U-Vote: Microservice Architecture

System Overview

UVote is an online voting system built for small-scale elections (student councils, NGO boards, or similar organisations) where you need something more accountable than a Google Form but don't need the overhead of enterprise voting software.

The system is currently at MVP stage. The core approach is token-based: voters receive a one-time URL by email, click it, and vote. There's no voter account or password to manage. On the admin side, authentication uses JWT with bcrypt-hashed passwords.

Current stack:

- **Backend**: Python + FastAPI
- **Frontend**: Jinja2 (server-side rendered)
- **Database**: PostgreSQL 15
- **Deployment:** Kubernetes with Calico networking

What's implemented

- Token-based voting via one-time email URLs
- Admin login with password + JWT
- Bulk voter import via CSV
- One-vote-per-voter enforcement with anonymous ballots
- Email notification when results are ready
- WCAG AA compliant templates

Planned Implementation

- Immutable, hash-chained audit logs

User roles

- **Admins** create and manage elections, add candidates, import voters, and control when voting opens and closes.
- **Voters** receive an email with their unique voting link, cast their vote, and get a follow-up email when results are published.

Voting Flow Overview

The process runs in three phases:

Phase 1 - Setup

1. Admin registers an account (Auth Service)
2. Admin creates an election (Election Service)
3. Admin adds candidates (Admin Service)
4. Admin imports voters via CSV or manual entry (Admin Service)
5. Admin sends voting links tokens are generated and emails are dispatched (Admin Service + Email Service)

Phase 2 - Voting

6. Voter receives email with unique voting URL
7. Voter clicks the link; Voting Service validates the token and loads the ballot
8. Voter selects a candidate and submits - vote is recorded, token marked as used, confirmation shown
9. If the voter clicks the link again, they see "You already voted"

