Faculty of Engineering – Ain Shams University

Computer and Systems Engineering Department

zGate Gateway: A Zero Trust Database Access Proxy

Graduation Project Thesis

Supervisor:

Dr. Mohammed Sobh

Academic Year 2025–2026

Acknowledgments

We would like to thank...

Abstract

This project introduces a Zero Trust–based database access gateway (SecureDB Gateway)...

Contents

List of Figures

List of Tables

1. Team Information

1.1 Team Members

Name	ID	LinkedIn
Moustafa Ahmed	2100467	Moustafa Hashem
Kareem Ehab	2100913	Kareem Ehab
Hana Shamel	2100468	Hana Shamel
Karen Maurice	2100748	Karen Maurice
Michael George	2100709	Michael George
Mayar Walid	2100953	Mayar Walid
Rodina Mohammed	2100754	Rodina Mohammed

Table 1.1: Team Members Information

1.2 Roles & Responsibilities

2. Introduction & Problem Definition

2.1 Introduction

In the modern data-driven enterprise landscape, databases serve as the foundation for critical business operations, storing sensitive customer information, financial records, intellectual property, and operational data. However, the traditional approaches to database access management have not evolved at the same pace as the sophistication of cyber threats and the complexity of organizational structures. Development teams, database administrators, data analysts, and various other technical personnel routinely require direct database access to perform their duties, creating significant security challenges that existing solutions fail to adequately address.

zGate is a Zero Trust database access gateway designed to fundamentally transform how organizations manage, secure, and audit database access. Built on the principles of Zero Trust security architecture, zGate operates as an intelligent intermediary layer between users and database systems, enforcing identity-based access control, implementing dynamic query-level authorization, and ensuring complete auditability of all database operations. The system comprises three integrated components: a high-performance gateway server that intercepts and controls all database traffic, a command-line interface for streamlined user interactions, and a comprehensive web-based administration dashboard for policy management and monitoring.

By eliminating the need for developers and analysts to possess or handle production database credentials directly, zGate addresses the critical security gap that has led to numerous data breaches and compliance failures across industries. The system enforces the principle of least privilege through role-based access control policies, generates transient session-specific credentials, and provides real-time query filtering and data masking capabilities to protect sensitive information even when legitimate users are accessing the database.

2.2 Problem Statement

2.2.1 The Current State of Database Access Management

Contemporary organizations face a fundamental security dilemma in database access management. On one hand, operational efficiency demands that developers, data engineers, DevOps teams, and analysts have timely access to databases for development, troubleshooting, analytics, and maintenance activities. On the other hand, granting such access using traditional methods introduces severe security vulnerabilities that expose organizations to data breaches, insider threats, and regulatory non-compliance.

2.2.2 Critical Security Vulnerabilities

The prevailing practices in database access management present several critical vulnerabilities:

Static Credential Proliferation: Organizations typically rely on shared or long-lived static credentials that are distributed among team members. These credentials often appear in configuration files, environment variables, documentation, and even code repositories, creating numerous attack vectors. Once compromised, these credentials provide unrestricted access until manually rotated—a process that is infrequent and operationally disruptive.

Lack of Accountability and Auditability: When multiple users share the same database credentials, individual accountability becomes impossible. Security teams cannot determine which specific user executed a particular query, making post-incident investigation extremely difficult and enabling malicious insiders to operate with impunity. Traditional database audit logs capture the database username but not the actual human identity behind the action.

Excessive Privilege and Unrestricted Access: Developers and technical personnel are often granted broader database permissions than necessary for their specific tasks. A developer needing read-only access to a single table might receive full database access simply because granular permission management is too complex or time-consuming to implement properly. This violates the principle of least privilege and dramatically expands the attack surface.

Inadequate Protection of Sensitive Data: Even legitimate users with proper authorization may inadvertently expose sensitive information such as personally identifiable information (PII), financial data, or health records. Current systems lack the capability to dynamically mask or redact sensitive fields based on user identity and context, forcing organizations to choose between operational efficiency and data protection.

Insider Threat Vulnerability: Trusted insiders with legitimate database access

represent one of the most significant security risks. Whether through malicious intent, negligence, or social engineering, insiders can exfiltrate sensitive data, manipulate records, or cause operational disruptions with minimal detection risk under current access paradigms.

2.2.3 Impact and Consequences

These vulnerabilities have tangible consequences:

- **Data Breaches:** Compromised credentials or malicious insiders lead to unauthorized data exfiltration
- **Regulatory Non-Compliance:** Failure to meet requirements of GDPR, HIPAA, PCI-DSS, and other frameworks
- **Financial Losses:** Direct costs from breaches, regulatory fines, and operational disruptions
- **Reputational Damage:** Loss of customer trust and competitive disadvantage
- **Operational Inefficiency:** Cumbersome manual processes for credential management and access provisioning

2.2.4 The Need for Zero Trust Database Access

The transition to Zero Trust architecture in network and application security has demonstrated the effectiveness of "never trust, always verify" principles. However, database access has remained largely unchanged, still operating under implicit trust models. There is an urgent need for a solution that:

- Eliminates direct credential exposure by ensuring users never handle production database credentials
- Enforces identity-based access control tied to organizational identity management systems
- Implements dynamic, session-specific authorization rather than static permissions
- Provides query-level policy enforcement to control what operations each user can perform
- Ensures complete auditability with full traceability of every database operation to individual users

- Supports data-level security through dynamic masking and filtering of sensitive information
- Maintains operational efficiency without imposing excessive burden on legitimate users

2.3 Gaps In Existing Solutions

Organizations have historically relied on several security approaches to protect database access, each with significant limitations that fail to address the core vulnerabilities outlined previously.

2.3.1 Perimeter-Based Security

Traditional perimeter security operates on the assumption that threats exist outside the network boundary while everything inside is trustworthy. Firewalls, network segmentation, and IP whitelisting control which systems can reach database servers.

Limitations:

- **Lateral Movement:** Once an attacker breaches the perimeter (through phishing, compromised endpoints, or insider access), they can move freely within the network and access databases directly
- **No Identity Verification:** Perimeter controls verify network location, not user identity. Anyone on an authorized network or VPN can access databases
- **Coarse-Grained:** Controls apply at the network level, not at the query or data level. A user with network access has unrestricted database access
- **Cloud Incompatibility:** Modern cloud architectures and remote work arrangements render network perimeters increasingly porous and difficult to define

2.3.2 VPN-Based Access

Virtual Private Networks extend the corporate network to remote users, creating an encrypted tunnel that makes remote devices appear as if they're on the internal network.

Limitations:

- **All-or-Nothing Access:** VPN grants network-level access to all resources within its scope. A user connected via VPN can potentially access any database on that network segment

- **Shared Credentials Still Required:** VPN only solves the network connectivity problem; users still need database credentials, perpetuating the static credential problem
- **No Query-Level Control:** VPN cannot inspect, filter, or control database queries based on content or context
- **Session Persistence:** VPN sessions often remain active for extended periods, providing prolonged access windows for potential compromise
- **No Audit Trail:** VPN logs show connection events but provide no visibility into actual database operations performed

2.3.3 Bastion Hosts and Jump Servers

Organizations deploy intermediate servers that users must connect to before accessing databases, providing a centralized access point and audit logging.

Limitations:

- **Credential Exposure:** Users still retrieve and use actual database credentials, even if through a bastion host
- **Limited Policy Enforcement:** Bastion hosts log connections but typically cannot enforce query-level policies or filter sensitive data
- **Operational Overhead:** Requires maintaining additional infrastructure and managing access to the bastion itself
- **Session Recording Limitations:** While some bastion solutions record sessions, they provide after-the-fact forensics rather than real-time policy enforcement
- **Circumvention Risk:** Technical users can potentially bypass bastion hosts if they obtain credentials through other means

2.3.4 Database Native Access Controls

Modern databases include built-in authentication, authorization, and audit logging capabilities.

Limitations:

- **Complex Management:** Managing granular permissions across multiple databases and numerous users becomes administratively prohibitive at scale
- **Static Permissions:** Database roles and privileges are typically static and don't adapt to context (time, location, purpose)

- **Shared Account Pattern:** The complexity of per-user account management often leads organizations to share credentials anyway
- **Limited Masking Capabilities:** While some databases support row-level security and column masking, these features are database-specific, complex to configure, and inflexible
- **No Centralized Policy:** Each database system has its own permission model, preventing consistent policy enforcement across heterogeneous environments

2.3.5 Privileged Access Management (PAM) Solutions

PAM systems manage and audit privileged account credentials, often providing password vaulting, session recording, and credential rotation.

Limitations:

- **Still Credential-Based:** PAM distributes credentials to users, even if temporarily. Users still handle and potentially misuse actual database passwords
- **Session-Level, Not Query-Level:** PAM typically operates at the session level, recording entire sessions but not enforcing policies on individual queries
- **Limited Data Protection:** PAM cannot dynamically mask sensitive data fields based on user identity or query context
- **Operational Friction:** The check-out/check-in process for credentials adds significant overhead to developer workflows
- **PostgreSQL/MySQL Limitations:** Many PAM solutions were designed for privileged OS access and offer limited database protocol support

2.3.6 Database Activity Monitoring (DAM)

DAM solutions monitor and alert on database activity by analyzing network traffic or database logs.

Limitations:

- **Reactive, Not Preventive:** DAM detects suspicious activity after it occurs rather than preventing unauthorized actions proactively
- **No Access Control:** DAM cannot prevent users from executing queries; it only observes and reports
- **Alert Fatigue:** Organizations receive numerous alerts but lack the ability to block malicious activity in real-time

- **Identity Blindness:** DAM sees database usernames but often cannot tie actions to actual human identities when credentials are shared

2.3.7 The Fundamental Gap

All existing approaches share a common fundamental flaw: they separate authentication/authorization from the actual data access point.

