Boing Network — Quality Assurance at the Protocol Layer

> ***Mission addition:*** *True quality assurance with top-notch standards.*
> *The QA regulatory system* ***confirms that all assets are up-to-par before approving deployment*** *onto the blockchain. No asset is deployed until it passes. All regulatory QA processes are* ***automated and decentralized****—with a decentralization that* ***defends against the deployment of malicious assets*** *(no scams, no single gatekeeper).*

Quick reference — Primary factors: Reject vs Allow vs Unsure

Outcome	When it happens	Primary deterring / enabling factors
Reject	Deployment is **blocked** and never enters a block.	**Bytecode:** empty, over size limit (e.g. 32 KiB), invalid opcode, malformed (truncated PUSH, trailing bytes). **Purpose:** invalid category when provided (must be one of: dApp, token, NFT, meme, community, entertainment, tooling, other). **Legitimacy:** bytecode hash on blocklist (known scam/malware), or bytecode contains a known scam/malware **pattern** (governance-defined).
Allow	Deployment is **accepted** and can be included in a block.	**All hard rules pass:** bytecode within size limit, only valid Boing VM opcodes, well-formed instruction stream, not on blocklist, no scam pattern match. **Purpose** (if provided) is a valid category; **missing purpose is allowed** per spec. **No** policy-required pool review for this category; **no** soft rule triggering Unsure (e.g. "other" with minimal description). Meme, community, entertainment are valid and not discriminated against.
Unsure	Deployment is **referred to the community QA pool**; not auto-accepted.	**Soft rules:** e.g. purpose category `"other"` with very short or empty description (description length < 4 bytes). **Policy:** category is in the governance-defined "always review" list (e.g. high-stakes financial). When automation lacks sufficient evidence to Allow or Reject, the pool decides.

Summary: The main **reject** triggers are: **specs failure** (size, opcodes, well-formedness), **blocklist match**, **scam pattern match**, and **invalid purpose category**. The main **allow** condition is: **all hard rules pass** and **purpose is valid or omitted**. **Unsure** is used for **edge cases** (ambiguous declaration, policy-required review) so the community pool can decide without a single gatekeeper. For how to **update** QA rules (blocklist, content blocklist, scam patterns, etc.) via governance, see Appendix C: Governance-mutable QA rules.

Public transparency — live registry and canonical reference

- **Live rule registry (read-only):** JSON-RPC `boing_getQaRegistry` with params `[]` returns the **effective** `RuleRegistry` JSON the node is using (same shape as `qa_registry.json`). No authentication required.
- **Canonical reference JSON (for comparison and docs):** `docs/config/CANONICAL-QA-REGISTRY.md` describes `qa_registry.canonical.json` and `qa_pool_config.canonical.json`, aligned with code defaults. **Deployed networks may differ** after governance or operator policy updates—always verify against `boing_getQaRegistry` and `boing_qaPoolConfig` on the RPC you trust.

- **Explorer:** Boing Observer — QA transparency loads live pool parameters, pending queue, and registry JSON from public RPC.

Table of Contents

---

1. Vision

- **Sixth pillar (optimized):** True quality assurance with top-notch standards. All **currently known** edge cases are resolvable by automation (Allow or Reject); **leniency for meme culture**, **no maliciousness or malignancy** allowed; community pool only for genuinely unknown or policy-mandated cases.
- **Nothing deploys without passing QA.** The regulatory system confirms that every asset is up-to-par **before** approving deployment onto the blockchain. There is no "deploy first, fix later."
- **Two things must be confirmed:** the **specs** of the asset (technical standards) and the **purpose** (legitimate reason for deploying; **no scams whatsoever**).
- **All regulatory QA is automated and decentralized.** Automation runs the checks; when the system is unsure, a **community QA pool** decides—throughout the **lifespan** of the network. Decentralization is designed to **defend against malicious deployments**, not to enable them.
- **Transparent, auditable rules** so developers and the community know exactly what is required.

---

2. Two Pillars: Specs and Purpose

QA must confirm **both** of the following before any deployment is approved:

Pillar	What it means	How it's enforced

Specs of the asset	Technical compliance: bytecode size, allowed opcodes, well-formedness, interfaces, metadata, and any asset-type-specific standards (e.g. token supply, decimals, NFT metadata schema).	Automated checks: deterministic rules run at submit and execution. Fail → reject. Unclear → pool.
Purpose / legitimacy	There must be a **reason** for deploying that meets network standards. **No scams whatsoever.** The deployment must declare or demonstrate a legitimate use (e.g. dApp, token, NFT collection, tooling)—and must not match known scam patterns (fake tokens, phishing, rug-pull patterns, etc.).	Automated checks where possible: required purpose declaration or category, blocklist of known scam patterns, heuristics for obvious abuse. When automation cannot determine legitimacy (e.g. novel or borderline case), the asset **falls into the QA community pool** for a decentralized allow/reject decision.

Operationalizing "purpose" in an automated, decentralized way:

- **Declaration:** Deployments can be required to include a **purpose category** and optional **description** (or hash thereof) as part of the deploy payload or metadata. Missing or invalid declaration → reject or unsure.
- **Blocklists and patterns:** The protocol can maintain (via governance) a **blocklist** of known scam patterns, bytecode hashes, or behavioral signatures. Matches → reject.
- **Legitimacy heuristics:** Automated checks for common scam indicators (e.g. misleading naming, copy of known scam contract) can yield **reject** or **unsure**; unsure goes to the pool.
- **No single gatekeeper:** Legitimacy for edge cases is decided by the **community QA pool**, not a central authority—so decentralization defends against both malicious deployers and malicious gatekeeping.

---

3. Why This Matters

On many account-based L1s with contract layers:

- Fungible and NFT-style interfaces are usually **conventions**, not enforced by the protocol.
- Malformed, unsafe, or non-compliant contracts can be deployed and only fail or cause harm when called.
- There is no protocol-level gate that says “this bytecode does not meet our quality bar.”

Boing Network differentiates by:

- **Enforcing** at deployment time: if a deployment does not pass QA (automated + pool when needed), it is **rejected** and never enters a block.
- **Automation first**: fast, deterministic checks for known rules; human/social layer only for genuine edge cases.
- **Clear criteria**: a single source of truth for “what is allowed,” reducing ambiguity and improving security.

---

4. Asset Types and Scope

Asset type	Current Boing model	QA scope
Smart contracts	`TransactionPayload::ContractDeploy { bytecode }` — bytecode stored in state, executed by Boing VM	Bytecode rules (size, opcodes, well-formedness, security heuristics).

Native / protocol tokens	BOING is native; no separate “token deploy” today	If we add first-class “token mint” or “token template” deployments, QA applies to those.
NFTs / collectibles	Today: contracts holding state (e.g. “NFT” as contract + storage)	Same as smart contracts; optional **metadata/profile** checks if we define an on-chain NFT profile.
Future asset kinds	Extensible via new payload types or contract conventions	Each new kind gets a corresponding rule set and, if needed, pool policy.

For the **current** codebase, the only deployment path is **ContractDeploy**. So the first phase of QA is **contract bytecode QA**; the same framework can later cover tokens, NFTs, and other assets as they are added.

---

5. Rules and Standards (What Assets Must Meet)

These are the **attributes and rules** that deployments must satisfy. They are the input to both automation and (when unclear) the community pool.