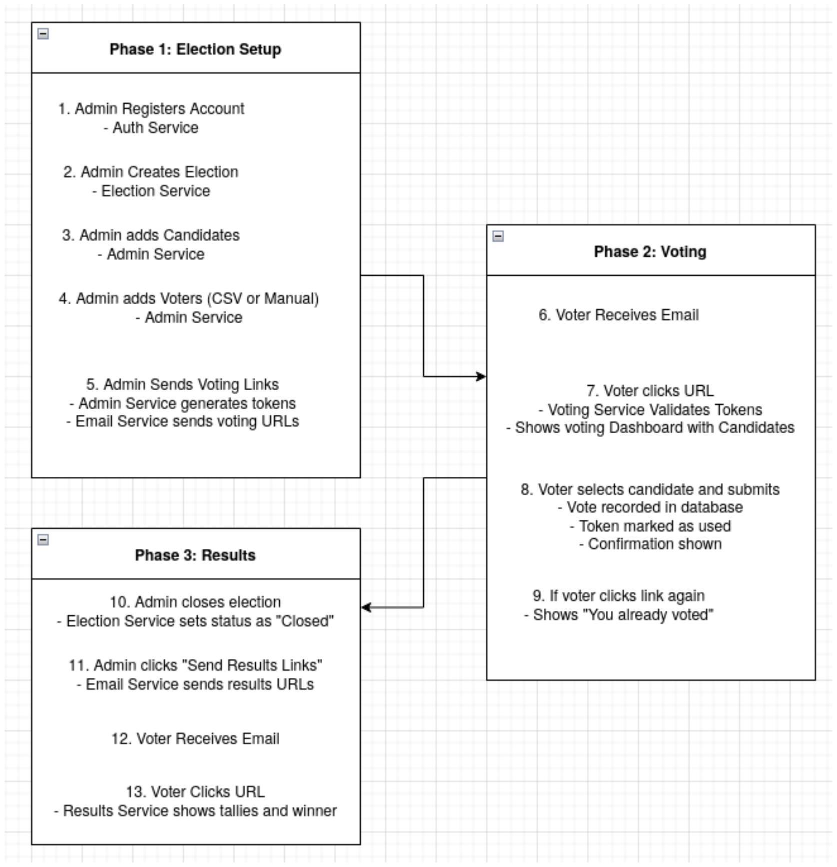
Phase 3 - Results

10. Admin closes the election (status set to "closed")

11. Admin triggers results emails (Admin Service → Email Service)

12. Voter receives results URL

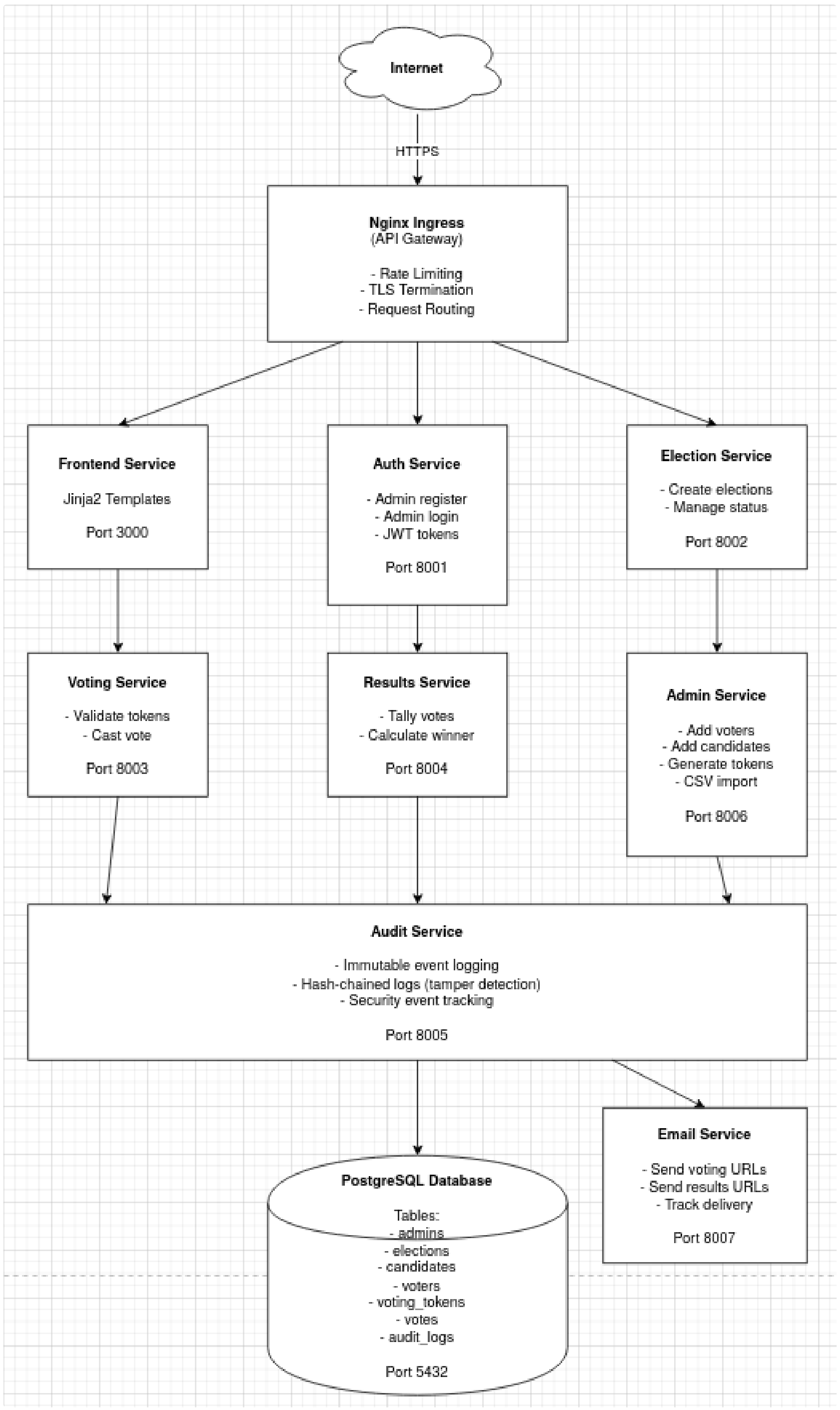
13. Voter views tallies and winner (Results Service)


Phase 1: Election Setup
1. Admin Registers Account - Auth Service
2. Admin Creates Election - Election Service
3. Admin adds Candidates - Admin Service
4. Admin adds Voters (CSV or Manual) - Admin Service
5. Admin Sends Voting Links - Admin Service generates tokens - Email Service sends voting URLs
Phase 2: Voting
6. Voter Receives Email
7. Voter clicks URL - Voting Service Validates Tokens - Shows voting Dashboard with Candidates
8. Voter selects candidate and submits - Vote recorded in database - Token marked as used - Confirmation shown
9. If voter clicks link again - Shows "You already voted"
Phase 3: Results
10. Admin closes election - Election Service sets status as "Closed"
11. Admin clicks "Send Results Links" - Email Service sends results URLs
12. Voter Receives Email
13. Voter Clicks URL - Results Service shows tallies and winner

---

Architecture Diagram

Service	**Port Number**	**Purpose**
Frontend	3000	Jinja2 User Interface
Auth	8001	Admin Authentication
Election	8002	Election Lifecycle
Voting	8003	Token validation & Vote Casting
Result	8004	Tallies & Winner Calculation
Audit	8005	Immutable Event Logging
Admin	8006	Voter/Candidate Management
Email	8007	Transactional Email Dispatch
PostgreSQL Database	5432	Persistent Storage



Service Descriptions

1. Nginx Ingress Controller (API Gateway)

Nginx acts as the single entry point for all client traffic. It handles TLS termination (HTTPS externally, plain HTTP internally), applies rate limiting to reduce DoS exposure, and routes requests to the appropriate service.