Users authenticate to VPNs, bastion hosts, or PAM systems, but ultimately receive raw database credentials and connect directly to databases. This creates an uncontrolled gap where policy enforcement, audit logging, and data protection cannot be reliably applied.

Additionally, none of these solutions adequately address the credential exposure problem. Whether stored in password vaults, configuration files, or manually entered, database credentials exist outside the security boundary and can be extracted, shared, or misused by authorized users.

2.4 Why Zero Trust for Databases is Different

Zero Trust database access represents a paradigm shift that fundamentally reimagines how database security should operate. Rather than attempting to secure the perimeter or audit after the fact, Zero Trust embeds security directly into the data access path itself.

2.4.1 Core Zero Trust Principles Applied to Databases

Never Trust, Always Verify: Every database access request is authenticated and authorized in real-time, regardless of network location or previous access history. There is no concept of "trusted internal network."

Least Privilege Access: Users receive the minimum necessary permissions for their specific task at a specific moment. Access rights are dynamically evaluated based on identity, role, context, and policy.

Assume Breach: The architecture assumes that credentials may be compromised and that internal threats exist. Therefore, every query is inspected and controlled, and no user ever possesses credentials that could be misused outside the controlled gateway.

2.4.2 The Zero Trust Database Gateway Architecture

Unlike traditional solutions that operate adjacent to database access, a Zero Trust gateway like zGate becomes the exclusive access point for all database operations. This architectural position enables capabilities impossible with peripheral solutions.

Identity-Based Authentication: Users authenticate using their organizational identity (JWT tokens, SSO integration) rather than database credentials. Authentication is tied to the specific human or service, eliminating shared accounts and enabling true accountability.

Credential Abstraction: The gateway maintains actual database credentials internally. Users never see, handle, or transmit production database passwords. Even if a user's authentication token is compromised, the attacker gains no direct database access—they must still pass through the gateway's policy enforcement.

Protocol-Aware Interception: By implementing native database protocols (MySQL, PostgreSQL, MSSQL, etc.), the gateway can parse and inspect every query at the SQL level. This enables surgical policy enforcement that perimeter tools cannot achieve:

- Block specific SQL commands (DROP, DELETE) based on user role
- Restrict queries to specific tables or schemas
- Prevent unauthorized joins or subqueries
- Control result set size and query execution time

Session-Specific Access: Each connection through the gateway represents a distinct, auditable session tied to a specific user identity. Sessions are short-lived, context-aware, and can be terminated immediately if suspicious activity is detected.

Dynamic Policy Enforcement: Policies are evaluated in real-time for every query based on:

- User identity and assigned roles
- Target database and table
- Query type and structure
- Time of day, day of week
- Historical behavior patterns
- Data classification and sensitivity

2.4.3 Query-Level Data Protection

Zero Trust database access enables data protection at the query level, something impossible with network-based or credential-based solutions:

Dynamic Data Masking: Sensitive fields (credit card numbers, social security numbers, personal health information) are automatically masked or redacted based on

the requesting user's clearance level. A developer sees masked data while an authorized analyst sees plaintext—from the same query.

Row-Level Filtering: The gateway can inject WHERE clauses or modify queries to restrict which rows a user can access, enforcing data boundaries without requiring database-native row-level security configuration.

Column-Level Restrictions: Certain columns can be completely hidden from specific roles, preventing even the awareness of sensitive data's existence.

2.4.4 Complete Auditability and Traceability

Zero Trust database access provides audit logging that captures not just that an action occurred, but the complete context:

- **Who:** Actual human or service identity, not just database username
- **What:** Exact SQL query executed, including results and data accessed
- **When:** Precise timestamp with session context
- **Where:** Source location, network details, client application
- **Why:** Request context, approval workflows if applicable
- **Outcome:** Success, failure, policy denials, data returned

This audit trail is immutable, centralized, and sufficient for forensic investigation, compliance reporting, and threat detection.

2.4.5 What zGate Implements

The zGate architecture embodies these Zero Trust principles through its three-component design:

Gateway Server: Implements protocol handlers for MySQL and MSSQL, intercepting all database traffic at the wire protocol level. The gateway's policy engine evaluates every query against configured rules, the dispatcher manages connection routing, and session managers track user activity in real-time. Users connect to zGate using standard database clients, but the gateway mediates all communication with backend databases.

Command-Line Interface: Provides developers and analysts with a streamlined authentication flow. Users authenticate with their organizational identity, receive time-limited JWT tokens, and establish database sessions without ever handling production credentials. The CLI manages token storage and renewal transparently.

Web Administration Dashboard: Enables security teams to define role-based access control policies, configure database connections, manage user permissions, and

monitor active sessions and query logs. Administrators visualize access patterns, audit historical activity, and respond to security events through a comprehensive management interface.

Figure 2.1: what zGate offers

2.4.6 Operational Advantages

Beyond security improvements, Zero Trust database access delivers operational benefits:

- **Faster Onboarding:** New developers gain database access through role assignment in minutes, not days of credential provisioning
- **Reduced Credential Rotation Burden:** Database passwords change infrequently since users never access them
- **Simplified Compliance:** Centralized policy enforcement and comprehensive audit logs satisfy regulatory requirements
- **Cross-Database Consistency:** Single policy framework applies uniformly across MySQL, PostgreSQL, MSSQL, and other database types
- **Developer Experience:** Legitimate users experience minimal friction—authentication is transparent and access is granted immediately upon authorization

2.4.7 Addressing the Gaps

Zero Trust database access directly addresses every gap in existing solutions:

Limitation	**Zero Trust Solution**
Perimeter breach enables full access	Gateway enforces identity verification regardless of network position
VPN grants network-wide access	Access is scoped per-database, per-session, per-query
Bastion hosts still expose credentials	Users never possess or see database credentials
Static database permissions	Dynamic policy evaluation per query
PAM credentials can be misused	Tokens are gateway-specific and cannot directly access databases
DAM is reactive only	Real-time policy enforcement prevents unauthorized queries
Shared credentials prevent accountability	Every action is tied to individual user identity

Table 2.1: Comparison of Traditional Limitations vs. Zero Trust Solutions

3. Project Definition

3.1 Project Definition and Scope

zGate is a comprehensive Zero Trust database access gateway platform designed to eliminate credential exposure, enforce identity-based access control, and provide complete auditability for database operations in enterprise environments. The project encompasses the design, implementation, and deployment of a three-tier architecture that intercepts, authenticates, authorizes, and audits all database access in real-time.

3.1.1 System Components

The platform consists of three integrated components working in concert:

Gateway Server (zGate Core)

- **Protocol Handlers:** Native implementation of MySQL and MSSQL wire protocols, enabling transparent protocol-level interception and inspection of database traffic
- **Authentication System:** JWT-based authentication with configurable token expiration, eliminating the need for users to possess database credentials
- **Policy Engine:** Real-time policy evaluation engine that performs fresh permission lookups for every database operation, supporting role-based and custom permission models
- **Dynamic Proxy Manager:** On-demand creation and management of database proxies with automatic port allocation, eliminating static listener configurations
- **Temporary Credential System:** Automated generation of session-specific database users with unique, cryptographically secure credentials that are automatically purged upon session termination
- **Session Management:** Per-user, per-database session tracking with context preservation and graceful cleanup mechanisms

- **API Server:** RESTful API exposing authentication, database listing, connection management, and session control endpoints

Command-Line Interface (zGate-CLI)

- **Secure Authentication Flow:** Interactive login process with secure credential input and token management
- **Token Storage & Management:** Cross-platform secure token storage using OS keyring with encrypted file fallback, automatic token refresh before expiration
- **Database Discovery:** List all databases accessible to the authenticated user based on their assigned roles
- **Connection Management:** Establish database connections through the gateway with automatic credential handling
- **Session Status:** Real-time visibility into active sessions, token expiration, and authentication state
- **Logout & Cleanup:** Proper session termination with server-side revocation and local credential cleanup

Web Administration Dashboard (zGate-WebUI)

Built with Next.js 16, React 19, and Radix UI, the dashboard provides comprehensive management capabilities:

- **Overview Dashboard:** Real-time system metrics, active sessions, and access patterns visualization
- **Database Management:** Configure database connections, credentials, and connection pooling parameters
- **User Management:** Create, modify, and deactivate user accounts with role assignments
- **Administrator Management:** Dedicated administrative user management with elevated privileges
- **Access Control:** Define and manage roles with granular permission specifications including database scoping and privilege levels (read-only, read-write, admin)
- **Active Sessions Monitoring:** View all active database sessions with user context, connection duration, and query activity

- **Query Execution:** Direct query interface with policy enforcement and audit logging
- **Shared Account Pools:** Manage reusable database account credentials for backend systems
- **Per-User Database Accounts:** Personal database account management for individual user needs
- **Activity Audit:** Comprehensive audit log viewer with filtering, search, and export capabilities
- **Connected Databases:** Real-time status of database connectivity and health

3.1.2 Scope Boundaries

In Scope:

- MySQL, Postgres and MSSQL protocol support with complete authentication and query interception
- Role-based access control with hierarchical permission models
- Temporary database user lifecycle management
- JWT-based authentication with automatic token refresh
- Dynamic proxy allocation and connection pooling
- Comprehensive audit logging of all database operations
- RESTful API for programmatic access
- Cross-platform CLI with secure credential storage
- Web-based administration interface
- Configuration-driven deployment (YAML-based)
- Session-level access control and monitoring

Out of Scope (Current Phase):

- Oracle, and other database protocol support (planned for future phases)
- Real-time query analysis and threat detection algorithms
- Integration with external identity providers (SAML, LDAP, Active Directory)