5.1 Smart contracts (bytecode)

- **Size limits**
  - Maximum bytecode size (e.g. 24 KiB or 32 KiB). Prevents abuse and keeps state bounded.
- **Opcodes**
  - Only opcodes defined in the Boing VM are allowed. Any byte that does not decode to a valid opcode (or valid PUSH immediate) → **reject**.
  - Today: `Stop, Add, Sub, Mul, Div, Mod, AddMod, MulMod, Lt, Gt, Eq, IsZero, And, Or, Xor, Not, Shl, Shr, Sar, Address, Caller, Dup1, Log0..Log4, MLoad, MStore, SLoad, SStore, Push1..Push32, Jump, JumpI, Return` (see `boing-execution/src/bytecode.rs`).
- **Well-formedness**
  - Valid instruction stream: PUSH immediates consumed (correct lengths), no jump targets to non-instruction boundaries, no truncated instructions at end of bytecode.
- **Init-code prefix (`0xFD`)**
  - Optional leading byte `CONTRACT_DEPLOY_INIT_CODE_MARKER` (`0xFD`): opcode / well-formedness checks apply to the bytes **after** the prefix; the marker itself is not executed as VM code. A payload that is only `0xFD` (no trailing init code) → **reject** (malformed). See `TECHNICAL-SPECIFICATION.md` §4.4.
- **Security and policy**
  - No use of opcodes or patterns that are **explicitly disallowed** by network policy (e.g. certain dangerous patterns if we define them).
  - Optional: static analysis heuristics (e.g. “no jump into PUSH data”) to catch obvious exploits; failures can be **reject** or **unsure** (send to pool).
- **Metadata / profile (optional)**
  - If we add a convention for “contract profile” (e.g. type: token / NFT / generic), deployment can require a valid profile; missing or invalid profile → reject or unsure.

5.2 Tokens and NFTs (when applicable)

- **Fungible reference calldata** (off-chain convention): see `BOING-REFERENCE-TOKEN.md`. Deployments declaring `token` should align wallet/tooling with that layout where practical; bytecode QA remains

opcode/size/well-formedness + purpose rules.

- **NFT reference calldata** (off-chain convention): see `BOING-REFERENCE-NFT.md` — `owner_of`, `transfer_nft`, optional `set_metadata_hash` selectors and 96-byte word layout. Deployments declaring `NFT` / `nft` should use that ABI for interoperability; optional on-chain metadata commitments are contract-defined.
- **Naming / metadata**: optional checks (e.g. content policy on `asset_name` / `asset_symbol` in `ContractDeployWithPurposeAndMetadata`) apply when those fields are present; NFT URI schemas remain off-chain unless governance adds rules.

5.3 Purpose and legitimacy (no scams)

- **Required declaration:** Purpose category and/or description (or hash) as part of deploy payload or metadata; missing or invalid → reject or unsure.
- **Valid purpose categories** include dApp, token, NFT, **meme**, **community**, **entertainment**, tooling, and other governance-defined categories. Meme and community/entertainment are explicit valid use cases; lack of a "solid" or traditional use-case is not grounds for reject or Unsure (see §10).
- **Blocklist:** Known scam patterns, bytecode hashes, or behavioral signatures (maintained via governance). Match → reject.
- **Legitimacy heuristics:** Automated checks for common scam indicators (e.g. misleading naming, copy of known scam). Reject or unsure; unsure → pool.
- **No scams whatsoever:** The network enforces that deployments have a legitimate reason; edge cases go to the community QA pool so there is no single gatekeeper.

5.4 Central registry of rules

- A single **on-chain or protocol-defined** registry (e.g. in `boing-governance` or a new `boing-qa` crate) that lists:
  - Rule IDs, descriptions, and whether each rule is **hard** (automated reject) or **soft** (can trigger "unsure" and pool).
- Governance can add or adjust rules over time (e.g. via existing time-locked governance), so the bar can evolve without a chain restart.

---

6. Automated QA Pipeline

- **When:** Run QA on every **ContractDeploy** (and in the future, on every deploy of other asset types).
- **Where (recommended):**
  1. **At mempool insert** (`boing-node`: after signature verification, before `mempool.insert`): run bytecode QA. Reject immediately with a clear error code so the client can show "Deployment rejected: reason X."
  2. **At execution (defense in depth):** In `execute_contract_deploy` (or equivalent), run the same (or a minimal) check again. If a bug or bypass allowed bad bytecode into a block, execution still refuses to apply it.
- **Outcomes:**
  - **Allow** → transaction is accepted (mempool and later execution).
  - **Reject** → transaction is not accepted; return a structured error (e.g. `QaRejected { rule_id, reason }`).
  - **Unsure** → transaction is **not** auto-accepted; it enters a **pending QA** flow and is referred to the **community QA pool** (see below). Optionally, we can time-limit: if no decision by time T, treat as reject (or allow, by policy).

6.1 Determinism and consensus

- All **automated** checks must be **deterministic** and **identical** across nodes. So:
  - Same bytecode → same Allow/Reject/Unsure result on every node.
  - No randomness, no external calls, no “node-specific” data in the decision.
- If we introduce optional **heuristics** that might differ (e.g. heavy static analysis), they should either be:
  - Standardized and specified so all nodes run the same logic, or
  - Used only off-chain (e.g. in wallets or dev tools), with on-chain QA remaining deterministic.

---

7. What Determines Allow, Reject, or Unsure (Edge Case)?

The automated system returns exactly one of **Allow**, **Reject**, or **Unsure**. What **truly determines** which outcome is chosen is defined below. The system only returns Allow or Reject when it has **sufficient, deterministic evidence**; when it does not have enough knowledge to firmly approve or reject, it returns **Unsure** and the asset becomes an **edge case** for the community QA pool.

7.1 Core principle: sufficient evidence

- **Allow** and **Reject** are returned only when the protocol has **enough knowledge** to be certain. If the evidence is missing, ambiguous, or defined as requiring human judgment, the outcome is **Unsure**.
- **Unsure (edge case)** means: the regulatory system **explicitly does not have enough knowledge** to firmly approve or firmly reject. The asset is neither approved nor rejected by automation; it is deferred to the community pool.

7.2 When the system returns **Reject**

The system **firmly rejects** when any of the following is true (deterministic, no discretion):

Condition	Meaning
A hard rule fails	Any rule in the central registry marked **hard** fails (e.g. bytecode over size limit, invalid opcode, malformed instruction stream, mandatory purpose declaration missing). Hard rules are binary and fully specified; failure → Reject.
Blocklist match	The deployment (e.g. bytecode hash, pattern, or behavioral signature) matches an entry on the protocol blocklist. No ambiguity; match → Reject.
Legitimacy heuristic: reject zone	A legitimacy check (e.g. similarity to known scam, forbidden naming pattern) returns a result in the **reject** zone as defined by the rule (e.g. similarity score above threshold). The rule explicitly says "this confidence → Reject."

7.3 When the system returns **Allow**

The system **firmly approves** only when **all** of the following hold:

Condition	Meaning
All hard rules pass	Every hard rule (specs + any hard purpose rules) passes.
No soft rule in "fail" state that is defined as Unsure	Soft rules that fail (see below) force Unsure, not Allow. So for Allow, either there are no soft rules or every soft rule passes.
Not on blocklist	No match to the blocklist.