Purpose: Single entry point for all client requests; security boundary

Responsibilities:

- TLS termination (HTTPS → HTTP internally)
- Rate limiting (prevent DoS attacks)
- Route requests to appropriate services

Port: 80 (HTTP), 443 (HTTPS)

Routing Rules:

```
/ → Frontend Service (3000)
/api/auth → Auth Service (8001)
/api/elections → Election Service (8002)
/api/voting → Voting Service (8003)
/api/results → Results Service (8004)
/api/admin → Admin Service (8006)
```

---

2. Frontend Service

Serves the admin and voter UI using server-side rendered Jinja2 templates. The frontend makes API calls to backend services through the gateway rather than connecting directly.

Purpose: Serve user interface (HTML via Jinja2 templates)

Responsibilities:

- Render admin dashboard
- Display voting interface
- Show results pages
- WCAG AA accessibility compliance
- Client-side form validation
- Make API calls to backend services

Technology: Python FastAPI + Jinja2
Port: 3000
Dependencies: All backend services (via API Gateway)

Key Templates:

- `admin_register.html` - Admin registration form
- `admin_login.html` - Admin login
- `admin_dashboard.html` - Election management
- `vote.html` - Voting interface (accessed via token URL)
- `vote_confirmation.html` - "Vote recorded" page
- `results.html` - Election results display

Routes:

```
GET  /                          # Home page
GET  /admin/register            # Admin registration page
GET  /admin/login               # Admin login page
GET  /admin/dashboard           # Admin dashboard
GET  /vote?token={token}        # Voting page (token validated)
GET  /results/{election_id}     # Results page
```

---

3. Auth Service

Handles admin registration and login. On successful login it issues a JWT that the admin includes in subsequent requests; the gateway validates this token on each call.

Purpose: Admin authentication and authorization

Responsibilities:

- Admin registration (open registration)
- Admin login (email + password)
- JWT token issuance (24-hour expiration)
- Token validation

- Logout (token invalidation)

Technology: Python FastAPI
Port: 8001
Database Access: `auth_service` user (SELECT, INSERT, UPDATE on `admins`)

API Endpoints:

```
POST   /api/auth/register          # Register new admin
POST   /api/auth/login             # Admin login, returns JWT
POST   /api/auth/logout            # Invalidate JWT
GET    /api/auth/verify            # Verify JWT is valid
```

Authentication Flow:

```
1. Admin submits email + password
2. Auth Service validates credentials
3. Auth Service generates JWT token (24-hour expiration)
4. Admin uses JWT for all subsequent requests
5. API Gateway validates JWT on each request
```

Security Measures:

- Bcrypt password hashing (cost factor: 12)
- JWT signing with HS256 algorithm
- Account lockout after 5 failed attempts
- Rate limiting on login endpoint

JWT Token Structure:

```json
{
  "sub": "admin_42",
  "email": "admin@example.com",
  "role": "admin",
  "iat": 1707667200,
  "exp": 1707753600
}
```

---

4. Admin Service

Manages voters, candidates, and voting token distribution. Only accessible by authenticated admins

Purpose: Voter and candidate management (admin-only functions)

Responsibilities:

- Add voters (manual entry)
- Upload voters (CSV file)
- Remove voters
- Generate voting tokens
- Trigger Email Service to send voting URLs
- Trigger Email Service to send results URLs
- Add/remove/update candidates

Technology: Python FastAPI
Port: 8006
Database Access: `admin_service` user (SELECT, INSERT, UPDATE, DELETE on `voters`, `candidates`, `voting_tokens`)

API Endpoints:

```
# Voter Management
POST   /api/admin/voters                      # Add single voter
POST   /api/admin/voters/bulk                # CSV upload
DELETE /api/admin/voters/{id}                # Remove voter
GET    /api/admin/voters?election_id={id}    # List voters for
election

# Token Management
POST   /api/admin/elections/{id}/send-voting-links    # Generate
tokens, send emails
POST   /api/admin/elections/{id}/send-results-links   # Send results
emails
POST   /api/admin/voters/{id}/resend-token            # Resend
individual token

# Candidate Management
POST   /api/admin/candidates                 # Add candidate
DELETE /api/admin/candidates/{id}            # Remove candidate
PUT    /api/admin/candidates/{id}            # Update candidate
```

CSV Upload Format:

```
email,first_name,last_name
alice@example.com,Alice,Johnson
bob@example.com,Bob,Smith
carol@example.com,Carol,White
```

Token Generation Process:

When the admin clicks "Send Voting Links", the service iterates over all voters in the election:

1. Generates a cryptographic token using `secrets.token_urlsafe(32)` (43-character random string)
2. Stores the token in `voting_tokens` with a 7-day expiry
3. Calls the Email Service with the voter's address and token URL

Example token: `*x8K3mP9qL2wN7vR4tY6uI1oE5sA0dF8gH3jK9mN2pQ*`

Dependencies:

- Email Service (to send voting/results URLs)
- Audit Service (to log admin actions)

---

5. Email Service

Sends transactional emails and tracks delivery status. Failures are retried automatically.

