Election Contract

A Multi-Role, Multi-Rule On-Chain Voting System on Ethereum

https://github.com/tvtrungg/iob-election *(Project repository)*

Vinh-Trung THIEU - 21415515
Yousr BENTOUNES - 21520010

February 05, 2026

Contents

1 Executive Summary

This report presents the design, implementation, and security analysis of `Election`, a decentralized voting system implemented as an Ethereum smart contract and a lightweight web frontend (HTML/CSS/JavaScript with Web3.js and Chart.js). The system supports three voting rules:

- **First-Past-The-Post with turnout quorum** (FPTP_Quorum),
- **Proportional representation by integer percentage** (Proportionnel),
- **Instant-Runoff Voting** (InstantRunoff / IRV).

A core design goal is to reduce single points of failure by distributing authority across two independent committees:

- an **Admin committee** that closes the election using a **2/3 super-majority threshold** (over N admins),
- a **Registrar committee** that validates voter registrations using a **2/3 super-majority threshold** (over M registrars).

The implementation enforces integrity constraints (one vote per registered address and one vote per identity hash), deterministic on-chain tallying, and event-based auditability. However, it does **not** provide strong ballot privacy: identifiers may be disclosed in transaction calldata.

2 Scope and Artifacts

2.1 In-Scope Components

- **Smart Contract**: `Election` (Solidity `^0.8.0`), deployed to an EVM-compatible chain.
- **Frontend dApp**: HTML/CSS UI and JavaScript logic using `Web3.js` and `Chart.js`.
- **Operational workflow**: deployment, registrar validation, vote casting, multi-admin election closure, and results visualization.
- **Security objectives addressed**: authorization, integrity, auditability, and deterministic tallying.

2.2 Out-of-Scope Components

- Formal verification, third-party audit, automated fuzzing, and advanced adversarial testing.
- Real-world identity verification (KYC) and strong Sybil resistance beyond the registrar committee process.
- Cryptographic privacy guarantees (receipt-freeness, coercion resistance, end-to-end verifiability).
- Support for additional voting rules beyond the three implemented systems.

3 System Overview

3.1 Roles

The protocol defines three actor categories:

- **Admins**: a fixed set of N addresses configured at deployment. Admins collectively close the election via a threshold approval.
- **Registrars**: a fixed set of M addresses configured at deployment. Registrars validate voter registration by attesting that an address corresponds to a provided identifier string.
- **Voters**: any non-admin address that becomes *fully registered* after receiving enough registrar validations, and can cast exactly one vote (subject to identity-hash uniqueness constraints).

3.2 Workflow

Step 1: **Deployment**: deploy `Election` with the voting system type, admin set, and registrar set.

Step 2: **Registration validation**: registrars call `validateVoter(voterAddress, idString)`; once the threshold is reached, the voter becomes registered.

Step 3: **Vote casting**: registered voters call `castVote(choices, idString)` with a rule-specific ballot format.

Step 4: **Election closure**: admins call `proposeAndCloseElection()` until the threshold is met and the election closes.

Step 5: **Results**: after closure, any user can call `getResults()`; the frontend displays the final chart.

4 Threat Model and Security Objectives

4.1 Assumptions

- The EVM execution environment is correct (deterministic execution, immutability of confirmed blocks).
- Admin and registrar keys are not compromised beyond the configured threshold.
- The registrar committee performs off-chain checks honestly; the contract cannot verify real-world identity authenticity.
- Voters use wallets capable of signing transactions; end-user devices are assumed uncompromised for the purpose of this report.

4.2 Security Objectives

- **Integrity**: prevent double voting by address and by identity hash.
- **Authorization**: restrict sensitive actions (registration validation and election closure) to the correct roles.
- **Auditability**: emit events for key transitions (validation, registration completion, vote casting, closure, results publication).
- **Deterministic tallying**: compute results exclusively from on-chain state.

4.3 Committee Threshold Rationale (Byzantine-Style Super-Majority)

The contract enforces a **super-majority quorum** (approximately 2/3) for critical actions: `validateVoter` (registrars) and `proposeAndCloseElection` (admins). This choice aligns with standard Byzantine fault-tolerance intuition: if fewer than one-third of committee members are faulty or malicious, a 2/3 quorum ensures that honest members dominate decisions and quorums intersect.