Purpose declaration valid (if required)	Category and/or description meet the required format; not missing, not invalid.
Legitimacy heuristics not in reject or unsure zone	Any legitimacy heuristic returns a result in the **allow** zone (e.g. below "suspicious" threshold, no similarity to known scam).
No policy-required pool review	The asset category or type is not one that governance has marked as "always send to pool" (see below).

If any of these conditions is not met, the system does **not** return Allow (it returns Reject or Unsure instead).

7.4 When the system returns **Unsure** (edge case)

The asset **falls into the category of edge case** and is sent to the community QA pool when the automated system **does not have enough knowledge** to firmly approve or reject. Concretely, **Unsure** is returned when any of the following is true:

Trigger	What it means — "not enough knowledge"
A soft rule fails	Rules in the registry can be marked **soft**. When a soft rule fails, the outcome is **Unsure** by definition: the protocol does not treat the failure as definitive reject (e.g. a heuristic that may have false positives). So the system explicitly does not claim "reject"—it defers.
Policy-required pool review	Governance can designate certain **asset categories** or **types** (e.g. "financial instrument", "token with mint capability") as "always review by pool." For these, automation **never** firmly approves; it always returns Unsure so the pool decides. The system does not claim to have enough knowledge for those categories.
Legitimacy in gray zone	A legitimacy heuristic returns a result in the **unsure** zone (e.g. "suspicious but not blocklisted", "similarity to known scam in [low, high] range"). The rule is defined so that in this band the system does **not** approve or reject—it has no firm conclusion.
Ambiguous or minimal declaration	Purpose declaration is present but **ambiguous** (e.g. category = "other", or description empty/minimal). Automation cannot verify purpose from this; it does not have enough information → Unsure.
New / unknown pattern	The asset (bytecode, metadata, or pattern) does **not** match any known-good or known-bad pattern in the system's data. It is the first of its kind. By definition the system has **no prior knowledge** to approve or reject → Unsure.
Conflicting signals	One check says "pass" (e.g. bytecode well-formed) and another says "suspicious" (e.g. similarity to scam below reject threshold but above allow threshold). The protocol does not automatically resolve the conflict → Unsure.

So **what truly determines** that an asset is an edge case is: **(1)** a **soft** rule failed, **(2)** policy says this category always gets pool review, **(3)** a heuristic is in a defined **gray zone**, **(4)** declaration is **ambiguous**, **(5)** the asset is **new/unknown** to the system, or **(6)** checks give **conflicting** results. In all cases, the design is: **when in doubt, defer to the pool**—the system does not guess.

7.5 Summary: one sentence per outcome

- **Reject:** The system has **deterministic evidence** that the asset violates a hard rule, is blocklisted, or is in a heuristic's reject zone.

- **Allow:** The system has **deterministic evidence** that all hard rules pass, blocklist does not match, purpose is valid, heuristics are in the allow zone, and no policy requires pool review.
- **Unsure (edge case):** The system **does not have enough knowledge** to be certain—either by rule design (soft rule, policy-required review), by gray zone (heuristic or ambiguous declaration), or by lack of data (new/unknown pattern, conflicting signals). The asset is neither approved nor rejected by automation; the community pool decides.

---

8. Community QA Pool (Edge Cases, Network Lifespan)

The QA community pool exists for **edge cases that occur throughout the lifespan of the network**. Whenever the regulatory QA system does not know whether it should allow or reject an asset, that asset **falls into this QA community pool** for a decentralized allow/reject decision. This applies from day one and for as long as the network runs—new patterns, borderline cases, and appeals will always be possible.

When the automated system returns **Unsure**, the deployment does not go through the normal path. Instead:

- **Pending QA queue:** The deployment (e.g. tx hash + bytecode hash + deployer) is recorded in a **pending QA** structure (on-chain or in a dedicated module).
- **Eligibility for the pool:** A set of **QA pool members** (addresses) who are allowed to vote. Eligibility can be:
  - Staked validators, or
  - Separate stake for "QA pool" role, or
  - Governance-selected set, or
  - Open to any staker above a threshold.
- **Voting:** For each pending item, pool members vote **Allow** or **Reject**. Optionally **Abstain**.
- **Decision rule:** e.g. "Allow only if at least 2/3 of voting members vote Allow within window T"; otherwise Reject.
- **Outcome:**
  - **Allow** → the original ContractDeploy tx is then treated as valid and can be included in a block (e.g. via a special "QA-approved" path or by re-submission with a proof).
  - **Reject** → the deployment is permanently rejected; the user can fix and try again (as a new tx).

8.3 Time limits: assets must not remain in the pool too long

It would be unfair for an asset to remain in the pool indefinitely if the community does not provide enough judgment (e.g. too few votes). The following ensure assets do **not** stay in limbo:

Mechanism	Description
Maximum time in pool (T)	Each item has a **deadline** (e.g. 7 or 14 days from entry). Governance sets T. After T, the item is **resolved by policy**—it does not remain in the pool.
Outcome when time expires	When the deadline is reached, one of two policies applies (chosen by governance and documented in the registry): **(A) Default to Reject** — if no quorum or no clear Allow by T, the deployment is **rejected**. The deployer can resubmit with changes. **(B) Default to Allow** — if no quorum or no clear Reject by T, the deployment is **allowed**. Most networks will prefer (A) for safety; the choice is explicit and transparent.
Quorum and early resolution	If **before** T the pool reaches quorum and meets the decision rule (e.g. 2/3 Allow or 2/3 Reject), the outcome is **immediate** and the item is removed from the pool. No need to wait until T.

Transparency	The deployer and the public can see time remaining and vote count. Pool members are encouraged to vote so assets are decided quickly; incentives (rewards for participation) support this.

Result: every asset that enters the pool receives a **final outcome** (Allow or Reject) by the deadline. None remain in the pool indefinitely.

8.4 Incentives and abuse

- **Rewards:** Pool members who participate in votes can earn a small reward (e.g. from protocol or from a fee on "QA-approved" deploys).
- **Slashing / reputation:** Malicious or clearly wrong decisions (e.g. Allow on malware) can be slashed or downgraded (reputation) if we have a way to detect abuse (e.g. later governance vote, or appeal by community).
- **Transparency:** All pending items and votes are public so the community can see what is "unsure" and how the pool decided.

8.5 Relation to existing governance

- The **community QA pool** can live in `boing-governance` (e.g. a new proposal type "QA vote") or in a dedicated `boing-qa` crate that the node and execution layer depend on.
- Pool **parameters** (who is in the pool, threshold, time window, maximum time T) can be changed via the existing time-locked governance, so we don't need a second chain.

---

9. Minimizing Edge Cases (Majority Decided by Automation)

A **majority** of assets should be **approved or rejected by the automated QA regulatory system**, not end up in the community pool. Edge cases should be the exception. The following algorithms, standards, and design choices keep the share of pool referrals low.