Purpose: Send transactional emails

Responsibilities:

- Send voting invitation emails with token URLs
- Send results notification emails
- Track email delivery status
- Handle email failures with retry logic

Technology: Python FastAPI
Port: 8007
External Dependencies: SMTP server (Gmail, SendGrid, AWS SES)

API Endpoints:

```
POST   /api/email/send-voting-invitation      # Send voting URL
POST   /api/email/send-results-notification   # Send results URL
GET    /api/email/status/{id}                 # Check delivery status
```

Email Templates: are HTML with Jinja2-style variable substitution. The voting invitation includes the voter's name, election title, a "Vote Now" button linking to the token URL, and the expiry date. The results notification includes the voter's name, election title, and a "View Results" button.

Configuration:

```
SMTP_HOST = "smtp.gmail.com"
SMTP_PORT = 587
SMTP_USERNAME = "noreply@evote.com"
SMTP_PASSWORD = "<from-secret>"
SMTP_USE_TLS = True
```

---

6. Election Service

Creates and manages elections. The service owns the election lifecycle state machine.

Purpose: Manage elections (create, configure, control status)

Responsibilities:

- Create new elections
- Update election details (title, description, dates)
- Set election status (draft, active, closed)
- Control results visibility
- Delete elections (only if no votes cast)

Technology: Python FastAPI
Port: 8002
Database Access: `election_service` user (ALL on `elections`)

API Endpoints:

```
POST   /api/elections                  # Create election
GET    /api/elections                  # List all elections (for
admin)
GET    /api/elections/{id}             # Get election details
PUT    /api/elections/{id}             # Update election
DELETE /api/elections/{id}             # Delete election (if no votes)
POST   /api/elections/{id}/activate    # Set status to "active"
POST   /api/elections/{id}/close       # Set status to "closed"
```

Election Lifecycle:

```
Draft (created, not yet started)->(admin activates)
Active (voting period - voters can vote)->(admin closes)
Closed (voting ended - results available)
```

Business Rules:

- An election cannot be activated unless it has at least 2 candidates
- An election that was never activated cannot be closed directly
- Elections with existing votes cannot be deleted, they must be archived instead
- All status changes are logged to the Audit Service

---

7. Voting Service

Handles everything from the moment a voter clicks their link to the moment their vote is stored.

Purpose: Handle token-based vote casting

Responsibilities:

- Validate voting tokens from URL
- Check token not expired
- Check token not already used
- Display voting ballot

- Accept vote submission
- Mark token as used
- Record votes anonymously

Technology: Python FastAPI
Port: 8003
Database Access: `voting_service` user (INSERT on `votes`, SELECT on `elections` / `candidates`, UPDATE on `voting_tokens`)

API Endpoints:

```
GET    /api/voting/validate-token?token={token}    # Validate token, return election info
GET    /api/voting/ballot?token={token}            # Get candidates for election
POST   /api/voting/cast-vote                       # Submit vote
```

Voting Flow:

Step 1: Validate Token

```
GET /api/voting/validate-token?token=abc123

Response (if valid):
{
  "valid": true,
  "election_id": 5,
  "election_title": "Student Council 2026",
  "voter_email": "alice@example.com",
  "expires_at": "2026-02-18T23:59:59Z",
  "has_voted": false
}

Response (if already used):
{
  "valid": false,
  "error": "This voting link has already been used",
  "has_voted": true
}

Response (if expired):
{
  "valid": false,
  "error": "This voting link has expired"
}
```

Step 2: Get Ballot

```
GET /api/voting/ballot?token=abc123

Response:
{
  "election_id": 5,
  "election_title": "Student Council 2026",
  "candidates": [
    {
      "candidate_id": 1,
      "name": "Alice Johnson",
      "description": "Experienced leader...",
      "photo_url": "/uploads/alice.jpg"
    },
    {
      "candidate_id": 2,
      "name": "Bob Smith",
      "description": "Passionate advocate...",
      "photo_url": "/uploads/bob.jpg"
    }
  ]
}
```

Step 3: Cast Vote

```
POST /api/voting/cast-vote
Body: {
  "token": "abc123",
  "candidate_id": 2
}
```

The service re-validates the token (to guard against a race condition), checks the election is still active, verifies the candidate belongs to this election, inserts the vote, marks the token as used, and logs the event to the Audit Service. The candidate choice is **NOT** included in the audit log.

```
Response:
{
  "success": true,
  "message": "Your vote has been recorded",
  "vote_hash": "a3f8c9d2e1b4..."  # SHA-256 hash for verification
}
```

Vote Anonymity:

The `votes` table has no `voter_id` column. It stores only `election_id`, `candidate_id`, and a timestamp. Audit logs record that voter X cast a vote in election Y, not which candidate they chose. Once cast, it is not possible to trace a vote back to a specific voter.

- Votes table does NOT link back to voter
- Only links: election → candidate
- Audit logs record "voter X used token in election Y" (NOT which candidate)

Database Constraint:

```
-- Voting tokens table ensures one vote per voter per election
UNIQUE(voter_id, election_id)
```

---

8. Results Service

Read-only service that tallies votes and determines the winner.