Let n denote the committee size ($n = N$ for admins or $n = M$ for registrars). A classical BFT feasibility bound is:

$$n \geq 3f + 1,$$

meaning the system can tolerate up to $f < n/3$ Byzantine members. A 2/3-style quorum ensures that two quorums intersect in at least one honest member when $f < n/3$, reducing the risk of conflicting decisions. Thus, a $2n/3$ approach as a condition is perfect.

Implementation detail. Using integer arithmetic, the contract computes the required number of signatures as:

$$Q = \left\lfloor \frac{2n+2}{3} \right\rfloor,$$

which is equivalent to $\lceil 2n/3 \rceil$ for integers $n \geq 1$.

Committee size n	Max tolerated f (intuition)	Threshold $Q = \lfloor (2n+2)/3 \rfloor$	Result
4	1	3	3/4 approvals required
10	3	7	7/10 approvals required
100	33	67	67/100 approvals required

Table 1: Super-majority threshold derived from contract arithmetic

4.4 Non-Objectives / Limitations

- **Strong privacy is not achieved**: identifiers may be observable, and ballots are linkable via event logs.
- **Front-running resistance is not explicitly addressed**.
- **Scalability is bounded**: Instant-Runoff requires iterating over voter addresses on-chain; practical only for small elections.

5 Smart Contract Specification

5.1 Contract Interface Summary

The contract exposes:

- role queries: `isAdmin(address)`, `isRegistrar(address)`
- election state: `currentSystem()`, `electionClosed()`, counters, vote tallies
- actions: `validateVoter(...)` (registrar), `castVote(...)` (voter), `proposeAndCloseElection()` (admin)
- results: `getResults()` (only after closure)

5.2 Voting Systems

```
enum VotingSystem { FPTP_Quorum , Proportionnel , InstantRunoff }
```

Listing 1: Voting system enum (conceptual)

5.2.1 FPTP with Quorum (FPTP_Quorum)

- Ballot: exactly one candidate choice $c \in \{1, \dots, 5\}$.
- Tally: the candidate with the most votes wins *if and only if* turnout is strictly above 50% of registered voters.
- Output: per-candidate counts and either a winner message or a quorum failure message.

5.2.2 Proportional (Proportionnel)

- Ballot: exactly one candidate choice $c \in \{1, \dots, 5\}$.
- Tally: compute integer percentages:

$$\text{percentage}[c] = \left\lfloor \frac{\text{votes}[c] \cdot 100}{\texttt{totalVotesCast}} \right\rfloor.$$

- Output: integer percentage shares; the contract reports the distribution rather than selecting a single winner.

5.2.3 Instant-Runoff (InstantRunoff / IRV)

- Ballot: a full ranking of all 5 candidates (array length must be 5).
- Tally: iteratively eliminate the lowest candidate and transfer ballots to the next non-eliminated preference until a candidate exceeds 50% of votes.
- Output: the final round vote distribution and a winner message if a strict majority is reached; otherwise a terminal message.

6 Data Model and State Variables

6.1 Constants and Counters

- `NB_CANDIDATES = 5`: fixed number of candidates.
- `totalVotersRegistered`: incremented once per voter when reaching the registrar threshold.
- `totalVotesCast`: incremented once per successful vote.

6.2 Voter Record

Each voter address maps to a `Voter` struct:

```
struct Voter {
    bool isRegistered;
    bool hasVoted;
    bytes32 hashedID;
    uint[] preferences;
    uint validationCount;
}
```

Listing 2: Voter struct (as implemented)

Semantics.

- `isRegistered`: becomes true after meeting the registrar signature threshold.
- `hasVoted`: becomes true after casting a vote once.
- `hashedID`: `keccak256(abi.encode(idString))` associated with the voter.
- `preferences`: used only in IRV; stores the ranking.
- `validationCount`: number of unique registrar validations received.