9.1 Set standards for every aspect of each asset

- **Complete rule coverage:** For each asset type (contracts, tokens, NFTs, etc.), define a **set standard** for every aspect: size, opcodes, well-formedness, purpose categories, declaration format, and (where applicable) metadata schema. When every aspect has a clear pass/fail or allow/reject threshold, automation can decide most cases without deferring.
- **Explicit purpose categories:** Maintain a **closed or well-defined list** of purpose categories (e.g. dApp, token, NFT, meme, community, entertainment, tooling, other). If the deployer picks a valid category and provides a valid declaration, automation can **Allow** without treating "no traditional use-case" as a reason for Unsure. This directly reduces edge cases for assets like memes (see §10).
- **Narrow gray zones:** Where heuristics are used (e.g. legitimacy, similarity to scam), define **narrow** gray zones or avoid them: e.g. two thresholds (reject above R, allow below A) with a small band [A, R] that yields Unsure. Prefer tightening A and R over time so most scores fall clearly in Allow or Reject.

9.2 Prefer hard rules; use soft rules sparingly

- **Default to hard rules:** Where a check is well-defined and binary (e.g. bytecode size, opcode whitelist, blocklist match), make it a **hard** rule: fail → Reject. That way automation never sends those cases to the pool.
- **Soft rules only when necessary:** Use **soft** rules only when the check has non-negligible false positives (e.g. a heuristic that might flag legitimate code). Keep the list of soft rules short and review it regularly; consider promoting a soft rule to hard once the criterion is well understood (e.g. after governance adds a precise definition).

9.3 Limit policy-required pool review

- **Short list of "always review" categories:** If governance designates certain categories as "always send to pool," keep that list **short** and high-stakes only (e.g. a specific "high-value financial" subtype), not broad categories like "token" or "meme." For most categories, automation should be able to Allow when specs and purpose pass.

9.4 New / unknown pattern: allow when all hard rules pass and declaration is valid

- **Avoid defaulting "first of its kind" to Unsure:** When an asset is new (does not match any known-good or known-bad pattern), the protocol can treat it as **Allow** if: (a) all hard rules pass, (b) it is not on the blocklist, and (c) it has a **valid purpose declaration** (e.g. category = meme, dApp, token, …). Only send to pool when there is a **positive reason** for Unsure (e.g. conflicting signals, or category is in the "always review" list). This prevents "new/unknown" from becoming a large source of edge cases.

9.5 Conflicting signals: resolve by rule priority or narrow the conditions

- **Rule priority:** If two checks conflict (e.g. one pass, one suspicious), define a **priority**: e.g. blocklist and hard rules override heuristics; if the only conflict is between two heuristics, use a single "combined" rule (e.g. reject only if both flag) so that most cases resolve to Allow or Reject.
- **Reduce conflicts over time:** When conflicts appear often for a certain pattern, add or refine a hard rule so that future cases are decided automatically.

9.6 Regular review and tightening

- **Governance reviews rules periodically:** Add new **hard** rules when new abuse patterns emerge (so more cases become automatic Reject). Widen allow criteria where safe (e.g. explicit meme category) so more cases become automatic Allow. Shrink gray zones and shorten the "always review" list. The goal is a **set standard** for every aspect and a **majority** of assets decided by automation.

---

10. Meme Assets and No Discrimination

Meme assets are explicitly allowed. There must be no discrimination, abuse, or bullying against any category—including meme. All categories are subject to the same QA standards (specs + no scam), but **meme** is a legitimate purpose that can be approved by automation when standards are met.

10.1 Meme as a valid purpose category

- **Purpose categories include meme (and community / entertainment):** The protocol's list of valid purpose categories **includes** "meme," "community," "entertainment," or equivalent. Deployers of meme assets declare this category (and optional description). Automation **does not** treat "no solid use-case" or "no traditional utility" as a reason to reject or to send to the pool.
- **Same QA bar:** Meme assets must still pass **all** applicable rules: bytecode specs, well-formedness, no blocklist match, valid declaration, and no scam heuristics in the reject zone. There is no lower bar for memes—only that **meme/entertainment/community is a valid use case** and is not discriminated against.

10.2 No discrimination in any category

- **No abuse or bullying:** The QA system and the community pool must not discriminate against or abuse any category (meme, cultural, experimental, etc.). The same rules apply to all; no category is singled out for harsher treatment or automatic Unsure.
- **Edge cases for meme:** If a meme asset triggers Unsure (e.g. ambiguous declaration, conflicting signal), it goes to the pool like any other edge case. Pool members must apply the same standards and must not reject solely because the asset is a meme. Governance can set and enforce policy that discrimination or abusive voting (e.g. "reject all memes") is subject to slashing or reputation penalty.

10.3 Summary

- Meme assets are **allowed** and can be **automatically approved** when they meet the set standard (specs + valid purpose declaration + no scam). They do not need a "solid use-case" beyond meme/community/entertainment.
- No category—including meme—may be discriminated against. The network enforces one quality bar for all; purpose diversity (including meme) is explicitly supported.

---

11. Resolving Currently Known Edge Cases by Automation (Pillar Optimization)

This section **enhances and optimizes** the sixth pillar (True quality assurance): **all currently known edge-case scenarios are resolved by the automation** within the regulatory QA system, so the community pool is only for genuinely unknown or future cases. **Leniency for meme culture** is built in; **no maliciousness or malignancy** is allowed.

11.1 Goal: known edge cases → Allow or Reject by automation

- The pool remains for **genuinely uncertain** situations (e.g. a truly new pattern the system has never seen, or a governance-mandated "always review" category). Every **currently known** type of edge case is mapped to a **deterministic rule** so the system returns **Allow** or **Reject**, not Unsure.
- Outcome: the sixth pillar is **optimized**—automation handles the full set of known scenarios; the pool handles only the residual unknown.

11.2 Currently known edge cases and how automation resolves them

The following table lists **currently known** edge-case types and the **automation rule or policy** that resolves each. Implement these so that these cases do **not** go to the pool.

Known edge case	How automation resolves it	Outcome
Meme / "no solid use-case"	Valid purpose category **meme**, **community**, or **entertainment**; declaration valid; not on blocklist; no scam heuristic in reject zone. Automation **does not** require "utility" or "solid use-case" for these categories.	**Allow** (leniency for meme culture).
Meme with minimal description	Category = meme/community/entertainment and (optional) short or minimal description is **valid**. Only missing or malformed declaration → Reject.	**Allow**.
Ambiguous declaration (e.g. category = "other")	Define in registry: **"other" is valid** if description (or hash) is non-empty and not forbidden. Only truly empty/invalid format → Reject. No Unsure for "other" + valid description.	**Allow** when description present and valid. **Reject** when declaration missing or invalid.
New / unknown pattern (first of its kind)	**Allow** when: (a) all hard rules pass, (b) not on blocklist, (c) valid purpose declaration (any valid category, including meme). Do **not** default new patterns to Unsure.	**Allow** (per §9.4).

