ADR-008: Microservices Architecture

Submitters

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

Change Log

- approved 2026-02-12

Referenced Use Case(s)

- UVote System Architecture Requirements - Application architecture for the UVote election system, covering deployment strategy, fault isolation, and module learning outcomes for PROJ I8009.

Context

The UVote system requires a clear architectural decision between a monolithic application and a microservices architecture. This decision has cascading effects on deployment complexity, fault isolation, scaling strategy, and how well the project demonstrates the DevOps competencies required by the PROJ I8009 module.

Microservices was always the intended architecture for this project, driven by the module requirements. The PROJ I8009 brief requires "Build, test and deploy a substantial artefact while demonstrating best practice in modern DevOps" and "Demonstrate a thorough understanding of Development, Configuration Management, CI/CD and Operations including software tools for automation." Satisfying these requirements meaningfully demands multiple independently deployed services, which a single-container monolith cannot provide.

During early development, a basic single Flask application emerged organically as the codebase grew to include authentication, election management, voting, results, and audit logging. This was not a deliberate architectural choice but an accidental outcome of rapid prototyping. It served as a useful proof of concept, establishing what functionality and domain boundaries the system would need before the architecture was formalised. The monolith was not carried forward into production; it was replaced by the microservices design described in this ADR.

Proposed Design

Services defined:

Service	Port	Domain	Responsibility
auth-service	5001	Authentication	Admin auth, JWT, token validation, MFA, blind ballot issuance
election-service	5002	Elections	Election CRUD, lifecycle, voter management, token generation
voting-service	5003	Voting	Voter-facing web app, identity verification, ballot submission

results-service	5004	Results	Vote tallying, winner calculation, decryption
frontend-service	5000	Admin UI	Admin dashboard, election management interface
voter-service	5002	Voter Mgmt	Voter list management, CSV import (partially merged with election-service in current implementation)

Service structure:

Each service follows a consistent directory structure:

```
service-name/
├── app.py              # FastAPI application
├── Dockerfile          # Container image definition
├── requirements.txt    # Python dependencies
├── static/css/         # CSS stylesheets (if frontend-facing)
└── templates/          # Jinja2 templates (if frontend-facing)

```

Shared code is maintained in `shared/`:

```
shared/
├── database.py     # Async connection pool (asyncpg)
├── security.py     # Hashing, token generation, encryption
└── schemas.py      # Pydantic request/response models

```

Configuration:

- Communication pattern: synchronous HTTP via `httpx.AsyncClient`
- Service discovery: Kubernetes DNS (e.g., `http://auth-service:5001`)
- Shared library: copied into each Docker image at build time
- Health probes: `GET /health` on each service

Integration points:

- ADR-003 (Kubernetes): each service is a Kubernetes Deployment and Service resource
- ADR-010 (Zero-Trust): NetworkPolicies control inter-service communication
- ADR-013 (Domain Separation): domain-driven boundaries define service responsibilities
- ADR-014 (DB Users): each service uses a dedicated PostgreSQL user

Note on voter-service and election-service:

The voter-service and election-service functionality is partially merged in the current implementation. These may be fully separated or consolidated depending on development velocity and operational experience in Stage 2.

Considerations

Monolithic application (not selected as production architecture): A single FastAPI application containing all routes, business logic, and templates deployed as one container is the simplest possible structure. It requires no inter-service communication, has a single log stream, and is easier to debug in isolation. However, a monolith deployed as a single Kubernetes pod cannot demonstrate NetworkPolicies (there are no service boundaries to isolate), per-service database users (a single application user holds all privileges), independent scaling, or rolling updates across distinct components. It would not satisfy the module requirements for DevOps practice demonstration. The accidental monolith that emerged during early prototyping confirmed what the system would need to do, but was never a candidate for the final architecture.

Serverless functions (not selected): Deploying each endpoint as a serverless function via AWS Lambda or Azure Functions removes infrastructure management overhead and scales automatically. It was not selected because cold-start latency of up to 15 seconds is incompatible with the voter-facing experience, it introduces vendor lock-in, and it does not align with the Kubernetes learning objectives of the module.

Modular monolith (not selected): A single deployment with internal module boundaries provides cleaner structure than a flat monolith and can evolve into microservices later. It was not selected because it remains a single deployment unit and cannot demonstrate per-service NetworkPolicies, per-service database users, or independent rolling updates. The module assessment explicitly rewards DevOps practices that require multiple independently deployed services.

Inter-service latency: HTTP calls between services add approximately 5 to 10ms per call within the Kind cluster. This is mitigated by minimising cross-service calls and ensuring that most operations complete within a single service.

Shared library management: The `shared/` directory is copied into each Docker image at build time rather than published as a package. This is a pragmatic choice for a two-semester project and is a known limitation to revisit in Stage 2.

Single developer overhead: Managing six Dockerfiles, six Deployment manifests, and six Service manifests as a single developer is more work than a monolith. This is mitigated by a consistent structure across services and a deployment automation script.

Decision

A microservices architecture with six services was selected for UVote.

This was the intended approach from the outset of the project, driven by the PROJ I8009 module requirements. A monolith deployed as a single Kubernetes pod would technically satisfy the "containers" requirement but would not demonstrate service discovery, NetworkPolicies, rolling updates, independent scaling, or inter-service communication, all of which are expected by the module brief. The early accidental monolith was valuable as a proof of concept that clarified the domain boundaries and responsibilities each service would need, but it was never the target architecture.

Six services provides enough independently deployable components to demonstrate the full range of Kubernetes and DevOps practices while remaining manageable for a single developer over two semesters. Per-service database users (ADR-014) and per-service NetworkPolicies (ADR-010) are only possible with separate services, and both are required for the zero-trust security model.

The trade-off of higher operational complexity over a monolith is accepted. The additional complexity is the point: it is what enables the DevOps demonstration the module requires, and it produces a more realistic architecture for a system handling sensitive election data.

A review is scheduled for the end of Stage 2 (April 2026) to assess whether the service count is optimal or whether consolidation or expansion is warranted based on operational experience.

Other Related ADRs

- ADR-003: Kubernetes Platform - each service deployed as a Kubernetes Deployment and Service
- ADR-010: Zero-Trust Network Security - NetworkPolicies between services
- ADR-013: Domain-Driven Service Boundaries - rationale for how domains are divided
- ADR-014: Per-Service Database Users - least-privilege database access per service

References

- Martin Fowler - Microservices
- EdgeX Foundry ADR Template