- Multi-factor authentication (MFA) enforcement
- Database activity replay and forensic analysis tools
- High-availability and clustering capabilities
- Advanced query rewriting and optimization
- Automated compliance reporting generation

3.2 Objectives

The zGate project aims to achieve the following measurable technical and security objectives:

3.2.1 Core Security Objectives

Eliminate Direct Credential Exposure

- Implement JWT-based authentication system where users never handle production database passwords
- Develop automatic temporary database user creation with session-specific, cryptographically secure credentials
- Ensure temporary credentials are purged immediately upon session termination, preventing credential accumulation
- Achieve zero instances of production credentials appearing in user configuration files, environment variables, or CLI history

Enforce Identity-Based Access Control

- Build a role-based access control (RBAC) engine supporting hierarchical roles and custom permissions
- Implement real-time permission evaluation that reflects role changes immediately without requiring user re-authentication
- Support per-database, per-user authorization with multiple privilege levels (read-only, read-write, admin)
- Enable policy decisions based on fresh configuration lookups rather than stale token claims

Implement Protocol-Level Interception

- Develop native protocol handlers for MySQL and MSSQL wire protocols
- Enable transparent database proxy functionality allowing standard client tools (MySQL Workbench, SSMS) to function without modification
- Intercept and route all database traffic through the gateway with sub-100ms latency overhead for typical operations
- Maintain protocol compatibility ensuring 100% of standard database operations work through the proxy

Provide Comprehensive Audit Logging

- Log all authentication attempts with user identity, timestamp, success/failure status, and failure reasons
- Record complete session lifecycle: establishment, duration, database target, and termination circumstances
- Capture connection metadata sufficient for forensic investigation and compliance verification
- Generate structured logs compatible with standard log aggregation and SIEM tools

3.2.2 Operational Objectives

Deliver Multi-Component Integration

- Build RESTful API server exposing authentication, database listing, connection management, and session control
- Develop cross-platform CLI with secure token storage using OS keyring and automatic token refresh
- Create responsive web administration dashboard supporting all management functions
- Ensure seamless data flow between all three components (Gateway, CLI, WebUI)

Enable Centralized Administration

- Provide web-based interface for managing users, roles, database connections, and access policies

- Support real-time monitoring of active sessions with ability to view connection details and terminate sessions
- Allow administrators to modify permissions and configurations without system restarts
- Implement user and admin account separation with appropriate privilege boundaries

Support Multi-Database Environments

- Design modular architecture allowing support for MySQL and MSSQL through unified interfaces
- Enable addition of new database protocols through implementation of standard handler and manager interfaces
- Maintain per-database configuration for credentials, connection parameters, and access policies
- Support simultaneous connections to multiple database instances of different types

Ensure Production-Ready Stability

- Implement graceful error handling, connection cleanup, and resource management
- Support concurrent sessions from multiple users without resource conflicts
- Provide configuration validation and startup checks to prevent misconfiguration
- Enable zero-downtime configuration reloads for non-breaking changes

3.2.3 Academic and Technical Learning Objectives

Demonstrate Applied Security Architecture

- Apply Zero Trust principles specifically to database access management
- Implement secure authentication flows with proper token lifecycle management
- Design authorization systems with clear separation between authentication, policy evaluation, and enforcement
- Practice secure coding including input validation, SQL injection prevention, and secure credential generation

Develop Full-Stack System Integration Skills

- Build high-performance network services in Go with concurrent connection handling
- Implement database wire protocol parsers and connection proxies
- Create modern React-based web interfaces with real-time data updates
- Develop CLI applications with cross-platform compatibility and user experience focus
- Design and document RESTful APIs for inter-component communication

4. Requirements Engineering

4.1 Functional Requirements

4.1.1 FR-1: User Authentication & Authorization

- User login with bcrypt password hashing
- JWT token management (access & refresh tokens)
- Role-based access control (RBAC)
- Custom user permissions

4.1.2 FR-2: Database Connection Management

- Multi-database support (MSSQL, MySQL)
- Dynamic proxy creation with auto port allocation
- Temporary database users with auto-cleanup
- Secure credential generation

4.1.3 FR-3: Policy Engine

- Real-time access control validation
- Permission-based database filtering
- Multi-level permissions (read, write, admin)

4.1.4 FR-4: Command Line Interface

- Auto token refresh
- Database listing and connection
- Session status monitoring

4.1.5 FR-5: Web User Interface

- Database configuration management
- User and role management
- Admin account management
- Active session monitoring
- Shared account pools
- Personal database accounts
- Comprehensive audit trail

4.1.6 FR-6: Protocol Handling

- MSSQL, MySQL and Postgres protocol support with temp user creation

4.1.7 FR-7: Audit & Logging

- Security event logging
- User context in all logs
- Structured logging with configurable levels

4.2 Non-Functional Requirements

4.2.1 NFR-1: Security

- Zero trust architecture principles
- Bcrypt password hashing
- Encrypted connections
- Token-based session security

Key Metrics:

- Access token expiry: 15 minutes
- Refresh token expiry: 7 days
- Session isolation with temporary users

4.2.2 NFR-2: Performance

- Authentication: < 500ms
- Database listing: < 200ms
- Connection establishment: < 2 seconds
- WebUI page loads: < 1 second

4.2.3 NFR-3: Scalability Targets

- 100 concurrent sessions
- 1000 auth requests/minute
- Memory: < 512MB at normal load

4.2.4 NFR-4: Reliability

- 99.5% uptime during business hours
- Graceful error handling
- Data integrity guarantees
- Automatic resource cleanup

4.2.5 NFR-5: Usability

- Unix-style CLI conventions
- Responsive WebUI design
- Helpful error messages
- Comprehensive documentation

4.2.6 NFR-6: Maintainability

- Go best practices
- Externalized YAML configuration
- Structured logging
- Health check endpoints

4.2.7 NFR-7: Portability

- Cross-platform CLI (Linux, macOS, Windows)
- Modern browser support
- Docker deployment support

4.2.8 NFR-8: Compatibility

- MSSQL Server 2016+
- MySQL 5.7+
- API versioning
- Backward compatibility

4.3 Actors & Use Cases

4.3.1 System Actors

Actor descriptions

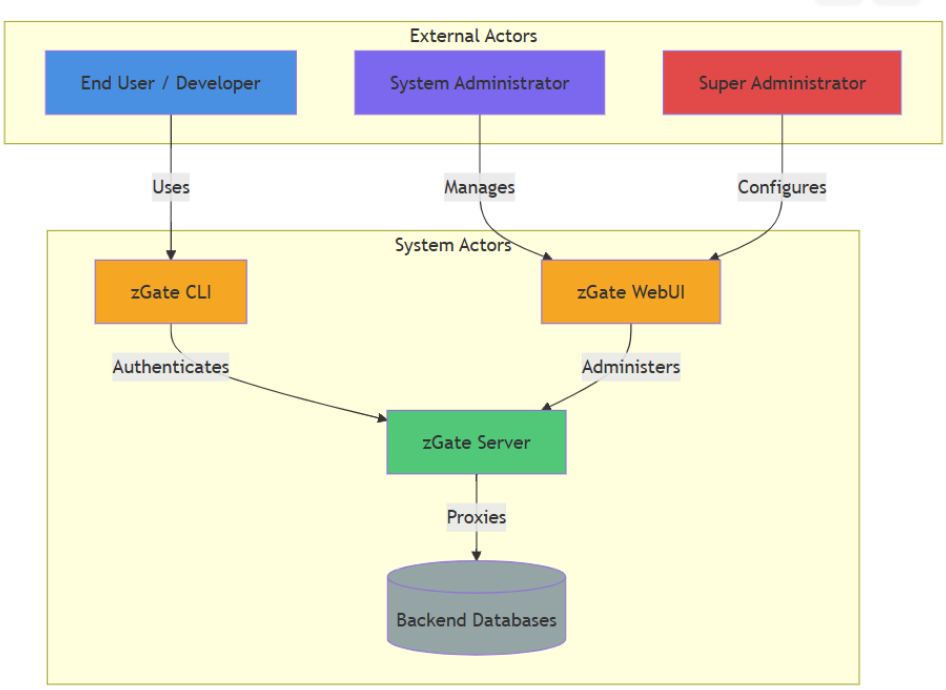


Figure 4.1: Overview of system actors: End Users, System Administrators, Super Administrators, and Backend Databases

End User / Developer

- **Description:** A regular user who needs to access databases for development, analytics, or operations
- **Responsibilities:**
  - Authenticate with the system using CLI
  - Request access to authorized databases
  - Execute database queries within granted permissions
- **Technical Level:** Intermediate (familiar with command line and database clients)

System Administrator

- **Description:** An admin user responsible for day-to-day management of the zGate platform
- **Responsibilities:**
  - Manage user accounts and role assignments
  - Configure database connections
  - Monitor active sessions
  - Review audit logs
  - Manage shared account pools
- **Technical Level:** Advanced (understands database administration and security)

Super Administrator

- **Description:** A privileged admin with full control over the system
- **Responsibilities:**
  - Create and manage admin accounts
  - Configure system-wide settings
  - Manage roles and permissions
  - Handle security incidents
- **Technical Level:** Expert (deep understanding of security and database systems)

zGate Server

- **Description:** The core backend system providing authentication, authorization, and proxy services
- **Responsibilities:**
  - Authenticate users and issue tokens
  - Enforce access policies
  - Create and manage temporary database users
  - Proxy database connections
  - Maintain audit logs

zGate CLI

- **Description:** Command-line client application for end users
- **Responsibilities:**
  - Provide user-friendly interface for authentication
  - Manage token storage and refresh
  - Display available databases
  - Establish database connections

zGate WebUI

- **Description:** Web-based administration dashboard
- **Responsibilities:**
  - Provide graphical interface for system administration
  - Display real-time system status
  - Facilitate configuration management
  - Present audit trail visualization