Purpose: Calculate and display election results

Responsibilities:

- Tally votes per candidate
- Calculate percentages
- Determine winner(s)
- Generate results reports
- Display results (when election closed)

Technology: Python FastAPI
Port: 8004
Database Access: `results_service` user (SELECT only - read-only)

API Endpoints:

```
GET    /api/results/{election_id}       # Get election results
```

Results Calculation:

```
SELECT
    c.candidate_id,
    c.name,
    c.photo_url,
    COUNT(v.vote_id) as vote_count,
    ROUND(COUNT(v.vote_id)::numeric / NULLIF(total.total_votes, 0) * 100, 2) as percentage
FROM candidates c
LEFT JOIN votes v ON c.candidate_id = v.candidate_id
CROSS JOIN (
    SELECT COUNT(*) as total_votes
    FROM votes
    WHERE election_id = $1
) total
WHERE c.election_id = $1
GROUP BY c.candidate_id, c.name, c.photo_url, total.total_votes
```

```
ORDER BY vote_count DESC;
```

Response Format:

```json
{
  "election_id": 5,
  "election_title": "Student Council 2026",
  "status": "closed",
  "total_votes": 150,
  "results": [
    {
      "candidate_id": 2,
      "name": "Bob Smith",
      "photo_url": "/uploads/bob.jpg",
      "vote_count": 75,
      "percentage": 50.0,
      "is_winner": true
    },
    {
      "candidate_id": 1,
      "name": "Alice Johnson",
      "photo_url": "/uploads/alice.jpg",
      "vote_count": 50,
      "percentage": 33.33,
      "is_winner": false
    },
    {
      "candidate_id": 3,
      "name": "Carol White",
      "photo_url": "/uploads/carol.jpg",
      "vote_count": 25,
      "percentage": 16.67,
      "is_winner": false
    }
  ]
}
```

Business Rules:

- Results are only returned if election status is `closed`
- Admins can query results at any time regardless of status
- Result views are logged to the Audit Service

---

9. Database (PostgreSQL 15)

Port: 5432 (internal only , not exposed outside the cluster)
Storage: 5Gi PersistentVolume (Kubernetes)

Schema

```sql
-- Admins
CREATE TABLE admins (
    admin_id      SERIAL PRIMARY KEY,
    email         VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    created_at    TIMESTAMP DEFAULT NOW(),
    last_login    TIMESTAMP
);

-- Elections
CREATE TABLE elections (
    election_id  SERIAL PRIMARY KEY,
    admin_id     INT REFERENCES admins(admin_id),
    title        VARCHAR(255) NOT NULL,
    description  TEXT,
    status       VARCHAR(20) DEFAULT 'draft',
    created_at   TIMESTAMP DEFAULT NOW(),
    activated_at TIMESTAMP,
    closed_at    TIMESTAMP,
    CHECK (status IN ('draft', 'active', 'closed'))
);

-- Candidates
CREATE TABLE candidates (
    candidate_id  SERIAL PRIMARY KEY,
    election_id   INT REFERENCES elections(election_id) ON DELETE
CASCADE,
    name          VARCHAR(255) NOT NULL,
    description   TEXT,
    photo_url     VARCHAR(500),
    display_order INT,
    created_at    TIMESTAMP DEFAULT NOW()
);

-- Voters
CREATE TABLE voters (
    voter_id    SERIAL PRIMARY KEY,
    election_id INT REFERENCES elections(election_id) ON DELETE
CASCADE,
    email       VARCHAR(255) NOT NULL,
    first_name  VARCHAR(100),
    last_name   VARCHAR(100),
    created_at  TIMESTAMP DEFAULT NOW(),
    UNIQUE(election_id, email)
);

-- Voting tokens (one-time URLs)
CREATE TABLE voting_tokens (
    token_id    SERIAL PRIMARY KEY,
    token       VARCHAR(64) UNIQUE NOT NULL,
    voter_id    INT REFERENCES voters(voter_id) ON DELETE CASCADE,
    election_id INT REFERENCES elections(election_id) ON DELETE
CASCADE,
    is_used     BOOLEAN DEFAULT FALSE,
    created_at  TIMESTAMP DEFAULT NOW(),
    expires_at  TIMESTAMP NOT NULL,
    used_at     TIMESTAMP,
    UNIQUE(voter_id, election_id)
);

-- Votes (anonymous ,  no voter_id)
CREATE TABLE votes (
    vote_id       SERIAL PRIMARY KEY,
    election_id   INT REFERENCES elections(election_id),
    candidate_id  INT REFERENCES candidates(candidate_id),
    vote_hash     VARCHAR(64),
    previous_hash VARCHAR(64),
    cast_at       TIMESTAMP DEFAULT NOW()
);

-- Audit logs (append-only)
CREATE TABLE audit_logs (
    log_id        SERIAL PRIMARY KEY,
    timestamp     TIMESTAMP DEFAULT NOW(),
    event_type    VARCHAR(50) NOT NULL,
    user_id       INT,
    election_id   INT,
    details       TEXT,
    ip_address    VARCHAR(45),
    previous_hash VARCHAR(64),
    current_hash  VARCHAR(64)
);
```

