

ADR-013: Domain-Driven Service Separation

Submitters

- Luke Doyle (D00255656)
- Hafsa Moin (D00256764)

Change Log

- approved 2026-02-14

Referenced Use Case(s)

- CVP-U-Vote Microservice Architecture

Context

Following the decision to adopt a microservices architecture (ADR-008), the next question was how to divide the system into services. The separation strategy determines each service's scope of responsibility, its database access requirements, its API surface area, and its failure boundaries. A poorly chosen strategy results in either services that are too coarse-grained, losing the benefits of microservices, or services that are too fine-grained, creating excessive inter-service communication overhead.

The approach taken here divides the system in two distinct ways, both informed by advice from the module's DevOps lecturer. The first split separates the system into two conceptually independent applications: a voter-facing voting application and an admin-facing authentication and management application. These represent genuinely different security domains with different user populations and different trust assumptions. This was for the main goal of de-coupling. The second split divides each of those applications further by functional domain, so that individual services such as the Results Service or the Voting Service can be deployed, scaled, and updated independently of one another.

Three separation strategies were researched before arriving at this approach: separation by technical layer, separation by business domain following Domain-Driven Design (DDD) bounded context principles, and separation by user role.

Proposed Design

The system is divided into eight services plus a database, each with a defined port, a dedicated PostgreSQL user, and its own Kubernetes Deployment, Service, and NetworkPolicy.

Services

The Nginx Ingress Controller acts as the single entry point for all client traffic. It handles TLS termination, applies rate limiting, and routes requests to the appropriate backend service. It is not a FastAPI service; it is the cluster's API gateway.

The Frontend Service (port 3000) serves the admin and voter interfaces using server-side rendered Jinja2 templates. It makes API calls to backend services through the gateway rather than connecting to them directly.

The Auth Service (port 8001) handles admin registration, login, JWT issuance, and token validation. Passwords are hashed with bcrypt at cost factor 12. JWTs use HS256 and expire after 24 hours. After five consecutive failed login attempts the account is locked. The service holds SELECT, INSERT, and UPDATE permissions on the `admins` table only.

The Election Service (port 8002) owns the election lifecycle. It creates elections, manages status transitions (draft to active to closed), and enforces the business rules that gate those transitions. It holds full permissions on the `elections` table only.

The Admin Service (port 8006) handles voter and candidate management. It processes manual voter entry and CSV bulk upload, generates one-time voting tokens using `secrets.token_urlsafe(32)`, and calls the Email Service to dispatch voting and results URLs. It holds SELECT, INSERT, UPDATE, and DELETE permissions on `voters`, `candidates`, and `voting_tokens`.

The Voting Service (port 8003) handles everything from the moment a voter follows their link to the moment their vote is stored. It validates the token from the URL, checks it has not expired and has not been used, displays the ballot, accepts the vote submission, marks the token as used, and logs the event to the Audit Service. The candidate choice is not included in the audit log entry. The service holds INSERT on `votes`, SELECT on `elections` and `candidates`, and SELECT plus UPDATE on `voting_tokens`.

The Results Service (port 8004) is read-only. It tallies votes per candidate, calculates percentages, and determines the winner once an election is closed. It holds SELECT on `votes`, `elections`, and `candidates` only.

The Audit Service (port 8005) provides append-only immutable event logging with hash-chained records for tamper detection. It holds INSERT and SELECT on `audit_logs`.

The Email Service (port 8007) sends transactional emails and tracks delivery status. Failures are retried automatically. It depends on an external SMTP server (Gmail, SendGrid, or AWS SES).

Database

PostgreSQL 15 runs on port 5432, which is not exposed outside the cluster. Storage is provisioned as a 5Gi PersistentVolume. Each service connects with a dedicated database user that holds only the permissions listed above. Vote immutability is enforced at the database level by two triggers that raise an exception on any UPDATE or DELETE to the `votes` table. A third trigger computes a SHA-256 hash on each INSERT into `votes`, chaining the new row to the previous vote in the same election. The same chaining pattern applies to `audit_logs`.

The `votes` table has no `voter_id` column. It stores only `election_id`, `candidate_id`, and a timestamp. There is no query that can link a cast vote back to a specific voter.