Legitimacy heuristic in middle band	**Minimize gray zone:** define clear allow threshold A and reject threshold R. For categories **meme, community, entertainment**: do **not** use "no utility" or "no use-case" as a reject factor; only blocklist and explicit scam indicators (e.g. copy of known scam, phishing pattern) → Reject. Leniency: meme culture is not penalized by vague "suspicious" scores.	**Allow** when below R and not blocklisted; **Reject** only when above R or blocklist match.
Conflicting signals (one pass, one suspicious)	**Rule priority:** (1) Blocklist match → **Reject**. (2) Any hard rule fail → **Reject**. (3) For heuristics: use a **single combined rule** (e.g. reject only if *both* of two heuristics flag, or reject only if similarity > R). No Unsure for "one says pass, one says suspicious"—priority resolves it.	**Allow** or **Reject** by priority; no Unsure.
Soft rule failure (known soft rules)	**Convert to hard where possible:** if a soft rule has a precise definition (e.g. "no jump into PUSH data"), promote to **hard** so failure → Reject. For soft rules that remain (e.g. speculative heuristic): if **meme/community/entertainment** and not blocklisted and not in scam reject zone → **Allow** (meme leniency).	**Reject** if hard rule (after promotion); **Allow** if meme category + no malice.
Policy-required pool review	Keep the list **minimal**: only explicitly high-stakes categories (e.g. a specific "high-value financial" subtype) are "always review." **Meme, token, NFT, dApp, tooling, community, entertainment** are **not** in this list—they are resolved by automation.	**Unsure** only for the short, governance-defined "always review" list. All other categories → Allow or Reject by rules above.

11.3 Leniency for meme culture; no maliciousness or malignancy

Principle	Implementation
Leniency for meme culture	Meme, community, and entertainment are **valid** purpose categories. Automation **does not** reject or send to pool for "no solid use-case," "no utility," or "minimal description" when the category is one of these. Meme assets that pass specs and are not on the blocklist and do not trigger scam heuristics are **Allowed**.
No maliciousness / malignancy allowed	**Blocklist** (known scams, phishing, rug-pulls, malware hashes/patterns) → **Reject**. **Scam heuristics** in the reject zone (e.g. copy of known scam contract, forbidden naming pattern used to deceive) → **Reject**. **Hard rules** (bytecode safety, well-formedness, size) → fail → **Reject**. So: **lenient on purpose** (meme is fine), **strict on harm** (no scams, no malice).
Single line	Meme culture is welcome; malicious or malignant assets are not. Automation enforces both by explicit rules.

11.4 Pool only for genuinely unknown cases

After the above is implemented, the community QA pool receives **only**:

- Assets in a governance-defined **"always review"** category (short list), or
- A **genuinely new** situation the system has never encountered and for which no rule yet exists (e.g. a novel pattern that does not match any known-good or known-bad rule and triggers a conflict the priority rules do

not cover).

The goal is that **all currently known edge cases are resolved by automation**; the pool is the exception for the unknown and the policy-mandated few.

---

12. Automated and Decentralized by Design

All regulatory QA processes are automated and decentralized. The design explicitly defends against the deployment of malicious assets while avoiding a single point of control.

Principle	Implementation
Automation	Every check that can be run deterministically is run by the protocol: specs (bytecode, size, opcodes, well-formedness) and, where possible, purpose/legitimacy (declarations, blocklists, heuristics). No human-in-the-loop for clear allow/reject.
Decentralization	No single gatekeeper. Edge cases go to the **community QA pool**, whose members are designated by stake or governance. Threshold voting (e.g. 2/3) determines allow/reject. Parameters (members, threshold, time window) are set and updated via on-chain governance.
Defending against malicious deployments	Automation rejects known-bad patterns and non-compliant specs. Purpose checks and blocklists catch scams. When automation is unsure, the pool decides—and can be slashed or reputation-penalized for wrong decisions (e.g. allowing malware). Transparency: all pending items and votes are public so the community can hold the pool accountable.
No back doors	The rules are in the open; governance can change them only through the existing time-locked process. Decentralization is not "anyone can deploy anything"—it is "the network, collectively, enforces a quality bar without a central censor."

---

13. Integration with the Existing Network

- **No chain restart required** for adding QA. The current chain is append-only; we are only adding **new checks** on ContractDeploy (and future asset types).
- **Hook points:**
  1. **Node / RPC:** When handling `boing_submitTransaction` for a tx with `ContractDeploy`, run the QA module **before** `node.submit_transaction(signed)`. On Reject, return an RPC error with reason; on Unsure, return “pending QA” and record in pending QA queue.
  2. **Mempool:** Optionally, also run QA inside `Mempool::insert` for ContractDeploy so that even internal callers cannot insert rejected deploys. Same Reject/Unsure handling.
  3. **Execution:** In `Vm::execute_contract_deploy`, before `state.set_contract_code`, run the same (or a minimal subset of) checks. If they fail (e.g. due to a bug elsewhere), return `VmError` and the block fails (defense in depth).
- **Backward compatibility:** Existing deployed contracts are **unchanged**. They were deployed before QA existed; we do not retroactively remove them. New deployments must pass QA.
- **RPC and CLI:** New error codes and optional methods, e.g.:
  - `boing_submitTransaction` returns `-32050` (or similar) with message “Deployment rejected by QA: rule X”.
  - Optional: `boing_qaCheck(bytecode)` for pre-flight checks without submitting.

13.1 What automated QA does about malicious assets (morality / harm)

Automated QA does **not** apply open-ended "moral" judgment. It operationalizes **malice** (harm, abuse, scams) through **deterministic, protocol-defined rules** so that:

Layer	What automation does about malicious assets
Blocklist	Governance maintains a list of **bytecode hashes** of known scam/malware/phishing/rug-pull contracts. If the deployment's bytecode hash matches any entry → **Reject**. No deployment of that exact (or identical) code.
Scam patterns	Governance can add **byte sequences** (or behavioral signatures) that indicate known malicious code. If bytecode contains a listed pattern → **Reject**. Catches variants and copies of known-bad contracts.
Purpose	Only **valid purpose categories** are accepted (dApp, token, NFT, meme, community, entertainment, tooling, other). There is no "scam" or "malware" category; declaring a harmful purpose is not possible. Invalid category → Reject.
Hard rules	Size, opcodes, and well-formedness limit what can be deployed (e.g. no invalid opcodes that could hide exploit code). These are **spec** checks, not moral ones, but they reduce the space for harmful bytecode.
Edge cases (Unsure)	When automation is unsure, the **community QA pool** decides using the **canonical definition of malice** (Appendix B): scams, phishing, rug-pulls, malware, impersonation, deceptive naming, Ponzi patterns, spam/abuse. Pool members vote Reject if the asset fits those categories; otherwise Allow (with leniency for meme/experimental).

In short: Automation **rejects** malicious assets when they match the **blocklist** or **scam patterns** (governance-defined). It does **not** deploy vague morality—only the bar set by the protocol and governance. The **canonical malice definition** (Appendix B) is the **moral/harm bar**; automation enforces it via blocklist + patterns, and the **pool** enforces it for Unsure cases. **No scams whatsoever** is policy; automation and the pool together enforce it.

13.2 Implementation standards & industry alignment

The Boing QA implementation is designed to meet **quality and industry standards** for protocol-level deployment gating:

Standard	How Boing QA meets it
Determinism & consensus	All automated checks are **deterministic**: same (bytecode, purpose, rule set) → same Allow/Reject/Unsure on every node. No randomness, no external calls. Ensures consensus and replayability (see §6.1).
Single source of truth for opcodes	The **opcode whitelist** in `boing-qa` is aligned with `boing-execution` Boing VM bytecode (arithmetic, compare/bitwise, memory, storage, control flow, push, return — see `TECHNICAL-SPECIFICATION.md` §7.2). Any other byte → Reject. Prevents deployment of bytecode the VM cannot execute.
Defense in depth	QA runs at **mempool insert** (reject before broadcast) and again at **execution** (Vm::execute_contract_deploy). A bug or bypass in one layer does not allow bad bytecode into state.