6.3 Governance Committees and Thresholds

6.3.1 Admin Committee

- `admins`: list of N admin addresses.
- `isAdmin[a]`: membership mapping.
- `closeElectionVotes[a]`: tracks whether admin a has approved closure.
- `requiredAdminSignatures`: threshold computed at deployment.

Threshold formula. The contract computes:

$$Q_N = \left\lfloor \frac{2N+2}{3} \right\rfloor \equiv \left\lceil \frac{2N}{3} \right\rceil.$$

6.3.2 Registrar Committee

- `isRegistrar[r]`: membership mapping for registrar addresses.
- `requiredRegistrarSignatures`: computed similarly as $\lceil 2M/3 \rceil$.

6.4 Identity and Validation Tracking

- `idToAddress[hashedID]`: binds a hashed identifier to a single voter address (set on first validation).
- `hasValidatedRegistration[hashedID][registrar]`: prevents the same registrar from validating the same ID twice.
- `idHasAlreadyVoted[hashedID]`: prevents reuse of the same hashed identity across multiple votes.

Privacy note. These integrity mappings may also create a persistent link between an identity hash and a voter address. If the identifier has low entropy or is disclosed on-chain, observers can potentially de-anonymize participants.

6.5 Vote Storage

- For FPTP and Proportional: `firstChoiceCounts[1..5]` counts votes by candidate.
- For IRV: votes are stored as per-voter rankings plus the `voterAddresses` list used during tallying.

Indexing convention. `firstChoiceCounts` is declared as `uint[6]` and uses indices 1 to 5. Index 0 is unused.

7 Events and Auditability

The contract emits events to support off-chain monitoring and UI updates:

- `VoterValidationReceived(voter, registrar)`: a registrar validated a voter/ID pair.
- `VoterFullyRegistered(voter, hashedID)`: voter reached threshold and became registered.
- `VoteCast(voter, choices)`: a vote was cast (choices are stored in logs).
- `AdminActionApproved(approver, action)`: admin approvals for closure.
- `ElectionClosed()`: election transitioned to the closed state.
- `ResultsPublished(system, winner, scores)`: results emitted at closure time.

Transparency vs. privacy. Event logs improve auditability but reduce practical ballot secrecy because choices are publicly linkable to voter addresses.

8 Functional Specification (By Function)

This section describes the intended behavior and key checks for each externally relevant function.

8.1 `constructor(VotingSystem, N, admins[], M, registrars[])`

Purpose: initialize election configuration and committee memberships.

Key checks and effects:

- Requires `admins.length == N` and `registrars.length == M`.
- Requires $N > 0$ and $M > 0$.
- Sets `currentSystem`, `N_admins`, `M_registrars`.
- Computes thresholds using integer arithmetic consistent with $\lceil 2n/3 \rceil$.
- Populates `isAdmin`, `admins`, and `isRegistrar`.

8.2 `proposeAndCloseElection()`

Access control: `onlyMultiAdmin`.

Purpose: collectively close the election via threshold approvals.

Checks:

- Election must not already be closed.
- Caller must not have already approved closure.

Effects:

- Records caller approval in `closeElectionVotes`.
- Counts total approvals by iterating over `admins`.
- Emits `AdminActionApproved`.
- If approvals $\geq$ `requiredAdminSignatures`, sets `electionClosed = true` and emits `ElectionClosed`.
- Publishes results by computing `getResults()` and emitting `ResultsPublished`.

Operational note. The contract does not reset `closeElectionVotes` after closure, which is acceptable since the election is single-shot.

8.3 `validateVoter(address voterAddress, string idCarte)`

Access control: `onlyRegistrar`.

Purpose: validate that a voter address corresponds to a provided identifier string.

Core logic:

- Computes $h = keccak256(abi.encode(idCarte))$.
- Binds h to `voterAddress` on first use via `idToAddress`.
- Prevents the same registrar from validating the same ID twice.
- Prevents validation if the voter is already fully registered.
- Increments `validationCount` and emits `VoterValidationReceived`.
- If `validationCount` reaches threshold, sets `isRegistered=true`, increments `totalVotersRegistered`, and emits `VoterFullyRegistered`.

8.4 `castVote(uint[] choices, string idCarteFourni)`

Purpose: allow a registered voter to cast a rule-compliant ballot.