Indexes

```
CREATE INDEX idx_votes_election   ON votes(election_id);
CREATE INDEX idx_votes_candidate  ON votes(candidate_id);
CREATE INDEX idx_tokens_token     ON voting_tokens(token);
CREATE INDEX idx_tokens_election  ON voting_tokens(election_id);
CREATE INDEX idx_audit_timestamp  ON audit_logs(timestamp);
CREATE INDEX idx_audit_event      ON audit_logs(event_type);
CREATE INDEX idx_elections_status ON elections(status);
```

Vote immutability

Two triggers lock the `votes` table against modification or deletion. Any attempt raises an exception at the database level, independent of application logic.

```
CREATE OR REPLACE FUNCTION prevent_vote_modification()
RETURNS TRIGGER AS $$
BEGIN
    RAISE EXCEPTION 'Votes cannot be modified or deleted';
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER prevent_vote_update
    BEFORE UPDATE ON votes FOR EACH ROW EXECUTE FUNCTION
prevent_vote_modification();

CREATE TRIGGER prevent_vote_delete
    BEFORE DELETE ON votes FOR EACH ROW EXECUTE FUNCTION
prevent_vote_modification();
```

Automatic vote hashing

On every INSERT into `votes`, a trigger computes a SHA-256 hash chaining the new vote to the previous one for that election. This happens at the database level so it cannot be bypassed by the application.

```sql
CREATE OR REPLACE FUNCTION generate_vote_hash()
RETURNS TRIGGER AS $$
DECLARE
    prev_hash VARCHAR(64);
BEGIN
    SELECT vote_hash INTO prev_hash
    FROM votes
    WHERE election_id = NEW.election_id
    ORDER BY cast_at DESC
    LIMIT 1;

    NEW.previous_hash := COALESCE(prev_hash, REPEAT('0', 64));

    NEW.vote_hash := encode(
        digest(
            NEW.election_id::text ||
            NEW.candidate_id::text ||
            NEW.cast_at::text ||
            NEW.previous_hash,
            'sha256'
        ),
        'hex'
    );

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER generate_vote_hash_trigger
    BEFORE INSERT ON votes FOR EACH ROW EXECUTE FUNCTION
generate_vote_hash();
```

Database users

Each service gets a dedicated user scoped to the minimum permissions it needs. Change all passwords before deploying to any non-local environment.

```sql
CREATE USER auth_service     WITH PASSWORD 'auth_pass_CHANGE_ME';
CREATE USER voting_service   WITH PASSWORD 'voting_pass_CHANGE_ME';
CREATE USER election_service WITH PASSWORD 'election_pass_CHANGE_ME';
CREATE USER results_service  WITH PASSWORD 'results_pass_CHANGE_ME';
CREATE USER audit_service    WITH PASSWORD 'audit_pass_CHANGE_ME';
CREATE USER admin_service    WITH PASSWORD 'admin_pass_CHANGE_ME';

-- Auth Service
GRANT SELECT, INSERT, UPDATE ON admins TO auth_service;
GRANT USAGE, SELECT ON SEQUENCE admins_admin_id_seq TO auth_service;

-- Voting Service
GRANT INSERT ON votes TO voting_service;
GRANT SELECT ON elections, candidates TO voting_service;
GRANT SELECT, UPDATE ON voting_tokens TO voting_service;
GRANT USAGE, SELECT ON SEQUENCE votes_vote_id_seq TO voting_service;

-- Election Service
GRANT SELECT, INSERT, UPDATE, DELETE ON elections TO election_service;
GRANT USAGE, SELECT ON SEQUENCE elections_election_id_seq TO
election_service;

-- Results Service (read-only)
GRANT SELECT ON votes, elections, candidates TO results_service;

-- Audit Service
GRANT INSERT, SELECT ON audit_logs TO audit_service;
GRANT USAGE, SELECT ON SEQUENCE audit_logs_log_id_seq TO
audit_service;

-- Admin Service
GRANT SELECT, INSERT, UPDATE, DELETE ON voters, candidates,
voting_tokens TO admin_service;
GRANT USAGE, SELECT ON SEQUENCE voters_voter_id_seq TO admin_service;
GRANT USAGE, SELECT ON SEQUENCE candidates_candidate_id_seq TO
admin_service;
GRANT USAGE, SELECT ON SEQUENCE voting_tokens_token_id_seq TO
admin_service;
```

Seed data (local dev / testing only)

```sql
INSERT INTO admins (email, password_hash) VALUES
```

```sql
('admin@uvote.com', '$2b$12$LQv3c1yqBWVHxkd0LHAkCOYz6TtxMQJqhN8/LewY5GyWVxKfF8.WO');
-- plaintext: admin123,  do not use in production

INSERT INTO elections (admin_id, title, description, status) VALUES
(1, 'Student Council 2026', 'Annual student council election', 'draft');

INSERT INTO candidates (election_id, name, description, display_order) VALUES
(1, 'Alice Johnson', 'Experienced leader focused on student welfare', 1),
(1, 'Bob Smith',     'Passionate about campus sustainability', 2),
(1, 'Carol White',   'Advocate for improved facilities', 3);
```

Security Measures

1. Admin Authentication

Passwords are hashed with bcrypt at cost factor 12. Login issues a JWT (HS256, 24-hour expiry). After 5 failed attempts the account is locked. The login endpoint is rate-limited.