Backend Databases

- **Description:** Target database systems (MSSQL, MySQL) that users need to access
- **Responsibilities:**
  - Store application data
  - Accept connections from zGate proxy
  - Enforce database-level permissions

4.3.2 Use Case Catalog

UC-1: User Authentication via CLI

Actor: End User
Preconditions: User has valid credentials

Main Flow:

1. User executes `zgate login` command
2. CLI prompts for username and password
3. CLI sends credentials to zGate Server
4. Server validates credentials against user configuration
5. Server generates access and refresh tokens
6. Server returns tokens to CLI
7. CLI stores tokens securely in OS keyring

Postconditions: User is authenticated and tokens are stored
Alternative Flows:

- 3a: Network connection fails → CLI displays error and retries
- 4a: Invalid credentials → Server returns 401, CLI displays error

UC-2: List Available Databases

Actor: End User
Preconditions: User is authenticated

Main Flow:

1. User executes `zgate list` command
2. CLI retrieves stored access token
3. CLI sends list request to zGate Server with token
4. Server validates token signature and expiry
5. Server retrieves user permissions from configuration
6. Server filters database list based on user permissions
7. Server returns list of accessible databases with permission levels

8. CLI displays formatted database list

Postconditions: User sees their accessible databases
Alternative Flows:

- 2a: Token expired → CLI auto-refreshes using refresh token
- 4a: Token invalid → CLI prompts for re-login

UC-3: Connect to Database

Actor: End User
Preconditions: User is authenticated and has permission to target database
Main Flow:

1. User executes `zgate connect --database <db_name>`
2. CLI sends connection request to zGate Server
3. Server validates user has permission for requested database
4. Server retrieves database configuration
5. Server allocates dynamic local port for proxy
6. Server retrieves protocol handler for database type
7. Server creates temporary database user with appropriate permissions
8. Server establishes connection to backend database
9. Server starts proxy listener on allocated port
10. Server returns connection details (host, port, temp credentials) to CLI
11. CLI displays connection information to user
12. User connects using database client with provided credentials

Postconditions: Secure proxy connection established, temporary user created
Alternative Flows:

- 3a: User lacks permission → Server returns 403, CLI displays error
- 4a: Database not found → Server returns 404
- 7a: Backend database unavailable → Server returns error, cleanup any partial state
- 7b: Temporary user creation fails → Server aborts and returns error

UC-4: Auto Disconnect on Session End

Actor: System (zGate Server)
Trigger: User terminates database connection or network fails
Main Flow:

1. Server detects connection closure
2. Server drops temporary database user from backend
3. Server stops proxy listener
4. Server releases allocated port
5. Server logs disconnect event with user context

Postconditions: All session resources cleaned up, temporary user deleted

UC-5: Admin - Add Database Configuration

Actor: System Administrator
Preconditions: Admin is logged into WebUI
Main Flow:

1. Admin navigates to Databases page
2. Admin clicks ”Add Database” button
3. WebUI displays database creation form
4. Admin enters database details (name, type, backend address, admin credentials, permissions)
5. WebUI validates form inputs
6. WebUI sends create request to zGate Server API
7. Server validates admin token
8. Server tests connection to backend database
9. Server adds database configuration
10. Server persists configuration to databases.yaml
11. Server returns success response
12. WebUI displays updated database list

Postconditions: New database is available in configuration
Alternative Flows:

- 5a: Validation fails → Display errors on form
- 8a: Backend connection test fails → Return error, don't persist
- 9a: Duplicate database name → Return error

UC-6: Admin - Create User with Roles

Actor: System Administrator
Preconditions: Admin is logged into WebUI, roles exist
Main Flow:

1. Admin navigates to Users page
2. Admin clicks "Create User" button
3. WebUI displays user creation form
4. Admin enters username, password, and selects roles
5. WebUI validates form inputs
6. WebUI sends create user request to zGate Server API
7. Server validates admin token
8. Server hashes password using bcrypt
9. Server adds user configuration with assigned roles
10. Server persists configuration to users.yaml
11. Server returns success response
12. WebUI displays updated user list

Postconditions: New user can authenticate with assigned permissions
Alternative Flows:

- 5a: Validation fails → Display errors
- 9a: Username already exists → Return error

UC-7: Admin - Define Role with Permissions

Actor: System Administrator
Preconditions: Admin is logged into WebUI, databases exist
Main Flow:

1. Admin navigates to Access Control page
2. Admin clicks ”Create Role” button
3. WebUI displays role creation form
4. Admin enters role name and description
5. Admin selects databases and permission levels for each
6. WebUI validates inputs
7. WebUI sends create role request to zGate Server API
8. Server validates admin token
9. Server adds role configuration
10. Server persists configuration to roles.yaml
11. Server returns success response
12. WebUI displays updated role list

Postconditions: New role is available for assignment to users
Alternative Flows:

- 6a: Invalid permission level → Display error
- 9a: Role name already exists → Return error

UC-8: Admin - Monitor Active Sessions

Actor: System Administrator
Preconditions: Admin is logged into WebUI
Main Flow:

1. Admin navigates to Sessions page
2. WebUI sends request to fetch active sessions
3. Server validates admin token

4. Server retrieves all active proxy sessions
5. Server returns session details (user, database, start time, connection info)
6. WebUI displays sessions in table format
7. Admin reviews active connections

Postconditions: Admin has visibility into current system usage
Alternative Flows:

- Admin terminates a session:
    1. Admin clicks "Terminate" on a session
    2. WebUI confirms action
    3. Server closes proxy connection
    4. Server deletes temporary database user
    5. WebUI updates session list

UC-9: Admin - Review Audit Logs

Actor: System Administrator
Preconditions: Admin is logged into WebUI
Main Flow:

1. Admin navigates to Activity Audit page
2. Admin optionally applies filters (user, date range, action type)
3. WebUI sends filtered log request to server
4. Server validates admin token
5. Server retrieves matching audit log entries
6. Server returns paginated log data
7. WebUI displays logs in chronological order with search/filter capabilities

Postconditions: Admin can track user activities and security events

UC-10: Auto Token Refresh

Actor: System (CLI)
Trigger: Access token nearing expiry

Main Flow:

1. CLI detects access token will expire within 1 minute
2. CLI retrieves refresh token from secure storage
3. CLI sends refresh request to zGate Server
4. Server validates refresh token
5. Server generates new access token
6. Server returns new access token
7. CLI updates stored access token
8. CLI proceeds with original request

Postconditions: User session continues seamlessly

Alternative Flows:

- 4a: Refresh token expired → CLI prompts user to login again

UC-11: Logout and Token Revocation

Actor: End User
Preconditions: User is authenticated

Main Flow:

1. User executes `zgate logout` command
2. CLI retrieves stored tokens
3. CLI sends logout request to zGate Server with refresh token
4. Server invalidates refresh token
5. Server returns success response
6. CLI deletes tokens from secure storage
7. CLI confirms logout to user

Postconditions: User session is terminated, tokens are invalid

UC-12: Configure Shared Account Pool

Actor: System Administrator
Preconditions: Admin is logged into WebUI, database exists

Main Flow:

1. Admin navigates to Shared Accounts page
2. Admin clicks "Add Shared Account"
3. WebUI displays configuration form
4. Admin selects database, permission level, enters credentials, sets max concurrent users
5. WebUI validates inputs
6. WebUI sends create request to server
7. Server validates admin token
8. Server tests credentials against database
9. Server creates shared account pool configuration
10. Server persists configuration
11. WebUI displays updated shared account list

Postconditions: Shared account pool is available for session allocation
Alternative Flows:

- 8a: Credential test fails → Return error, don't persist

4.4 Use Case Diagrams

4.4.1 End User Case Diagrams

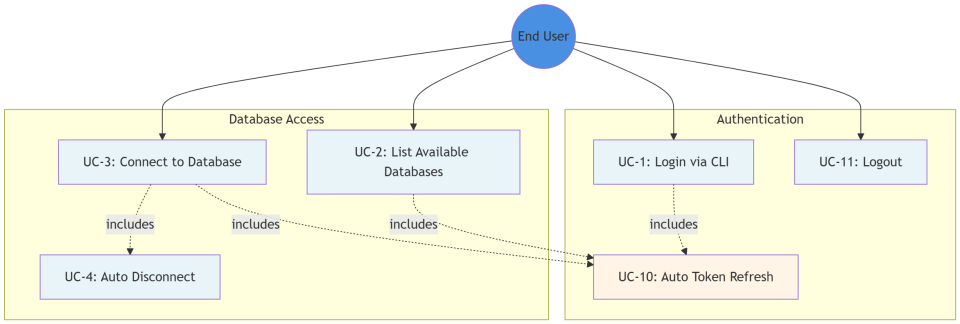


Figure 4.2: End user workflows: authentication, database listing, connection, session management, and logout

4.4.2 Administrator Use Case Diagrams

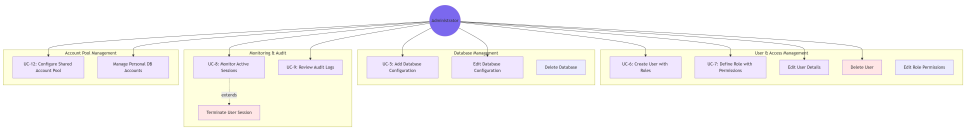


Figure 4.3: Administrator functions: database configuration, user and role management, session monitoring, and audit review