Checks:

- Election must be open.
- Admins are forbidden from voting.
- Sender must be fully registered.
- Provided ID hash must equal stored `hashedID`.
- Sender must not have voted before.
- The hashed ID must not have been used to vote before (`idHasAlreadyVoted`).

Rule-dependent ballot validation:

- **IRV**: requires `choices.length == NB_CANDIDATES`. Stores ranking and appends sender to `voterAddresses`.
- **FPTP / Proportionnel**: requires `choices.length == 1`, candidate in $[1, 5]$, increments `firstChoiceCounts[c]`.

Effects:

- Sets `hasVoted=true` and `idHasAlreadyVoted=true`.
- Increments `totalVotesCast`.
- Emits `VoteCast`.

8.5 `getResults()`

Type: `view`.

Purpose: return computed results after election closure.

Check: requires `electionClosed == true`.

Return values:

- `system`: string label for the voting rule.
- `winnerMsg`: a winner or status message.
- `finalScores`: an array of size 6 (indices 1..5 meaningful).

8.6 Internal Result Functions

8.6.1 `_calculateFPTP()`

- Quorum: if `totalVotesCast` $\leq \lfloor$`totalVotersRegistered`$/2\rfloor$, returns `"Quorum not reached (<50%)"`.
- Otherwise returns the candidate with the maximum vote count.

8.6.2 `_calculateProportionnel()`

- If no votes were cast, returns `"No votes"` and an all-zero score array.
- Otherwise returns integer percentages per candidate.

8.6.3 `_calculateInstantRunoff()`

- Maintains an `eliminated` set and recomputes round totals by scanning each voter's preference list until a non-eliminated candidate is found.
- If any candidate exceeds $\lfloor$`totalVotesCast`$/2\rfloor$, declares a winner.
- Otherwise eliminates the non-eliminated candidate with the minimum votes and repeats.
- Stops after `NB_CANDIDATES-1` rounds.

Tie handling. If multiple candidates tie for the minimum, the implementation eliminates the first encountered candidate with that minimum. This is deterministic but may be undesirable in real elections; explicit tie-breaking rules are recommended.

8.7 `uint2str(uint)`

Utility function converting an unsigned integer to its decimal string representation for result messages.

9 Correctness and Invariants

9.1 Key Invariants

Let R be the set of addresses such that `voters[a].isRegistered == true`.

- **No double voting by address**: once `voters[a].hasVoted` is true, subsequent `castVote` calls from a revert.
- **No double voting by identity hash**: once `idHasAlreadyVoted[h]` is true, any attempt to vote with the corresponding ID reverts.
- **Registration threshold monotonicity**: `validationCount` only increases; `isRegistered` transitions from false to true at most once.
- **FPTP/Proportional tally consistency**: for these systems, $\sum_{c=1}^{5}$ `firstChoiceCounts[c]` $=$ `totalVotesCast` when every vote increments exactly one candidate count.

9.2 Edge Cases

- **No votes cast**: proportional returns `"No votes"`; other systems return their defined default outcomes.
- **Low turnout**: FPTP returns quorum failure even if a candidate leads.
- **IRV malformed ballots**: duplicates are not prevented on-chain; this may bias round allocation.

10 Security Analysis

10.1 Strengths

- **Threshold governance**: registration validation and election closure require a super-majority approval.
- **Clear authorization boundaries**: modifiers restrict registrar-only and admin-only actions.
- **Double-vote defenses**: enforced at both address and identity-hash levels.
- **Reduced reentrancy risk**: no external calls to untrusted contracts; functions follow checks-before-effects patterns.

10.2 Weaknesses and Practical Risks

10.2.1 On-Chain Privacy Leakage

Although the contract stores `hashedID`, voters provide `idCarteFourni` in plaintext when calling `castVote`. Observers can read it from transaction calldata, mempool data, or node traces. If identifiers have predictable formats, `keccak256(id)` is also vulnerable to offline dictionary attacks. Therefore, this design should not be used with real personally identifying information.

