

- Add a dev flag: ?bridgeDebug=1
- Log inbound/outbound postMessage events in debug mode.
- Provide a local “parent harness” HTML to simulate the Arcade during dev.

END