4.4.3 Complete System Use Case Diagrams

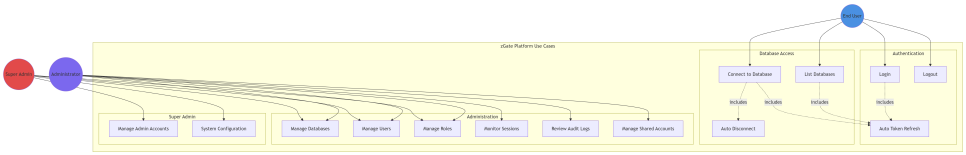


Figure 4.4: Complete system use case diagram showing all actor interactions within the zGate Gateway

4.4.4 System Components Interaction

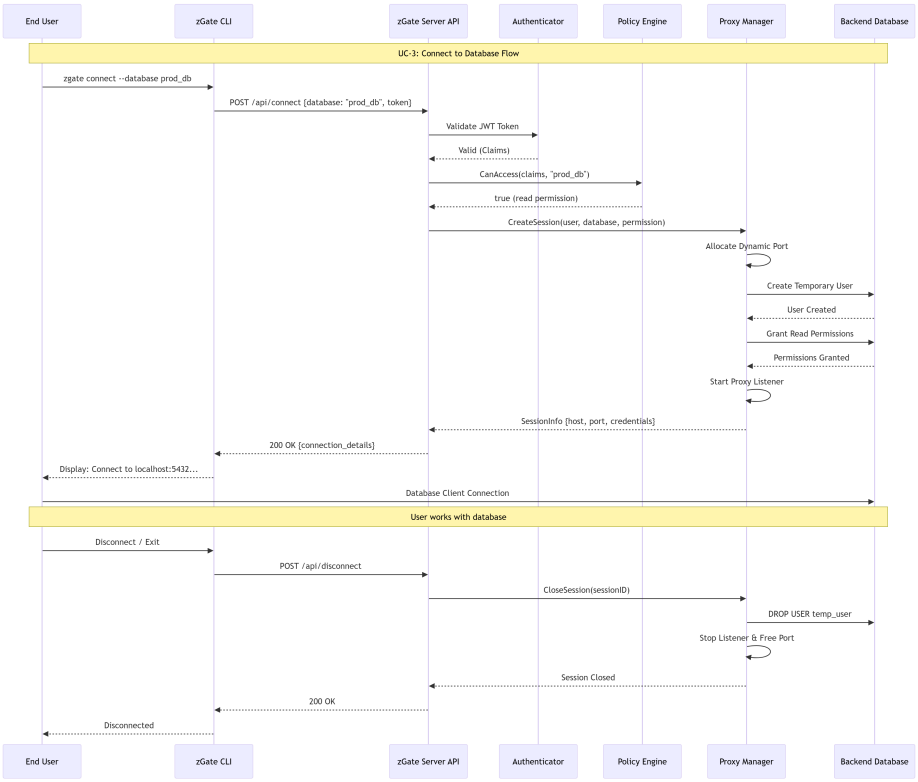


Figure 4.5: System component interactions showing data flow between CLI, Gateway Server, Web Dashboard, Policy Engine, and Backend Databases

4.5 User Stories

4.5.1 End User Stories

Epic 1: Secure Database Access

US-1.1: Login with Credentials

As an end user,
I want to
login using my username and password via the CLI
So that
I can securely authenticate and access authorized databases
Acceptance Criteria:

- CLI prompts for username and password
- Credentials are validated against the server

- Access and refresh tokens are generated on success
- Meaningful error message shown for invalid credentials
- Login completes within 500ms

US-1.2: View My Accessible Databases

As an authenticated user,
I want to
list all databases I have permission to access
So that
I know which databases I can connect to
Acceptance Criteria:

- Command `zgate list` shows my accessible databases
- Each database shows: name, type, my permission level, status, description
- Only databases I have permission for are displayed
- Offline databases are clearly marked
- List loads within 200ms

US-1.3: Connect to a Database

As an authenticated user,
I want to
connect to a specific database using the CLI
So that
I can access data within my permissions
Acceptance Criteria:

- Command `zgate connect --database <name>` initiates connection
- System validates my permission before allowing access
- Temporary credentials are auto-generated
- Connection details (host, port, username, password) are displayed
- I can use any database client with the provided credentials
- I receive clear error if I lack permission
- Connection establishes within 2 seconds

US-1.4: Automatic Session Cleanup

As an end user,
I want
my temporary database access to be automatically cleaned up when I disconnect
So that
I don't have to worry about security hygiene
Acceptance Criteria:

- Temporary database user is deleted when I close my connection
- Proxy resources are released automatically
- Cleanup happens within 30 seconds of disconnect
- No manual intervention required

US-1.5: Seamless Token Refresh

As an authenticated user,
I want
my session to continue without interruption
So that
I don't have to re-login frequently
Acceptance Criteria:

- Access tokens are automatically refreshed before expiry
- I don't experience any interruption during refresh
- I'm only prompted to re-login if my refresh token expires
- Refresh happens transparently in the background

US-1.6: Check Session Status

As an authenticated user,
I want to
check my current session status
So that
I know if I'm logged in and when my session expires
Acceptance Criteria:

- Command `zgate status` shows my current session
- Display includes: username, token expiry, server connection status
- Clear indication if I'm not logged in

US-1.7: Secure Logout

As an authenticated user,
I want to
logout and revoke my session
So that
my credentials are invalidated when I'm done
Acceptance Criteria:

- Command `zgate logout` terminates my session
- Tokens are revoked on the server
- Local token storage is cleared
- Confirmation message is displayed

4.5.2 Administrator Stories

Epic 2: User & Access Management

US-2.1: Create User Accounts

As a system administrator,
I want to
create new user accounts with assigned roles
So that
team members can access databases based on their responsibilities
Acceptance Criteria:

- I can access user creation form in WebUI
- I can enter username and initial password
- I can assign one or more roles to the user
- Password is securely hashed before storage

- User can login immediately after creation
- Duplicate usernames are prevented with clear error

US-2.2: Define Roles with Granular Permissions

As a system administrator,
I want to
define roles with specific database permissions
So that
I can implement least-privilege access control
Acceptance Criteria:

- I can create named roles (e.g., ”data_analyst”, ”developer”)
- For each role, I can specify permissions per database
- Permission levels include: read, write, admin
- Roles can be assigned to multiple users
- Changing role permissions affects all assigned users immediately

US-2.3: Manage User-Role Assignments

As a system administrator,
I want to
add or remove roles from existing users
So that
I can adjust access as team responsibilities change
Acceptance Criteria:

- I can view current role assignments for any user
- I can add new roles to a user
- I can remove roles from a user
- Changes take effect immediately for new connections
- Audit log captures all role changes

Epic 3: Database Configuration

US-3.1: Add Database Configurations

As a system administrator,
I want to
add new database connections to the system
So that
users can access additional data sources
Acceptance Criteria:

- I can provide: name, type (MSSQL/MySQL), backend address, admin credentials
- I can specify available permission levels for the database
- Connection is tested before saving
- Configuration is persisted to databases.yaml
- Database appears in user lists immediately
- Clear error shown if connection test fails

US-3.2: Edit Database Settings

As a system administrator,
I want to
update existing database configurations
So that
I can reflect infrastructure changes
Acceptance Criteria:

- I can modify connection details (address, credentials)
- I can update available permissions
- Changes are validated before saving
- Active sessions using old config are not disrupted
- New connections use updated configuration

US-3.3: Remove Decommissioned Databases

As a system administrator,
I want to
delete database configurations that are no longer needed
So that
users don't see outdated options
Acceptance Criteria:

- I can delete a database configuration
- System confirms deletion to prevent accidents
- Database is immediately removed from user lists
- Deletion is logged in audit trail

Epic 4: Monitoring & Audit

US-4.1: Monitor Active Sessions

As a system administrator,
I want to
see all active database sessions in real-time
So that
I can monitor system usage and detect anomalies
Acceptance Criteria:

- I can view list of all active connections
- Each session shows: user, database, connection time, status
- List updates in real-time or on refresh
- I can see connection details (proxy port, temp username)

US-4.2: Terminate Suspicious Sessions

As a system administrator,
I want to
forcefully terminate active sessions
So that
I can respond to security incidents

Acceptance Criteria:

- I can select any active session and terminate it
- System confirms action before executing
- Session proxy is closed immediately
- Temporary database user is deleted
- Action is logged in audit trail
- User receives connection closed error

US-4.3: Review Comprehensive Audit Logs

As a system administrator,
I want to
review all user activities and system events
So that
I can investigate issues and maintain compliance
Acceptance Criteria:

- I can view chronological audit log in WebUI
- Each entry includes: timestamp, user, action, outcome, context
- I can filter by: user, date range, action type, success/failure
- I can search logs by keywords
- Logs include: logins, connection requests, admin actions, errors
- Logs are paginated for performance

US-4.4: Export Audit Logs

As a system administrator,
I want to
export audit logs for compliance reporting
So that
I can provide evidence of access controls
Acceptance Criteria:

- I can export filtered logs to CSV or JSON

- Export includes all relevant fields
- Large exports don't timeout or crash

Epic 5: Advanced Access Control

US-5.1: Configure Shared Account Pools

As a system administrator,
I want to
create pools of shared database credentials
So that
multiple users can share backend accounts efficiently
Acceptance Criteria:

- I can create shared account pool for a database
- I can specify: database, permission level, credentials, max concurrent users
- System allocates shared account when user connects
- System prevents exceeding max concurrent limit
- User transparently gets shared credentials
- Account is returned to pool on disconnect

US-5.2: Assign Personal Database Accounts

As a system administrator,
I want to
assign personal database accounts to specific users
So that
some users have dedicated credentials for auditing
Acceptance Criteria:

- I can create personal account for user-database pair
- I can provide specific database credentials
- User always gets their personal account when connecting
- Personal accounts override shared pools
- I can revoke personal accounts

US-5.3: Define Custom User Permissions

As a system administrator,
I want to
grant custom permissions to individual users beyond their roles
So that
I can handle special cases without creating new roles
Acceptance Criteria:

- I can add database permissions directly to a user
- Custom permissions combine with role-based permissions
- I can see both role and custom permissions in user view
- Custom permissions can be removed independently

4.5.3 Super Administrator Stories

Epic 6: Platform Administration

US-6.1: Manage Admin Accounts

As a super administrator,
I want to
create and manage admin user accounts
So that
I can grant administrative access to trusted personnel
Acceptance Criteria:

- I can create new admin accounts with secure passwords
- I can view all existing admin accounts
- I can disable or delete admin accounts
- Regular users cannot access admin functions
- Admin account changes are logged

US-6.2: Configure System Settings

As a super administrator,
I want to
configure system-wide settings
So that
I can tune the platform for our environment
Acceptance Criteria:

- I can set token expiry durations (access and refresh)
- I can configure backend connection timeouts
- I can set log levels (DEBUG, INFO, WARN, ERROR)
- Settings are validated before applying
- Configuration changes are persisted

US-6.3: Perform System Health Checks

As a super administrator,
I want to
check the health of all system components
So that
I can proactively identify issues
Acceptance Criteria:

- I can view status of: API server, proxy manager, backend databases
- Each database shows: online/offline status, last check time
- I can manually trigger connectivity tests
- Failed health checks generate alerts

4.5.4 System Stories (Non-Interactive)

Epic 7: Automated Operations

US-7.1: Automatic Resource Cleanup

As the zGate system,
I need to
automatically clean up temporary resources
So that
database systems don't accumulate orphaned users
Acceptance Criteria:

- Temporary database users are deleted within 30s of disconnect
- Proxy listeners are stopped immediately on session close
- Allocated ports are freed for reuse
- Cleanup happens even if client disconnects ungracefully
- Failed cleanup attempts are retried with exponential backoff
- Persistent cleanup failures are logged as errors

US-7.2: Real-Time Permission Enforcement

As the zGate system,
I need to
evaluate permissions at request time using current configuration
So that
permission changes take effect immediately
Acceptance Criteria:

- Each connection request checks latest roles and permissions
- Configuration file changes are detected and reloaded
- Users with revoked permissions cannot establish new connections
- Permission checks complete within 50ms
- Permission evaluation is thread-safe

US-7.3: Graceful Error Handling

As the zGate system,
I need to
handle errors gracefully without leaking sensitive information

So that
users get helpful feedback while maintaining security
Acceptance Criteria:

- User-facing errors don't expose internal paths or stack traces
- Authentication failures return generic "invalid credentials" message
- Network errors suggest retry with backoff
- All errors are logged with full context
- Critical errors trigger alerts to administrators

US-7.4: Comprehensive Audit Logging

As the zGate system,
I need to
log all security-relevant events
So that
administrators can audit access and investigate incidents
Acceptance Criteria:

- All authentication attempts are logged (success and failure)
- All database access requests are logged with user context
- All administrative actions are logged
- Logs include: timestamp, user, action, outcome, IP address, session ID
- Logs are structured (JSON format option)
- Log rotation prevents disk exhaustion
- Sensitive data (passwords, credentials) are never logged

5. Proposed Solution

5.1 Solution Overview

The proposed solution is **zGate**, a comprehensive Zero-Trust Database Access Platform that fundamentally changes how organizations manage database security and access control. Rather than relying on shared credentials and static permissions, zGate introduces a dynamic, policy-driven approach where temporary credentials are created for each user session, eliminating the risks associated with credential sprawl and unauthorized access.

5.1.1 The Core Problem

Traditional database access models suffer from several critical security and operational challenges:

1. **Shared Credentials:** Multiple users share common database accounts, making it impossible to audit who performed which actions
2. **Static Permissions:** Once granted, permissions remain indefinitely, increasing the attack surface
3. **Credential Sprawl:** Database passwords stored in configuration files, wikis, and password managers
4. **Lack of Visibility:** No centralized view of who has access to which databases
5. **Manual Overhead:** Granting and revoking access requires manual database administration
6. **Compliance Gaps:** Difficult to prove least-privilege access and maintain audit trails

5.1.2 The zGate Solution

zGate addresses these challenges through a multi-layered architecture built on three core principles:

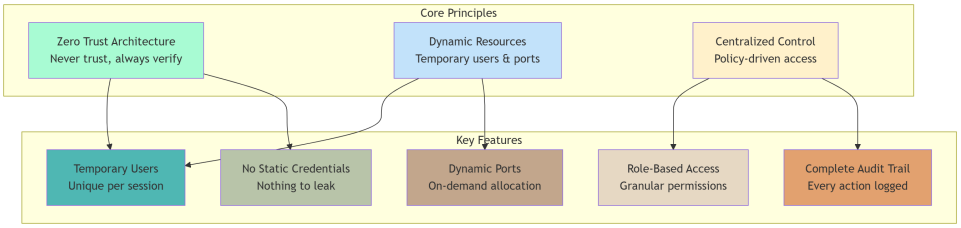


Figure 5.1: zgate proposed solution architecture

5.2 Core Components

5.2.1 zGate Server (Backend)

The server is the heart of the platform, handling all security, policy enforcement, and proxy operations.

Backend Layers

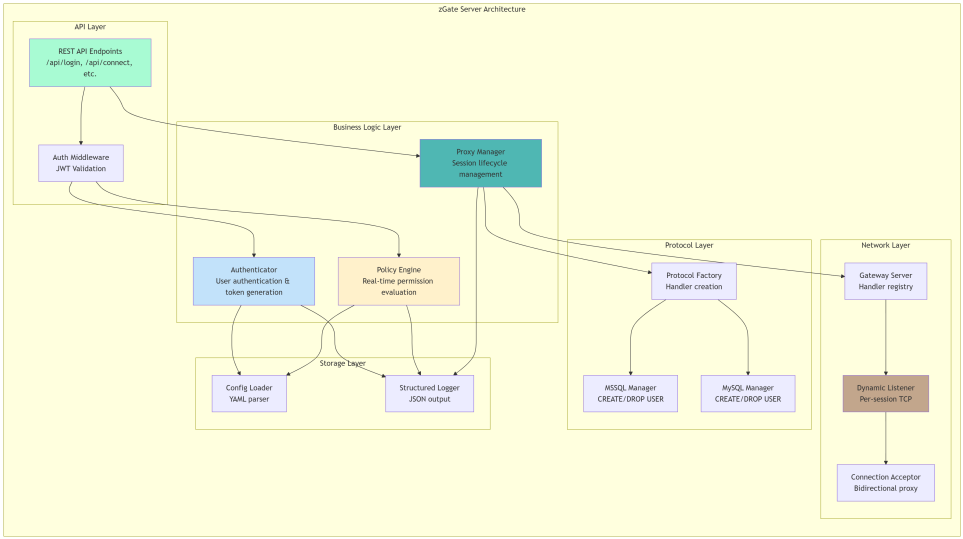


Figure 5.2: zGate backend architecture showing authentication, policy engine, and proxy layers

Key Features

1. JWT-Based Authentication

- Access tokens (15-minute TTL) for API requests
- Refresh tokens (7-day TTL) for session persistence
- Secure token generation with HMAC-SHA256 signatures

- Automatic token refresh in CLI

2. Real-Time Policy Enforcement

- Permissions evaluated from current configuration on every request
- Role-based access control (RBAC) with support for multiple roles per user
- Custom permissions at user level for exceptions
- Immediate effect when permissions are changed (no cache invalidation needed)

3. Dynamic Proxy Management

- On-demand creation of proxy sessions
- Automatic port allocation from ephemeral range
- Temporary database user creation with appropriate grants
- Automatic cleanup on disconnect or crash recovery

4. Protocol Abstraction

- Pluggable architecture for database types
- Current support: MSSQL, MySQL
- Easy extensibility for PostgreSQL, Oracle, etc.
- Database-specific SQL for user management

5. Comprehensive Logging

- Structured logging with key-value pairs
- Every security event logged with user context
- Support for JSON output for log aggregation
- Configurable log levels (DEBUG, INFO, WARN, ERROR)

5.2.2 zGate CLI (Client)

The command-line interface provides developers with a simple, secure way to access databases.