10.2.2 Ballot Secrecy vs. Auditability

The `VoteCast` event includes `choices`, making ballots publicly linkable to voter addresses. This is acceptable for demonstrations but violates secret ballot requirements.

10.2.3 Instant-Runoff Scalability

`_calculateInstantRunoff` iterates over `voterAddresses`. On-chain tallying cost grows with the number of voters:

$$O(V \cdot C \cdot R),$$

where V is the number of IRV voters, $C = 5$ candidates, and $R \leq 4$ rounds. Large V may exceed block gas limits.

10.2.4 Input Validation Gap (IRV Uniqueness)

Uniqueness of ranked candidates is enforced only in the frontend; a direct contract call can bypass it.

10.3 Recommended Hardening (Toward Production)

- Enforce IRV uniqueness on-chain (e.g., boolean `seen[6]`).
- Replace plaintext ID submission with commit-reveal using salt, or adopt ZK-based privacy.
- Reduce ballot disclosure in events (emit minimal data or store encrypted commitments).
- Define explicit tie-breaking rules for IRV elimination.
- Consider verifiable off-chain tallying for scalability (e.g., Merkleized ballots + proofs).

11 Frontend (dApp) Architecture

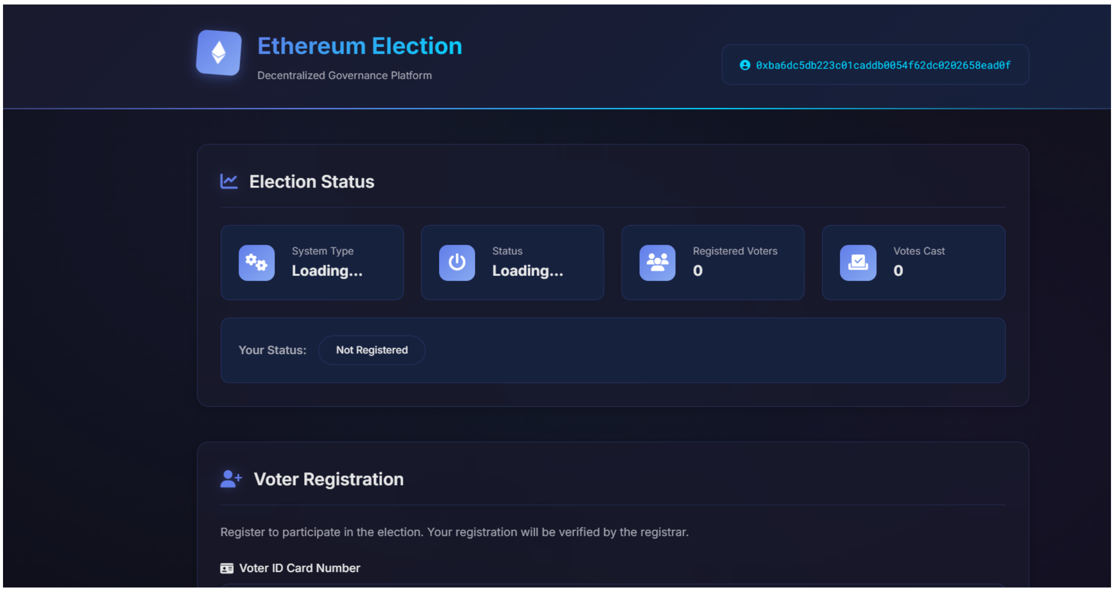

Figure 1: Website main interface

11.1 Technology Stack

- **HTML/CSS**: responsive UI layout and styling.
- **JavaScript (ES6+)**: UI logic and Web3 integration.
- **Web3.js**: contract calls and transactions via MetaMask.
- **Chart.js**: results visualization (pie chart for proportional, bar chart otherwise).

11.2 Key UI Components

- **Wallet Connection**: requests `eth_requestAccounts` and displays the connected address.
- **Status Dashboard**: reads `currentSystem`, `electionClosed`, `totalVotersRegistered`, `totalVotesCast`, and the voter record for the current account.
- **Registration Panel**: collects a voter ID in the UI; in the current implementation it does not submit an on-chain registration request, but informs users to wait for registrar validation.