API Gateway routing

Path	Service	Port
/	Frontend	3000
/api/auth	Auth	8001
/api/elections	Election	8002
/api/voting	Voting	8003
/api/results	Results	8004
/api/admin	Admin	8006

Election lifecycle states

Draft (created, not yet active) → Active (voting open) → Closed (results available).

An election cannot move to Active unless it has at least two candidates. An election that was never activated cannot be closed directly. Elections that have received votes cannot be deleted; they must be archived.

Voting flow (end to end)

The admin creates an election, adds candidates, imports voters via CSV or manual entry, and activates the election. The Admin Service then generates a cryptographic token for each voter and calls the Email Service to send each voter their unique URL. The voter follows the link, the Voting Service validates the token, the voter selects a candidate and submits, and the vote is recorded. The token is immediately marked as used. When the admin closes the election, results URLs are sent to all voters. The Results Service computes tallies and determines the winner.

Considerations

Option 1: Technical layer separation

This approach separates services by technical function: an API gateway layer, a business logic layer, and a data access layer. It was rejected because it is an anti-pattern for microservices. Any feature change requires modifying all three layers simultaneously. There is no domain isolation, and per-domain security policies cannot be applied. This produces a distributed monolith: the complexity of microservices without the benefit of domain isolation.

Option 2: Domain-driven boundaries (chosen)

Separate services by business domain following DDD bounded context principles. Each service owns a complete business domain, changes are localised, and service boundaries map naturally onto security boundaries. The main drawbacks are that some concerns cut across domains (audit logging touches every service) and that voter management partially overlaps with election management. Both are managed by practical compromise: audit logging is handled by a shared Audit Service that all services call directly, and voter CRUD is assigned to the Election Service as part of election setup while the Auth Service handles voter authentication.

Option 3: User role separation

Separate services by user type into an admin service and a voter service. This was rejected because the admin service would become an internal monolith, fine-grained security policies could not be applied, and voting could not be scaled independently from admin functions. Two services is also too few to meaningfully demonstrate Kubernetes deployment patterns.

Cross-cutting concerns

The auth-service currently has the broadest scope, covering organiser authentication, voter token validation, MFA verification, and blind ballot token issuance. In a later iteration this could be split into an organiser-auth-service and a voter-identity-service, but the current scope is manageable within a two-semester timeline with a single developer.

Domain purity vs practicality

Strict DDD would require inter-service API calls for every cross-domain data access. Some services access tables from adjacent domains as a pragmatic compromise (for example, the auth-service reads the `voters` table for MFA). The alternative would add latency and complexity that is not justified at this scale.

Service count

Five core services is enough to demonstrate microservice patterns while remaining manageable for a single developer. The design could be expanded to eight or more services

in Stage 2 by adding dedicated audit, email, and admin services.

Decision

Domain-driven boundaries (Option 2) were chosen. Services are separated first by application domain (voting application vs authentication and management application, as recommended by the DevOps lecturer), then by functional domain within each application (Election Service, Voting Service, Results Service, and so on) to allow each to be scaled and updated independently.

The critical architectural boundary is between the Auth Service, which knows voter identity, and the Voting Service, which handles anonymous ballots. This boundary enforces vote anonymity at the code level, not just the policy level. The Auth Service has no access to encrypted ballots and the Voting Service has no access to voter identity information.

Security alignment follows from service boundaries directly. Each service's PostgreSQL user is scoped to the minimum tables and permissions that service requires (see ADR-014). Kubernetes NetworkPolicies and Calico default-deny rules follow the same boundaries (see ADR-010).

If results computation or voting load becomes a bottleneck, those services can be scaled or refactored without touching the rest of the system. The service boundaries will be reviewed at the end of Stage 2 (April 2026) based on development experience.

Other Related ADRs

- [ADR-008: Microservices Architecture](#) - Parent decision that this ADR implements
- [ADR-010: Zero-Trust Network Policies](#) - NetworkPolicies align with the service boundaries defined here
- [ADR-014: Per-Service Database Access](#) - Database users are scoped to the domains defined here
- [ADR-015: Anonymity Boundary](#) - The auth/voting separation defined here is what enforces the identity-ballot boundary

References

- CVP-U-Vote Microservice Architecture Document
- Eric Evans, Domain-Driven Design (Addison-Wesley, 2003)
- Investigation Log, section 4.2: Full analysis of separation strategies