CLI Workflow

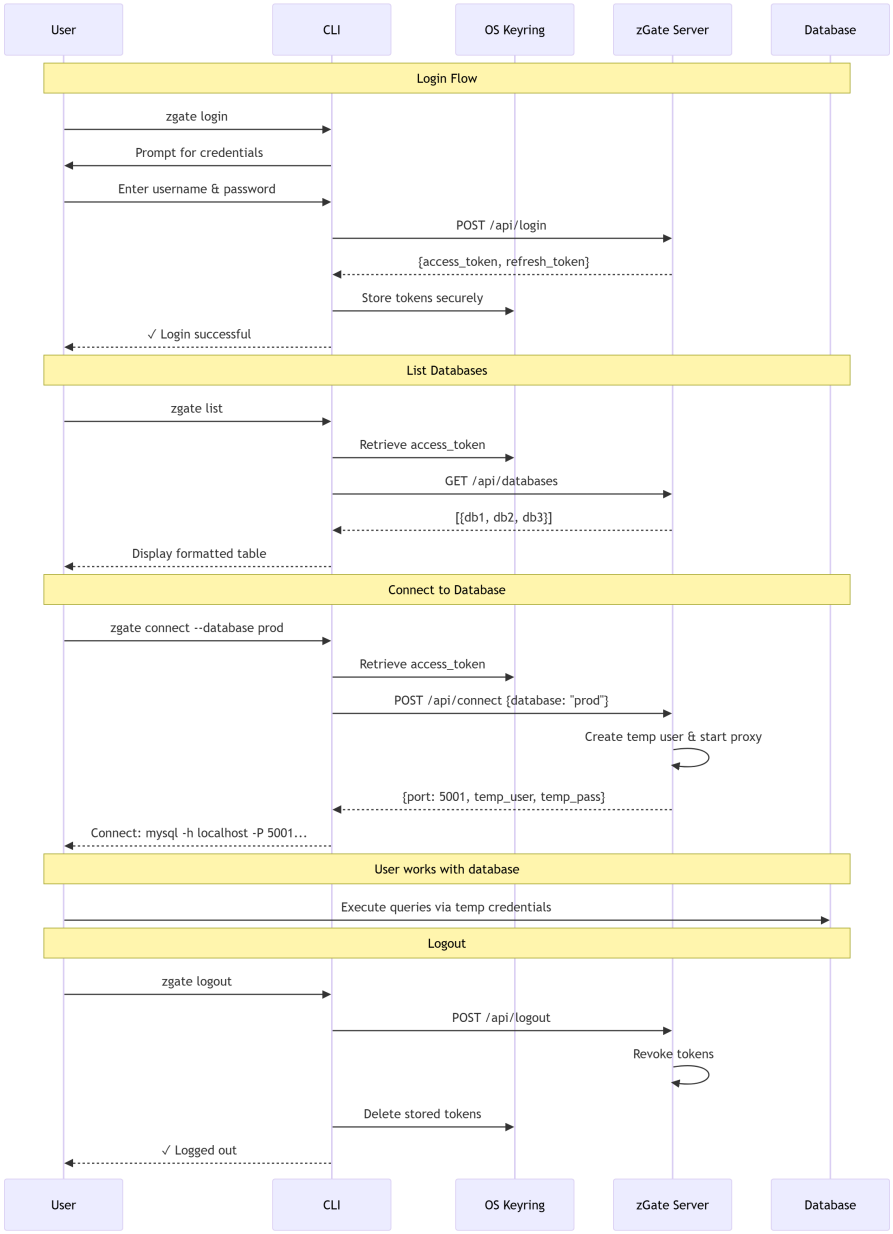


Figure 5.3: zGate CLI workflow illustrating token management and database access process

CLI Features

1. Secure Token Storage

- OS-native keyring integration (Keychain on macOS, Credential Manager on Windows, Secret Service on Linux)
- Encrypted file fallback for headless environments
- Never stores plaintext credentials

2. Automatic Token Refresh

- Detects token expiry proactively
- Refreshes access token using refresh token
- Seamless user experience without re-authentication

3. User-Friendly Interface

- Simple, intuitive commands following Unix conventions
- Colored output for better readability
- Helpful error messages with troubleshooting hints
- Progress indicators for long operations

4. Cross-Platform Support

- Single binary for Windows, macOS, Linux
- No runtime dependencies
- Portable and lightweight (~10MB)

5.2.3 zGate WebUI (Admin Dashboard)

The web interface provides administrators with a comprehensive control panel for managing the entire platform.

Dashboard Architecture

Admin Features

1. Database Management

- Add/edit/delete database configurations
- Test database connectivity before saving
- View all configured databases with status
- Manage available permissions per database

2. User & Role Management

- Create users with bcrypt-hashed passwords

- Assign multiple roles to users
- Define roles with database-specific permissions
- Add custom permissions to individual users

3. Session Monitoring

- Real-time view of active database connections
- See: user, database, port, temporary username, duration
- Terminate sessions if needed (security incidents)
- Filter and search sessions

4. Audit Trail

- Comprehensive logs of all user activities
- Filter by user, date range, action type
- Search within log entries
- Export for compliance reporting

5. Advanced Features

- Shared account pools for efficient credential management
- Personal database accounts for specific users
- Query execution interface (future)
- Connected databases view

5.3 Key Technologies & Design Decisions

Technology Stack

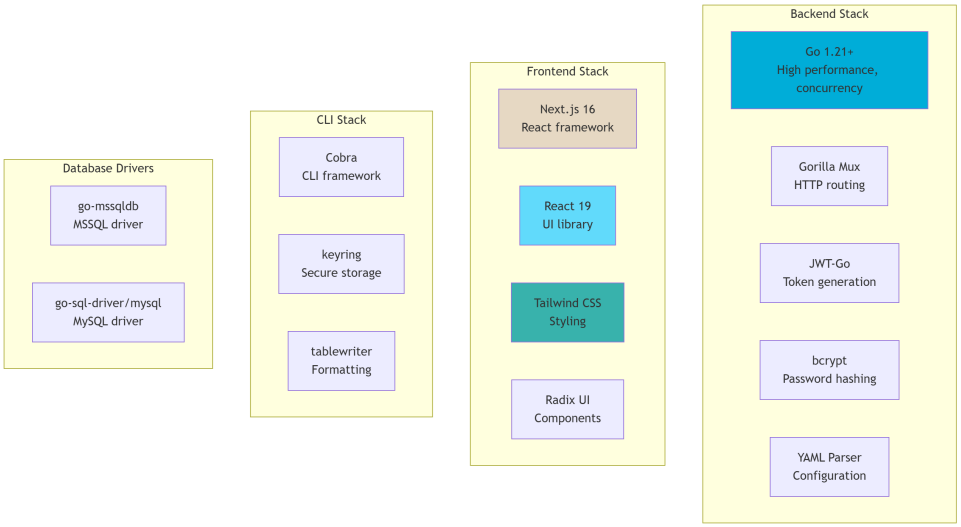


Figure 5.4: zGate technology stack

5.4 Security Architecture

Security is woven into every layer of the zGate platform through multiple defense mechanisms.

5.4.1 Security Layers

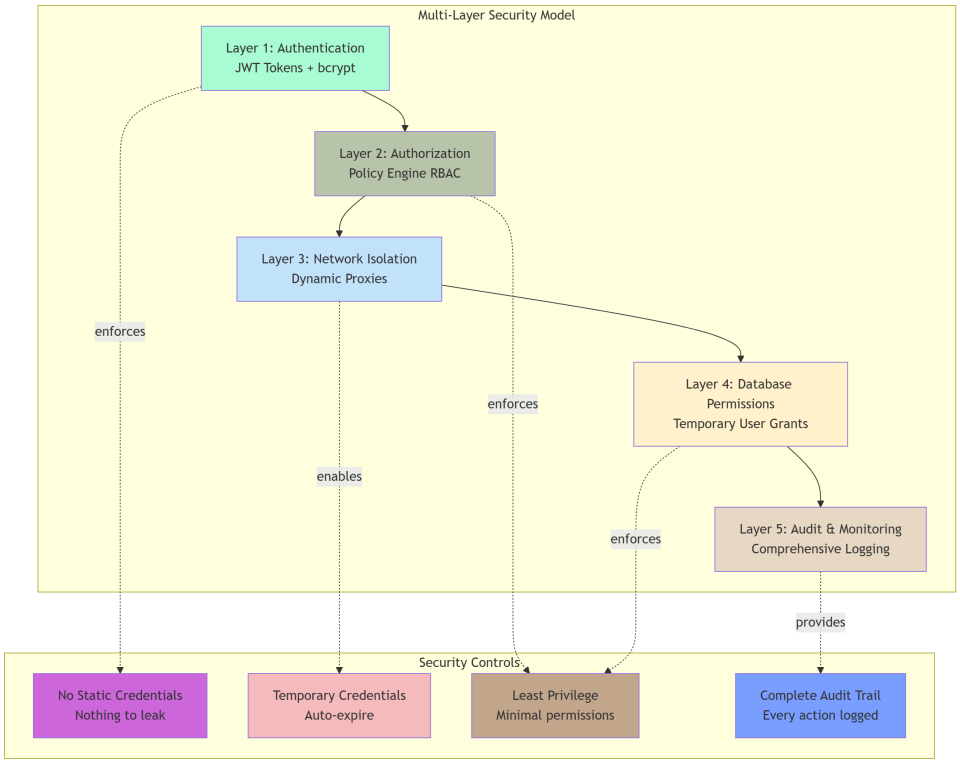


Figure 5.5: Multi-layered security architecture in zGate platform

5.4.2 Security Guarantees

Threat	Mitigation	Implementation
Credential Theft	No static credentials exposed to users	Temporary users with auto-generated passwords
Privilege Escalation	Database-level permission enforcement	Grants limited to user's role (read/write/admin)
Unauthorized Access	Real-time policy evaluation	Every connection request checked against current config
Session Hijacking	Short-lived JWT tokens	15-minute access token expiry
Password Cracking	Strong password hashing	bcrypt with work factor 10
Audit Evasion	Mandatory logging	Every API call, connection, disconnect logged
Resource Exhaustion	Automatic cleanup	Temporary users deleted within 30 seconds of disconnect
Man-in-the-Middle	Encrypted connections	TLS support for backend database connections

Table 5.1: Security threats and their mitigations in zGate

5.5 Advantages of the Proposed Solution

Key Benefits

Enhanced Security

- Eliminates shared credentials
- Temporary users auto-deleted
- No credential sprawl
- Cryptographically secure random passwords
- Short-lived JWT tokens
- Database-level permission enforcement

Operational Efficiency

- Automated user provisioning
- Self-service for developers (via CLI)
- Centralized access management
- No manual database administration
- Quick onboarding/offboarding

Compliance & Audit

- Complete audit trail
- User-attributed actions
- Provable least-privilege
- Real-time access reporting
- Exportable logs

Developer Experience

- Simple CLI commands
- Works with existing tools
- No VPN/bastion required
- Automatic token refresh
- Cross-platform support