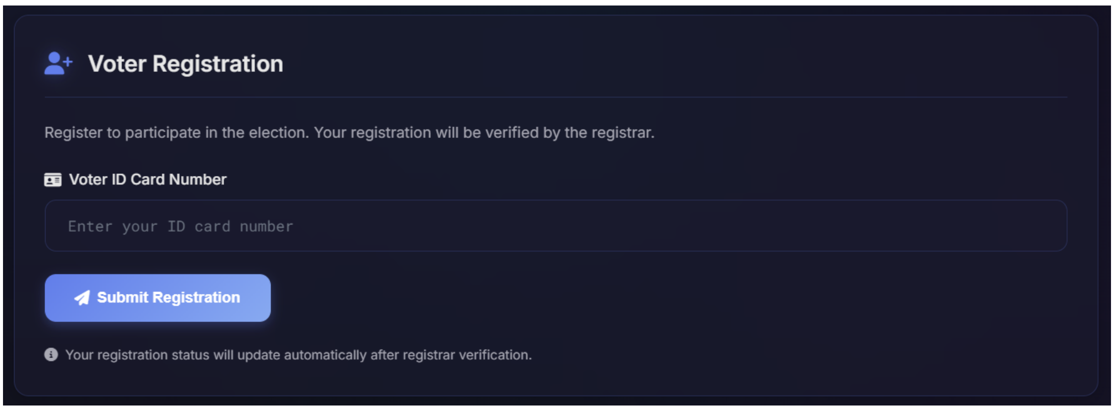

Figure 2: Registration Panel

- **Voting Panel**: displays either (i) radio selection for FPTP/Proportional or (ii) drag-anddrop ranking for IRV.

- **Admin/Registrar Panel**: shown conditionally based on `isAdmin` or `isRegistrar`.
- **Results View**: calls `getResults()` after closure and renders the chart.

11.3 Frontend–Contract Interaction Summary

Table 2: Frontend calls to smart contract

Feature	Contract Method	Notes
Load status	`currentSystem()`, `electionClosed()`, `totalVotersRegistered()`, `totalVotesCast()`	polled every 5s
Voter info	`voters(account)`	shows `isRegistered`, `validationCount`, `hasVoted`
Cast vote	`castVote(choices, id)`	sends transaction with a gas limit
Close election	`proposeAndCloseElection()`	admins only; disabled if already closed
Validate voter	`validateVoter(voterAddr, id)`	registrars only
Load results	`getResults()`	callable only after closure

11.4 Operational Note: Registration UX

The current `registerVoter()` frontend function does not write on-chain state; it only informs the user. In practice, registrars must input the voter address and ID in the registrar panel and submit `validateVoter`. A more robust UX would include an on-chain registration request event emitted by voters to notify registrars, without storing plaintext identifiers on-chain.

12 Deployment and Configuration Guide

12.1 Parameters

At deployment time, the following must be decided:

- Voting system: `FPTP_Quorum` / `Proportionnel` / `InstantRunoff`
- Admin set size N and addresses
- Registrar set size M and addresses

12.2 Local Development (Example: Ganache)

A typical local workflow:

- Start Ganache at `http://127.0.0.1:7545`.
- Deploy `Election` with constructor parameters via Remix or scripts.
- Export ABI into `abi.json`.
- Configure frontend constants:
  - `contractAddress`
  - `chainId` (Ganache commonly uses `1337` or a Ganache-specific ID)
  - provider endpoint (the current code uses `new Web3("http://127.0.0.1:7545")`)
- Use MetaMask accounts matching Ganache-funded keys.

Consistency note. If the frontend mentions Sepolia while using Ganache RPC, ensure `chainId` and RPC endpoint match the actual network.

13 Experimental Results

This section reports deterministic experimental scenarios derived directly from the on-chain logic and data model of the `Election` contract. All scenarios are constructed so that every step succeeds (registrations, voting, closure, and result retrieval).

13.1 Experimental Setup

Assumed environment. Local EVM development environment (Ganache at `http://127.0.0.1:7545`) with MetaMask accounts and the provided frontend dApp.

Fixed configuration (applied to all scenarios).