- Password hashing with bcrypt (cost factor 12)
- JWT tokens (HS256, 24-hour expiration)
- Rate limiting on login endpoints
- Account lockout after 5 failed attempts

2. Voter Authentication

Voters authenticate via a one-time cryptographic token (`secrets.token_urlsafe(32)`). Tokens expire after 7 days and are invalidated immediately on use. No password is required.

- Cryptographic tokens (secrets.token_urlsafe(32))
- One-time use (token marked as used after vote)
- Token expiration (7 days)
- No password needed (reduces friction)

3. Vote Anonymity

The `votes` table has no `voter_id` column only `election_id`, `candidate_id`, and timestamp are stored. Audit logs confirm that a voter participated but do not record which candidate they chose. There is no query that can link a cast vote back to a specific voter.

- Votes table has NO voter_id column
- Only stores: election + candidate + timestamp
- Audit logs do NOT record candidate choice
- Impossible to trace vote to voter after casting

4. Data Integrity

Database triggers block any UPDATE or DELETE on the `votes` table. Votes and audit logs are both hash-chained, so any tampering is detectable. Foreign key and unique constraints are enforced at the database level.

- Vote immutability (database triggers prevent UPDATE/DELETE)
- Hash chaining (tamper detection)
- Database constraints (UNIQUE, FOREIGN KEY)
- Audit log verification

5. Network Security

Calico network policies default to deny-all. Services cannot reach each other directly — all inter-service traffic is routed via the API Gateway. The database port is not exposed outside the cluster.

- Calico network policies (default deny all)
- Services isolated (cannot directly access each other)
- Database not externally accessible
- All traffic through API Gateway

6. Input Validation

All inputs are validated with Pydantic models before reaching business logic. SQL queries use parameterized statements throughout. Token format is validated on each request. Request size limits are enforced at the gateway.

- Email format validation
- SQL injection prevention (parameterized queries)
- Request size limits
- Token format validation

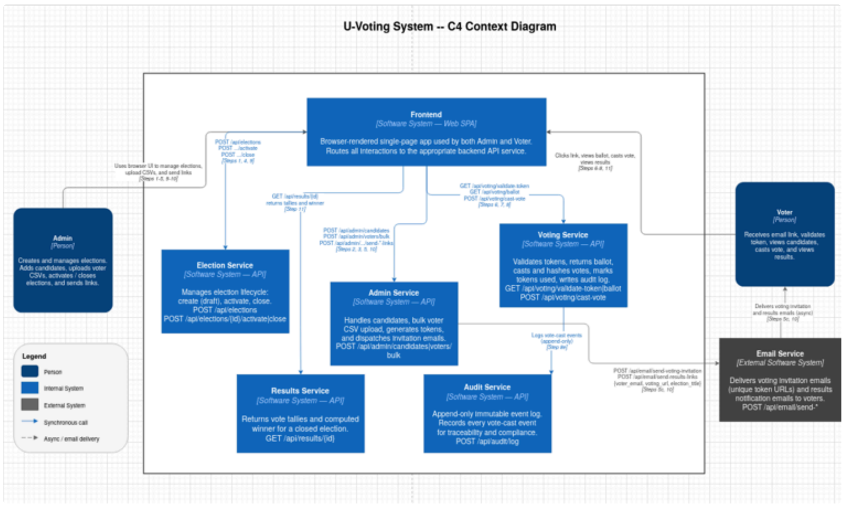
Data Flow Examples

Complete Voting Flow

1. Admin creates election
   Frontend → Election Service
   POST /api/elections
   {title: "Student Council 2026", description: "..."}
   → Creates election record (status='draft')
2. Admin adds candidates
   Frontend → Admin Service
   POST /api/admin/candidates
   {election_id: 5, name: "Alice Johnson", description: "..."}
   → Creates candidate record
3. Admin uploads voters CSV
   Frontend → Admin Service
   POST /api/admin/voters/bulk
   File: voters.csv
   → Creates voter records for each email
4. Admin activates election
   Frontend → Election Service
   POST /api/elections/5/activate
   → Updates election status='active'
5. Admin sends voting links
   Frontend → Admin Service
   POST /api/admin/elections/5/send-voting-links
   → For each voter:
     a. Generate token: "x8K3mP9qL2wN7vR4tY6uI1oE5sA0dF8gH3jK9mN2pQ"
     b. Store in voting_tokens table
     c. Call Email Service:
       POST /api/email/send-voting-invitation
       {
         voter_email: "alice@example.com",

    voting_url: "https://evote.com/vote?token=x8K3mP9q...",
    election_title: "Student Council 2026"
  }

