U-Vote: Tests

This document provides a complete breakdown of the automated test suite for the U-Vote platform. Tests are distributed across three Python scripts covering infrastructure setup, database validation, and platform deployment. Test types include integration, security, network policy enforcement, performance, and infrastructure health checks.

Test Type Legend

Type	Description
Integration	Tests that multiple components work together end-to-end
Security	Tests that security controls (triggers, permissions) enforce their rules
Network Policy	Tests that Calico network policies allow/deny traffic as specified
Health / Smoke	Tests that services are alive and responding to HTTP requests
Performance	Tests database behaviour under load
Infrastructure	Tests that the Kubernetes platform components are healthy
Schema / Integrity	Tests that the database schema, constraints, and indexes are correct

---

Full Test Inventory

`test_db.py`

#	Type	Test Name	What It Tests	How It Works

1	Infrastructure	**Pod Running**	Verifies the PostgreSQL pod is in Running state and 1/1 containers are ready.	Runs `kubectl get pods` filtered by `app=postgresql` label and checks output for `Running` and `1/1`. Acts as a gate — all subsequent tests are skipped if this fails.
2	Integration	**Database Connection**	Confirms a psql client can connect to the database and authenticate.	Executes `SELECT version()` via `kubectl exec` inside the pod. A successful response containing `PostgreSQL` confirms both connectivity and

				authentication.
3	Schema / Integrity	**Tables Exist**	Verifies all 7 required tables are present in the database.	Runs the `\dt` psql meta-command and checks output for: `admins`, `elections`, `candidates`, `voters`, `voting_tokens`, `votes`, `audit_logs`.
4	Schema / Integrity	**Sample Data Loaded**	Checks that seed data was inserted into core tables during schema initialisation.	Runs `SELECT COUNT(*)` on `admins`, `elections`, and `candidates` tables. Each must return at least 1 row.
5	Security	**Vote Immutability**	Verifies that votes cannot be updated	Inserts a test vote,

			or deleted once cast — a core security requirement for ballot integrity.	then attempts an `UPDATE` and a `DELETE`. Both must be blocked by the `prevent_vote_update` and `prevent_vote_delete` triggers with the expected error message. If either succeeds, this is flagged as a security risk.
6	Security	**Hash Chain Generation**	Verifies that the SHA-256 hash chain trigger fires on every vote `INSERT`, linking each vote to the previous for audit trail integrity.	Inserts 3 test votes and queries `vote_hash` and `previous_hash` columns. Confirms hashes are non-NULL and populated by

				the `generate_vote_hash_trigger` using pgcrypto's `digest()` function.
7	Security	**User Permissions (Least Privilege)**	Validates that each service database role can only access the tables it is permitted to — a core zero-trust requirement.	Connects as `auth_service`: confirms it can `SELECT` from `admins` but is denied access to `votes`. Connects as `results_service`: confirms it can `SELECT` from `votes` but is denied `INSERT`. Permission denied errors are expected outcomes

				and treated as passes.
8	Integration	**Vote Tallying Query**	Validates that the schema supports the complex JOIN query used by the results microservice.	Executes a multi-table `JOIN` across `candidates` and `votes` with `COUNT` aggregation and percentage calculation, mirroring the exact query pattern the results service runs.
9	Schema / Integrity	**Performance Indexes**	Confirms that performance indexes exist on high-traffic columns.	Queries `pg_indexes` for `idx_votes_election`, `idx_votes_candidate`, and `idx_tokens_token`. Missing indexes would cause

				slow queries under load.
10	Schema / Integrity	**Foreign Key Constraints**	Verifies referential integrity constraints are in place across tables.	Queries `information_schema.table_constraints` for `FOREIGN KEY` entries. Ensures a vote must reference a valid election and candidate.
11	Integration	**Pod Network Connection**	Tests the full network path from a separate pod to the database, validating Kubernetes service routing and Calico network policy enforcement.	Spins up a temporary `postgres:15-alpine` pod labelled `app=auth-service` (satisfying Calico policy rules), waits for readiness, then connects to the `postgre`

				`sql` ClusterIP service using `PGPASSWORD` retrieved from the Kubernetes Secret. Cleans up the test pod after completion.
12	Performance	**Load Test (1000 votes)**	Stress tests the database and hash chain trigger under bulk insert load.	Uses PostgreSQL `generate_series` to insert 1000 votes in a single statement with randomised candidate distribution. Verifies the distribution query returns correct results after the bulk insert.

`setup_k8s_platform.py`

#	Type	Test Name	What It Tests	How It Works
13	Infrastructure	**Prerequisites Check**	Verifies all required CLI tools are installed before platform setup begins.	Runs `--version` commands for `docker`, `kubectl`, `kind`, and `helm`. Fails immediately if any tool is missing rather than failing midway through setup.
14	Infrastructure	**Cluster Node Verification**	Confirms all 3 Kind cluster nodes are in Ready state after setup completes.	Runs `kubectl get nodes` and checks for `Ready` status on all 3 nodes (1 control-plane, 2 workers).
15	Infrastructure	**Calico Pod Verification**	Confirms Calico CNI pods are running in the `calico-system` namespace.	Runs `kubectl get pods -n calico-`

				`system` and checks for `Running` status. Calico must be healthy before network policies can be enforced.
16	Infrastructure	**Namespace Verification**	Confirms the 3 U-Vote namespaces were created correctly.	Runs `kubectl get namespaces` and checks for `uvote-dev`, `uvote-test`, and `uvote-prod`. All three must exist for proper environment separation.

`deploy_platform.py`

#	Type	Test Name	What It Tests	How It Works

17	Network Policy	**Service-to-Database Policy Tests**	Validates that Calico network policies allow or deny database access exactly as specified. Some connections are expected to succeed, others are expected to fail.	For each deployed service, attempts a TCP connection to `postgresql:5432` from inside the service pod using `nc` or `bash /dev/tcp`. Backend services (auth, voting, election, results, admin) must connect. Frontend must be blocked. Both outcomes are verified — failures in either direction are flagged.
18	Integration	**DNS Resolution**	Verifies Kubernetes internal DNS is resolving service	Runs `nslookup`

			names correctly inside the cluster.	`postgresql` or `getent hosts postgresql` from inside a service pod. Confirms the ClusterIP service is reachable by DNS name.
19	Health / Smoke	**Health Endpoint Tests**	Confirms all 6 microservices are alive and responding to HTTP requests.	Uses `kubectl port-forward` to forward each service's container port to a free localhost port, then makes an HTTP GET to the `/health` endpoint (or `/` for the frontend). Expects HTTP 200 from all services.
20	Infrastructure	**Pod Health Verification**	Waits for all deployed pods to	Polls `kubectl`

			reach Running and Ready state after deployment and captures logs from any that fail.	`get pods -o json` watching for `phase=Running` and `ready=true` on all containers. Detects `CrashLoopBackOff` and `ImagePullBackOff` states. Captures the last 30 lines of logs from failing pods for diagnosis.

> **Note:** Tests must run in order. `test_db.py` requires the platform to be provisioned by `setup_k8s_platform.py` first. `deploy_platform.py` builds on both and validates the full application stack including all 6 microservices.