- Candidates: `NB_CANDIDATES = 5` (indices 1..5).
- Admin committee: $N = 4 \Rightarrow Q_N = \left\lfloor \frac{2N+2}{3} \right\rfloor = 3$ approvals required.
- Registrar committee: $M = 3 \Rightarrow Q_M = \left\lfloor \frac{2M+2}{3} \right\rfloor = 2$ validations required.
- Voters: 10 non-admin addresses are fully registered (each receives exactly 2 registrar validations).

13.2 Scenario A: FPTP_Quorum

Votes. All 10 registered voters cast one vote. Distribution:

$$\texttt{firstChoiceCounts[1..5]} = [1, 4, 2, 3, 0], \quad \texttt{totalVotesCast} = 10.$$

Quorum is satisfied since $10 > \lfloor 10/2 \rfloor = 5$. Winner is Candidate 2.

getResults():

`system="FPTP", winnerMsg="Winner: Candidate 2", finalScores=[0,1,4,2,3,0].`

13.3 Scenario B: Proportionnel

Votes.

$$\texttt{firstChoiceCounts[1..5]} = [2, 1, 3, 4, 0], \quad \texttt{totalVotesCast} = 10.$$

Percentages (integer floor): $[20, 10, 30, 40, 0]$.

getResults()

`system="Proportionnel", winnerMsg="Distribution of votes", finalScores=[20,10,30,40,0].`

13.4 Scenario C: InstantRunoff (IRV)

Ballots. 10 valid rankings (length 5, no duplicates):

- 4 voters: $[1, 2, 3, 4, 5]$
- 3 voters: $[2, 3, 4, 1, 5]$
- 2 voters: $[3, 4, 2, 1, 5]$
- 1 voter: $[4, 3, 2, 1, 5]$

Rounds (summary).

- Round 1: $(1:4, 2:3, 3:2, 4:1, 5:0)$ eliminate 5.
- Round 2: eliminate 4.
- Round 3: transfer from 4 to 3 $\Rightarrow (1:4, 2:3, 3:3)$ tie-minimum is 2 and 3, eliminate 2 by deterministic order.
- Round 4: transfer 2's ballots to 3 $\Rightarrow (1:4, 3:6)$; Candidate 3 wins (strict majority).

getResults():

`system="Instant-Runoff", winnerMsg="Winner: Candidate 3", finalScores=[0,4,0,6,0,0].`

13.5 End-to-End Success Summary

In all scenarios: (i) every voter becomes registered after 2 registrar validations, (ii) every vote is accepted, (iii) closure succeeds after 3 admin approvals, and (iv) `getResults()` returns the expected deterministic outputs.

14 Testing Checklist

14.1 Functional Tests

- Deploy with N admins and M registrars; verify thresholds match $\lceil 2N/3 \rceil$ and $\lceil 2M/3 \rceil$.
- Validate a voter with fewer than threshold registrar approvals; verify `isRegistered=false` and `validationCount` increments.
- Validate up to threshold; verify `isRegistered=true` and `totalVotersRegistered` increments exactly once.
- Cast vote as unregistered voter: should revert.
- Cast vote with wrong ID: should revert.
- Cast vote twice: should revert.
- Reuse same ID for another address: should revert via `idHasAlreadyVoted`.

14.2 Rule-Specific Tests

- FPTP quorum not met: verify message indicates quorum failure.
- Proportional: verify percentages sum to ≤ 100 due to integer flooring.
- IRV: verify elimination steps on a small deterministic set of ballots.

14.3 Governance Tests

- Attempt closure below threshold: election should remain open.
- Reach threshold: election closes, and `getResults()` becomes callable.
- Attempt vote after closure: should revert.

15 Conclusion

The `Election` contract demonstrates an end-to-end on-chain voting pipeline with role separation, super-majority governance, and multiple tally rules. It is appropriate for academic demonstrations and controlled environments where transparency and auditability are prioritized.

For real-world voting, the major gaps are privacy (plaintext identifiers and public ballot revelation), IRV ballot uniqueness enforcement on-chain, and scalability constraints for on-chain IRV tallying. Addressing these limitations would require stronger cryptographic protocols and/or verifiable off-chain computation.