Structured rejection	Every Reject carries a **rule_id**, **message**, and optional **doc_url** (link to deployer checklist). Clients and wallets can show actionable feedback; RPC `boing_qaCheck` returns the same structure for pre-flight.
Bounded resource usage	**Max bytecode size** (e.g. 32 KiB) limits state growth and DoS; **well-formedness** (no truncated PUSH, no trailing bytes) ensures a single linear pass. No unbounded loops or recursion in the checker.
Governance-updatable rules	Blocklist, scam patterns, and "always review" categories live in a **RuleRegistry**; production can use an on-chain or config-driven registry so rules evolve without a chain restart.
Transparent, auditable rules	Rule IDs and valid purpose categories are **documented** in this spec and in code constants; canonical malice definition (Appendix B) gives the pool a clear bar.
No single gatekeeper	Unsure → community QA pool; pool parameters and membership are governance-defined. Automation handles the vast majority; edge cases are decided collectively.

Primary deterring factors (reject) are implemented as **hard rules**: empty bytecode, size over limit, invalid opcode, malformed stream, blocklist match, scam pattern match, invalid purpose category. **Allow** requires all hard rules to pass and no soft/policy trigger for Unsure. This aligns with industry practice: **gate at deployment time**, **deterministic checks**, **clear error reporting**, and **decentralized edge-case handling**.

---

14. Implementation Phases

Phase	Content
1. Rules and crate	Define the central list of rules (bytecode size, opcode whitelist, well-formedness, purpose categories including meme/community/entertainment). Add a `boing-qa` crate that implements deterministic checks and returns Allow / Reject / Unsure. Implement the **currently known edge-case resolutions** in §11 so automation handles meme leniency, ambiguous declaration, new/unknown pattern, conflicting signals, and soft-rule handling without sending those to the pool.
2. Node and mempool	Integrate QA at submit and/or mempool insert for ContractDeploy. Return clear errors; add optional `boing_qaCheck` RPC.
3. Execution defense in depth	Call QA (or a minimal subset) in `execute_contract_deploy`; fail execution if check fails.
4. Pending QA and pool	Implement pending QA queue and QA pool (members, voting, time window, outcome). Enforce **maximum time T** in pool and default outcome when T expires (e.g. default Reject). Integrate with block production so "QA-approved" deploys can be included.
5. Governance of rules	Allow governance to add or modify rules (e.g. new rule IDs, soft vs hard) via existing governance.
6. Tokens / NFTs	When we add first-class token or NFT deployment, extend the same QA framework to those asset types.

---

15. Chain Restart vs. Extension

- **Extension is sufficient.** Adding QA is a **new validation layer** on deployment; it does not change block format, transaction format, or past state. Upgrading nodes to a version that runs QA is enough; no need to "start the entire blockchain over."
- **Restart** would only be necessary if we changed something that breaks existing history (e.g. a new transaction format that invalidates old blocks, or a change in execution semantics for already-deployed contracts). That is **not** required for "only allow quality assets" going forward.

---

16. Additional Enhancements & Optimizations

The following enhancements would further strengthen the QA pillar. They are not required for the core design but improve developer experience, transparency, fairness, and operational clarity.

16.1 Pre-flight checks and developer experience

Enhancement	Benefit
Local QA in SDK/CLI	Run the same QA checks **locally** before submit (e.g. `boing deploy --dry-run` or SDK `qaCheck(bytecode, declaration)`). Developers see "Rejected: bytecode exceeds MAX_SIZE" or "Would be allowed" without hitting the network. Fewer failed submissions and faster iteration.
Structured rejection payload	Every Reject returns a **rule_id**, **human-readable message**, and optional **doc link** (e.g. "MAX_BYTECODE_SIZE: Bytecode exceeds 32 KiB. See QA rules."). Wallets and CLIs can display clear, actionable feedback.
Deployer checklist / one-pager	A short "How to pass QA" doc: valid purpose categories, max bytecode size, declaration format, how to avoid blocklist. Reduces edge cases caused by confusion and improves first-time success.

16.2 Transparency and governance

Enhancement	Benefit
Public QA metrics	RPC or dashboard: **Allow / Reject / Unsure counts** (e.g. per day or per category), **rejection reason distribution**, **average time in pool**, **pool resolution rate**. Supports 100% transparency and helps governance tune rules (e.g. if too many memes hit Unsure, widen allow rules).
Versioned rule sets	Each governance update produces a **rule set version** (e.g. hash or version id). Deployments are evaluated against the version in effect at submit time (or block time—document which). Prevents ambiguity when rules change and allows replay/audit ("this tx was checked under v2").
Regular rule-set review cadence	Governance commits to a **review cadence** (e.g. quarterly): add blocklist entries, promote soft→hard where safe, shrink gray zones, update "always review" list. Keeps the pillar operational and responsive to new abuse patterns.

16.3 Fairness and appeal

Enhancement	Benefit

Formal appeal path	If an asset is **Rejected** (or pool rejects), allow a **one-time appeal**: deployer submits a new tx with same bytecode + optional extra evidence (e.g. clarified declaration). Re-evaluated under current rules; if pool previously rejected, a second vote with clear "appeal" flag. Reduces wrongful rejections and builds trust.
Canonical definition of malice	Publish a **single list** of what "maliciousness/malignancy" means for QA: scams, phishing, rug-pulls, malware, impersonation, deceptive naming, Ponzi patterns. Implementers and pool members use this as the bar; no ambiguity.

16.4 Performance and scale (optional)

Enhancement	Benefit
Known-good fast path	Optional: a **whitelist** of bytecode or pattern hashes for previously approved contracts (e.g. by governance or after N successful deployments). Matching deployments get **instant Allow** without re-running all heuristics. Reduces load; use sparingly and with governance control to avoid gaming.
Pool quorum for early resolution	Require a **minimum number of votes** before a pool decision is final (e.g. at least 50% of pool members voted). Below quorum at deadline T, apply default (Reject or Allow by policy). Ensures decisions are representative.

16.5 Determinism and audit

Enhancement	Benefit
Replay and verification	Document that **same (bytecode, declaration, rule set version)** yields the **same result** on every node. Nodes can **replay** QA for any past deployment to verify. Reinforces trust and helps debugging.
QA attestation in block	Optional: block header or metadata includes a **QA summary** (e.g. "all deploys in this block: Allow by automation" or "one pool-approved"). Light clients or indexers can trust without re-running full QA.

Prioritizing **pre-flight checks**, **structured errors**, **public metrics**, and **versioned rule sets** would yield the highest impact for the QA pillar; **appeal path** and **canonical malice definition** strengthen fairness and clarity. The rest can be adopted as the network matures.

Appendix A: Deployer Checklist (How to Pass QA)

Quick reference for developers deploying contracts on Boing Network.

Pre-flight: Use `boing_qaCheck` First

Before submitting a deployment, run a pre-flight check:

```
curl -s -X POST http://127.0.0.1:8545/ -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","id":1,"method":"boing_qaCheck","params":
["0x<hex_bytecode>","meme",""]}'
```

Result will be `allow`, `reject`, or `unsure`. If `reject`, the response includes `rule_id` and `message`.