6. Voter receives email and clicks link
   Browser → https://evote.com/vote?token=x8K3mP9q...
   Frontend → Voting Service
   GET /api/voting/validate-token?token=x8K3mP9q...
   → Returns election info if valid
7. Voter sees candidates
   Frontend → Voting Service
   GET /api/voting/ballot?token=x8K3mP9q...
   → Returns list of candidates
8. Voter clicks "Vote for Bob Smith"
   Frontend → Voting Service
   POST /api/voting/cast-vote
   {token: "x8K3mP9q...", candidate_id: 2}
   → Voting Service:
     a. Validates token again
     b. Checks election active
     c. Inserts vote (with hash)
     d. Marks token as used
     e. Logs to Audit Service
   → Returns confirmation
9. Admin closes election
   Frontend → Election Service
   POST /api/elections/5/close
   → Updates election status='closed'
10. Admin sends results links
    Frontend → Admin Service
    POST /api/admin/elections/5/send-results-links
    → Email Service sends results URLs to all voters
11. Voter views results
    Browser → https://evote.com/results/5
    Frontend → Results Service
    GET /api/results/5
    → Returns vote tallies and winner


U-Voting System -- C4 Context Diagram
Frontend
[Software System — Web SPA]
Browser-rendered single-page app used by both Admin and Voter. Routes all interactions to the appropriate backend API service.
Admin
[Person]
Creates and manages elections. Adds candidates, uploads voter CSVs, activates / closes elections, and sends links.
Uses browser UI to manage elections, upload CSVs, and send links [Steps 1-5, 9-10]
POST /api/elections POST .../activate POST .../close [Steps 1, 4, 9]
GET /api/results/{id} returns tallies and winner [Step 11]
POST /api/admin/candidates POST /api/admin/voters/bulk POST /api/admin/.../send-*-links [Steps 2, 3, 5, 10]
GET /api/voting/validate-token GET /api/voting/ballot POST /api/voting/cast-vote [Steps 6, 7, 8]
Clicks link, views ballot, casts vote, views results [Steps 6-8, 11]
Election Service
[Software System — API]
Manages election lifecycle: create (draft), activate, close. POST /api/elections POST /api/elections/{id}/activate|close
Admin Service
[Software System — API]
Handles candidates, bulk voter CSV upload, generates tokens, and dispatches invitation emails. POST /api/admin/candidates|voters/bulk
Voting Service
[Software System — API]
Validates tokens, returns ballot, casts and hashes votes, marks tokens used, writes audit log. GET /api/voting/validate-token|ballot POST /api/voting/cast-vote
Voter
[Person]
Receives email link, validates token, views candidates, casts vote, and views results.
Delivers voting invitation and results emails (async) [Steps 5c, 10]
Logs vote-cast events (append-only) [Step 8e]
POST /api/email/send-voting-invitation POST /api/email/send-results-links {voter_email, voting_url, election_title} [Steps 5c, 10]
Email Service
[External Software System]
Delivers voting invitation emails (unique token URLs) and results notification emails to voters. POST /api/email/send-*
Results Service
[Software System — API]
Returns vote tallies and computed winner for a closed election. GET /api/results/{id}
Audit Service
[Software System — API]
Append-only immutable event log. Records every vote-cast event for traceability and compliance. POST /api/audit/log
Legend
Person
Internal System
External System
Synchronous call
Async / email delivery

Future Enhancements

Potential Password Migration (Approach B)

If user feedback indicates voters want to verify their vote later, we can add password authentication:

Phase 1: Add Password Option (Hybrid)

```
1. Admin adds voter
2. Voter receives invite email
3. Voter clicks invite → OPTION: Set password or vote immediately
4. If password set: Future logins require password + OTP
5. If vote immediately: Same as current (token-based)
```

Phase 2: Full Password System

```
1. Admin adds voter → sends invite
2. Voter MUST set password
3. Voter logs in (password + OTP)
4. Voter votes (session-based, no token)
5. Voter can log in again to verify vote
```

Database Migration:

```sql
ALTER TABLE voters ADD COLUMN password_hash VARCHAR(255);
ALTER TABLE voters ADD COLUMN account_status VARCHAR(20) DEFAULT
'invited';

-- NULL password_hash = token-only (current system)
-- NOT NULL password_hash = password-based (new system)
```

Other Features

- [ ] Ranked-choice voting (STV algorithm)
- [ ] Multiple admins per election
- [ ] Candidate photos/bios upload
- [ ] Vote receipt (anonymized confirmation code)
- [ ] Results analytics dashboard
- [ ] Export results to CSV/PDF

Technical

- [ ] WebSocket for real-time updates
- [ ] Redis caching layer
- [ ] CDN for static assets
- [ ] Database read replicas
- [ ] Automated backups