Hard Rules (Must Pass)

Rule	Limit	Action
Bytecode size	≤ 32 KiB	Shrink or split contract
Valid opcodes only	Boing VM opcode set in `boing-qa` / §7.2 (includes Div, Mod, AddMod, MulMod, compare, bitwise, etc.)	Remove invalid bytes
Well-formed	No truncated PUSH, no trailing bytes	Fix bytecode stream
Not blocklisted	Bytecode hash not in blocklist	Contact governance if wrongly blocked
Purpose (if provided)	Must be valid category	Use one of the valid categories below

Valid Purpose Categories

dApp / dapp, token, NFT / nft, meme, community, entertainment, tooling, other (provide description hash for best outcome).

Common Rejections and Fixes

rule_id	Cause	Fix
`MAX_BYTECODE_SIZE`	Bytecode > 32 KiB	Reduce size or split
`INVALID_OPCODE`	Byte contains non-Boing opcode	Use only Boing VM opcodes
`MALFORMED_BYTECODE`	Truncated PUSH or trailing bytes	Validate instruction stream
`BLOCKLIST_MATCH`	Hash matches known scam/malware	Cannot deploy; contact if error
`SCAM_PATTERN_MATCH`	Contains known malicious pattern	Remove or refactor
`PURPOSE_DECLARATION_INVALID`	Invalid category	Use valid category from list above

When You Get "Unsure"

"Unsure" means the deployment is referred to the community QA pool. Common triggers: purpose = "other" with minimal or no description; category in "always review" list. Provide a clear purpose category and (for "other") a description hash to reduce pool referrals. **Meme, community, and entertainment are valid purposes** — no extra justification required.

Appendix B: Canonical Definition of Malice

Single source of truth for what "maliciousness" and "malignancy" mean in Boing's QA system. Pool members and implementers use this as the bar.

Scope

Malice = assets that should be **Rejected** (or never Allowed by the pool) because they cause harm or abuse.

Malice Categories

Category	Definition	Examples
Scams	Deceptive schemes to extract value without fair exchange	Fake tokens, phishing contracts
Phishing	Impersonation or deceptive UX to steal credentials or funds	Fake wallet interfaces, fake airdrops
Rug-pulls	Promised functionality removed or disabled to trap funds	Liquidity withdrawal, exit scams
Malware	Code designed to harm user systems or steal data	Keyloggers, backdoors
Impersonation	Misleading naming or branding to appear as a trusted entity	Fake "official" tokens, copycat projects
Deceptive naming	Names intended to mislead about purpose or risk	"Safe" in name of risky asset, misleading tickers
Ponzi patterns	Unsustainable reward structures that rely on new deposits	Pyramid schemes, unsustainable yields
Spam / abuse	Bulk low-value deployments to clog state or confuse users	Mass empty contracts

How This Is Enforced

- **Automation:** Blocklist (bytecode hashes), scam patterns (byte sequences), hard rules (size, opcodes).
- **Community pool:** When automation returns Unsure, pool members vote Allow or Reject using this definition.
- **Governance:** Can add blocklist entries, scam patterns, or "always review" categories.

What Is NOT Malice

- **Meme assets** — Legitimate purpose when declared.
- **Experimental or novel** — New patterns that don't match malice categories.
- **Unconventional art or culture** — As long as not deceptive or harmful.
- **Failed or unpopular projects** — Poor quality ≠ malice.

Pool Member Guidance

When voting on an Unsure item: (1) Does it match any malice category above? → **Reject**. (2) Is there reasonable doubt it could cause harm? → **Reject**. (3) Is it merely unproven or unconventional? → **Allow** (per meme/community leniency). (4) In doubt? → Default to **Reject** (safety-first).

Vulgarity, offensiveness, and content moderation (current scope)

Current scope: The canonical malice definition above is **harm-focused** (scams, phishing, malware, impersonation, etc.). The protocol does **not** today filter deployments for **vulgarity**, **offensiveness**, or other purely content-based criteria (e.g. slurs, obscenity, hate speech in names or metadata).

Why: (1) **Bytecode** is opaque — the QA layer sees opcodes and data bytes, not human-readable text. (2) **Purpose** is a category plus an optional **description hash**, not plaintext, so the protocol does not see deployer-written text to

scan for offensive strings. (3) **Token names, symbols, NFT metadata** are not part of the current ContractDeploy payload that QA inspects; they typically live in contract storage or off-chain metadata, which the deploy-time checker does not evaluate.

Can it be added? Yes, **in principle**, if the protocol or ecosystem exposes **metadata** (e.g. token name, symbol, NFT name/description) at deploy or registration time and governance adopts a policy:

- **Blocklist of forbidden strings** — e.g. a governance-maintained list of slurs or hate terms; metadata containing a listed string → Reject (or Unsure for pool).
- **Pool policy** — Governance can instruct the community QA pool to Reject deployments whose **known metadata** (if any is submitted) is vulgar, offensive, or otherwise in violation of a published content policy.
- **"Always review" for certain asset types** — e.g. tokens or NFTs with a metadata field could be routed to the pool when metadata is present, and pool members apply a content policy.

Trade-offs: Protocol-level filtering of vulgarity/offensiveness is **subjective** and **jurisdiction-dependent**; what counts as offensive varies by culture and law. It can also be seen as **content moderation** or **censorship**, and can conflict with **leniency for meme culture** and **no discrimination** if applied unevenly. Deciding whether to extend QA to cover vulgarity/offensiveness is therefore a **governance and policy choice**, not a technical limitation. The current design keeps the bar **objective and harm-based**; any future content policy would need to be clearly defined and adopted through governance.

Appendix C: Governance-mutable QA rules

> ***Purpose:*** *How to update protocol QA rules (including the content blocklist for vulgarity/offensiveness) via governance so that vulgar and offensive assets are not deployed. All QA rule sets are* ***mutable*** *through the standard governance process.*

Read-only transparency: The effective registry on any node is returned by JSON-RPC `boing_getQaRegistry` (no params). For a **documented baseline** to diff against, see **docs/config/CANONICAL-QA-REGISTRY.md** and the JSON files in `docs/config/`.

C.1 What can be updated

The following are **governance-mutable**:

Item	Description
Content blocklist	Forbidden substrings in asset name/symbol (vulgarity, offensiveness). Case-insensitive. When a deploy includes `asset_name` or `asset_symbol` (see `ContractDeployWithPurposeAndMetadata`), any term on this list causes **Reject** (`CONTENT_POLICY_VIOLATION`).
Bytecode blocklist	Hashes of known scam/malware bytecode. Match → Reject.
Scam patterns	Byte sequences that, if found in bytecode, → Reject.
Always-review categories	Purpose categories that always go to the community QA pool (Unsure).
Max bytecode size	Governance can change the limit (e.g. 32 KiB).

C.2 Governance flow

1. **Propose** — Create a governance proposal with:
   - **target_key:** `qa_registry` (see `boing_qa::GOVERNANCE_QA_REGISTRY_KEY`)
   - **target_value:** JSON-serialized `RuleRegistry` (see below).
2. **Vote / cooling / execution** — Use the existing phased governance (Proposal → Cooling → Execution). When the proposal is **executed**, the node (or operator) applies the new registry:
   - Deserialize: `boing_qa::rule_registry_from_json(&target_value)`.
   - Replace the in-memory QA rule registry used by the mempool and RPC with this registry.
3. **Persistence** — Nodes should save the updated registry to `qa_registry.json` and pool config to `qa_pool_config.json` (under the data directory) so they survive restarts until the next governance update. Use `BoingNode::set_qa_policy` to apply both in memory and on disk.

C.3 JSON format for `RuleRegistry`

The `target_value` is a JSON object with the same shape as `RuleRegistry`:

```
{
  "max_bytecode_size": 32768,
  "blocklist": ["<base64 or hex of 32-byte hashes>"],
  "scam_patterns": ["<base64 or hex of byte sequences>"],
  "always_review_categories": ["token", "financial"],
  "content_blocklist": ["term1", "term2", "slur1", "vulgar_word"]
}
```

- **content_blocklist:** List of forbidden substrings. Deployment metadata (asset_name, asset_symbol) is checked case-insensitively; if any string in the list appears as a substring, the deploy is **Reject**ed with `CONTENT_POLICY_VIOLATION`.
- **blocklist:** In Rust `RuleRegistry` this is `Vec<[u8; 32]>`. In JSON you can use an array of hex strings (64 chars) or base64.
- **scam_patterns:** In Rust `Vec<Vec<u8>>`. In JSON use an array of hex or base64 strings.

Governance should maintain a **content_blocklist** of terms that the network does not allow in asset names or symbols (vulgarity, slurs, offensiveness). Add or remove terms by proposing a new full `RuleRegistry` JSON with the updated list.

C.3.1 QA pool governance (`qa_pool_config`)

Separate proposal key `qa_pool_config` (`boing_qa::GOVERNANCE_QA_POOL_CONFIG_KEY`) controls who may resolve **Unsure** deploys and how large the pending queue may grow. Deserialize with `boing_qa::qa_pool_config_from_json`. Nodes persist it as `qa_pool_config.json` next to `qa_registry.json`.

Example JSON:

```
{
  "administrators": ["0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"],
  "max_pending_items": 32,
  "max_pending_per_deployer": 2,
```

```
  "review_window_secs": 604800,
  "quorum_fraction": 0.5,
  "allow_threshold_fraction": 0.6666666666666666,
  "reject_threshold_fraction": 0.6666666666666666,
  "default_on_expiry": "reject",
  "dev_open_voting": false
}
```

- **administrators:** 32-byte account IDs (hex). Only these voters count for governance-final decisions on pooled deploys. Leave empty only with `dev_open_voting: true` (local dev).
- **max_pending_items:** Global cap; when reached, new Unsure submissions are refused (**anti-congestion**). Set `0` to disable the pool entirely.
- **max_pending_per_deployer:** Per-address cap (`0` = unlimited).
- `BoingNode::set_qa_policy` updates mempool, VM, executor, and pool together so rules stay aligned.

C.4 Deploy-time metadata (asset name / symbol)

To have content policy apply, deployers must use the **ContractDeployWithPurposeAndMetadata** payload and supply optional `asset_name` and/or `asset_symbol` :

- **asset_name:** Optional, max 256 UTF-8 bytes. Checked against `content_blocklist` .
- **asset_symbol:** Optional, max 32 UTF-8 bytes. Checked against `content_blocklist` .

If the deploy uses the legacy **ContractDeploy** or **ContractDeployWithPurpose** (without metadata), no name/symbol is checked and content policy does not apply. Wallets and CLI should be updated to support the new payload and to pass name/symbol when deploying tokens or NFTs so that vulgar/offensive names can be rejected.

C.5 RPC and SDK

- **boing_qaCheck** accepts optional params: `[hex_bytecode, purpose_category?, description_hash?, asset_name?, asset_symbol?]` . Use the last two to pre-flight content policy.
- **boing_submitTransaction** — When building a deploy tx, use the payload variant that includes `asset_name` and `asset_symbol` if you want them checked.

C.6 Summary

- **Content blocklist** is governance-mutable; add forbidden terms (vulgarity, offensiveness) via proposals.
- **Full registry** (blocklist, scam patterns, always-review categories, max size) is replaceable by executing a governance proposal with key `qa_registry` and value = JSON `RuleRegistry` .
- Deploy payload **ContractDeployWithPurposeAndMetadata** carries optional asset name/symbol; QA rejects if they contain any governance-forbidden term.

Appendix D: Upgrade / proxy patterns (QA context)

Boing does **not** use delegate-to-runtime indirection that swaps implementation code in place. **Immutable bytecode per** `AccountId` is the default after a successful deploy.

Permitted product patterns include a **fixed hub contract** that stores a pointer to the current implementation `AccountId` and **forwards** via `ContractCall` ; each **new** implementation is a **separate deploy** that must **pass QA** again. See full guidance: BOING-PATTERN-UPGRADE-PROXY.md.

Discouraged: opaque or evasive patterns whose purpose is to **circumvent** protocol QA visibility into what users execute. Such bytecode may be evaluated under **scam-pattern** and **legitimacy** rules (§5, Appendix B).

Summary

- **Mission:** The QA regulatory system confirms that all assets are up-to-par (specs + purpose) before approving deployment. No scams whatsoever. All processes are automated and decentralized, with decentralization defending against malicious deployments.
- **Two pillars:** (1) **Specs**—technical compliance (bytecode, size, opcodes, well-formedness, interfaces, metadata). (2) **Purpose**—legitimate reason for deploying; blocklists and heuristics catch scams; edge cases go to the pool.
- **Automation:** Reject deployments that violate specs or purpose rules; return Unsure when the system is not confident. No human-in-the-loop for clear cases.
- **Community pool:** For edge cases **throughout the network's lifespan**, whenever the system does not know allow vs reject, the asset falls into the QA community pool. What determines "edge case" (Unsure) is defined in §7. Assets **do not remain in the pool indefinitely**: a maximum time T (e.g. 7–14 days) is set by governance; by the deadline the item is resolved (default Reject or Allow by policy). See §8.3.
- **Majority decided by automation:** A majority of assets should be approved or rejected by the automated system, not the pool. §9 describes how to minimize edge cases: set standards for every aspect, prefer hard rules, narrow gray zones, limit "always review" categories, and allow new/unknown when hard rules pass and declaration is valid.
- **Meme assets and no discrimination:** Meme (and community/entertainment) are valid purpose categories. Meme assets are explicitly allowed and can be auto-approved when they meet QA standards; no discrimination or abuse against any category (§10).
- **Pillar optimization (§11):** All **currently known** edge cases are resolved by automation (Allow or Reject); the pool is only for genuinely unknown or policy-mandated cases. **Leniency for meme culture** (no reject for "no use-case"); **no maliciousness or malignancy** (blocklist and scam heuristics → Reject).
- **Rules:** A single, transparent set of attributes and rules for each asset type (specs and purpose), updatable via governance.
- **Apply to existing network:** Add a QA crate, hook at submit/mempool and execution; no chain restart. Existing contracts remain; only new deployments are gated.
- **Additional enhancements (§16):** Pre-flight checks, structured errors, public metrics, versioned rule sets, appeal path, canonical malice definition, deployer checklist, and optional fast-path/quorum/attestation improvements can further strengthen the pillar.

This document can be used as the spec for implementing the QA system in the Boing Network codebase and for communicating the feature to the community and developers.