Scalability

- Handles thousands of concurrent sessions
- Stateless API (horizontal scaling)
- Lightweight memory footprint
- Dynamic resource allocation

6. Alignment with International Standards

6.1 PCI DSS

- **Access Control (Req. 7):** Role-based access with least privilege per database
- **Authentication (Req. 8):** Unique user identification, no shared credentials, time-limited sessions
- **Audit Logging (Req. 10):** Complete tracking of all database access with user identity, timestamp, and session details

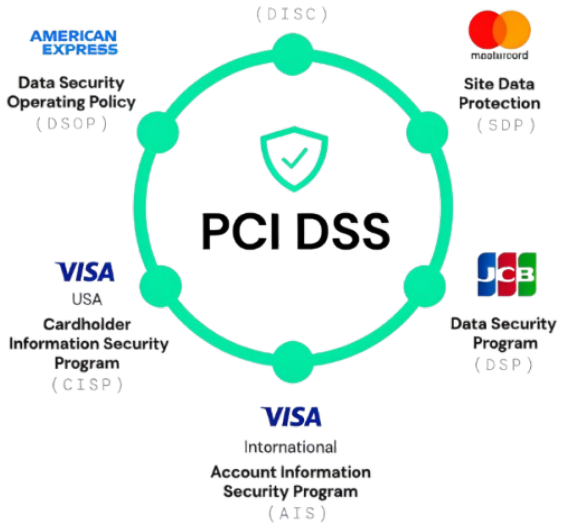


Figure 6.1: zGate alignment with PCI DSS requirements for access control, authentication, and audit logging

6.2 HIPAA

- **Access Control:** Unique user authentication and role-based database access to ePHI
- **Audit Controls:** Comprehensive logs of who accessed what database and when
- **Authentication:** JWT-based identity verification before database access
- **Automatic Log-off:** Session expiration and token timeout mechanisms

6.3 GDPR

- **Data Security (Art. 32):** Identity-based authentication, authorization, and encrypted credentials
- **Access Control (Art. 5):** Role-based permissions limiting data access scope
- **Audit Records (Art. 30):** Complete logging of processing activities with user context
- **Breach Detection (Art. 33):** Real-time monitoring and immediate session termination capabilities

Figure 6.2: zGate alignment with GDPR requirements for data security, access control, audit records, and breach detection

6.4 ISO 27001

- **User Access Management (A.9.2):** Formal registration, role assignment, and access revocation
- **Access Control (A.9.4):** Secure authentication, temporary credentials, session timeouts
- **Logging & Monitoring (A.12.4):** Event logging for all activities with protected audit trails
- **Compliance Reviews (A.18):** Audit logs and monitoring dashboard for compliance verification

7. Competitor & Market Analysis

7.1 Competitor Analysis

7.1.1 Comparison Table

7.1.2 What Competitors Lack

7.2 Market Research

7.2.1 Market Overview

7.2.2 Zero Trust Demand

7.2.3 Market Challenges & Needs

7.2.4 Regulatory Drivers

7.2.5 Trends & Opportunities

7.2.6 Landscape Summary

8. Scientific Research & Literature Review

8.1 Generative AI-Enhanced Cybersecurity Framework for Enterprise Data Privacy Management

8.1.1 Purpose of the Study

The paper addresses the growing need for organizations to secure sensitive enterprise data (e.g., financial transactions, patient records, IoT data) while still enabling advanced detection of cyber threats. Traditional security controls and anomaly detection methods often:

- Miss new/unknown attack patterns
- Require direct use of real sensitive data, creating privacy and compliance risks

Goal: The authors propose a Generative AI-enhanced framework that combines Generative Adversarial Networks (GANs), Variational Autoencoders (VAEs), and traditional machine learning (ML/DL) anomaly detection with strong privacy-preserving methods (Differential Privacy, encryption, masking). The aim is to balance data privacy, detection accuracy, and computational efficiency.

8.1.2 Framework Overview

The proposed framework works as an end-to-end pipeline with the following main components:

- **Data ingestion:** Collect logs (network, system, application) via tools like Splunk/ELK
- **Generative AI layer (GANs & VAEs):** Create synthetic, privacy-safe data that mimic real-world patterns without exposing identities
- **Privacy layer:** Apply Differential Privacy ($\varepsilon = 0.1$ for highly sensitive data), AES-256 encryption, TLS 1.3, and masking
- **Anomaly detection engine:** Train models like Random Forest, SVM, and LSTM on synthetic + sanitized data to detect unusual activity

- **Monitoring & alerting:** Real-time detection with dashboards and alert systems

Analogy: Instead of training guards with real customer data (which is risky), the system uses highly realistic "actors" (synthetic data) to train them — ensuring the guards learn effectively without ever seeing the real people.

8.1.3 Implementation & Experiments

The framework was tested in three simulated enterprise domains:

- **Finance (transaction logs):** 94% accuracy, 95% recall, ∼1.2–1.5 seconds per transaction
- **Healthcare (EHR access logs):** 96% accuracy, 93% precision, ∼1.5 seconds per event
- **Smart City/IoT (sensor data):** 91% accuracy, F1 ≈ 90%, latency <100 ms at the edge

Performance trade-offs:

- GAN framework: ∼96% accuracy, moderate compute (4GB GPU, 2.5h training)
- LSTM: ∼97% accuracy but higher GPU needs (6GB)
- Traditional ML (RF/SVM): lower accuracy (∼92–94%) but lighter
- Very high-accuracy CNN (>99%): impractical resource usage

8.1.4 Privacy & Security Features

- **Differential Privacy:** Adds noise to hide individual user data, ensuring compliance with GDPR/HIPAA
- **Encryption:** AES-256 for data at rest, TLS 1.3 for data in transit
- **Access control:** Role-based restrictions
- **Data masking:** Obscures identifiers in logs

8.1.5 Contributions to the Paper

- First comprehensive framework integrating Generative AI + privacy techniques + anomaly detection
- Provides balanced performance: strong accuracy without extreme computational demands
- Applicable across finance, healthcare, and IoT
- Offers implementation guidance with practical tools (TensorFlow, PyTorch, Scikit-learn, PySyft)

8.1.6 Advantages & Limitations

Advantages:

- Protects privacy while enabling effective training
- Can detect novel/rare attacks better by augmenting datasets with synthetic samples
- Works across domains, modular and adaptable
- More resource-efficient than some deep CNN methods

Limitations:

- Results are simulated, not from live production environments
- Quality of synthetic data can affect detection accuracy
- Managing GANs, VAEs, DP, and anomaly detectors is operationally complex
- Differential Privacy trade-off: stronger privacy (smaller ε) may reduce model accuracy

8.1.7 Relevance to Our Project

This study is directly relevant because:

- Our project focuses on secure access and monitoring of sensitive databases
- The paper's synthetic-data + anomaly detection pipeline is a practical approach to train models without exposing real database queries/records
- Techniques like Differential Privacy, RBAC, AES-256 encryption overlap with Zero Trust principles (least privilege, continuous monitoring, encryption everywhere)
- Their results show that real-time detection with privacy is feasible, which strengthens the foundation for our Zero Trust access model

8.2 The Significance of Artificial Intelligence in Zero Trust Technologies: A Comprehensive Review

8.2.1 Problem Addressed

Traditional security models assume anything inside a company's network can be trusted. With today's cloud, remote work, and hybrid environments, this assumption no longer holds. Attackers exploit cloud resources, lateral movement inside networks, and slow manual controls. The study addresses how Artificial Intelligence (AI) can enhance the Zero Trust (ZT) model to meet these modern challenges.

8.2.2 Methodology

- Literature review (20+ studies examined)
- Synthesized how AI is applied across ZT building blocks (IAM, MFA, EDR, ZTNA, SASE, Network Analytics)

8.2.3 Key Contributions of AI to Zero Trust

Identity & Access Management (IAM)

- **Authentication:** Adaptive and continuous (AI monitors typing, device, location; flags anomalies)
- **Authorization:** Intelligent Role-Based Access Control (AI suggests roles, prevents "over-privilege")
- **Administration:** Automated onboarding/offboarding, policy adjustments
- **Audit/Compliance:** AI generates audit trails, suggests policies, detects compliance gaps

Adaptive Multi-Factor Authentication (AMFA)

- AI adjusts authentication strength based on risk (low $\rightarrow$ password; medium $\rightarrow$ OTP; high $\rightarrow$ biometrics)
- Balances usability with security

Endpoint Detection & Response (EDR)

- AI baselines device behavior, detects anomalies, reduces false positives
- Automates containment (isolate compromised laptops)

Zero Trust Network Access (ZTNA) & Secure Access Service Edge (SASE)

- ZTNA grants application-level access (not full network like VPN)
- SASE combines SD-WAN + ZTNA + CASB + FWaaS; AI analyzes telemetry, recommends segmentation
- AI enables dynamic microsegmentation and automated policy creation

8.2.4 Findings

- AI strengthens Zero Trust by making it continuous, adaptive, and automated
- AI reduces human error, speeds up detection, and scales across large organizations
- AI integration is critical for modern cloud and hybrid infrastructures

8.2.5 Comparison with Traditional Methods

- Traditional perimeter security = trust anyone inside
- Zero Trust with AI = checkpoint at every request making context-based decisions in real time

8.2.6 Relevance to Our Project

- IAM insights → directly applicable for database user role mining & continuous verification
- Adaptive MFA → useful for database login protection
- EDR concepts → extend to database clients/endpoints
- ZTNA & SASE → inspire database-level microsegmentation (grant per-query or per-app access)
- Network analytics → parallels database traffic analysis for anomaly detection

8.3 Securing Digital Identity in the Zero Trust Architecture: A Blockchain Approach to Privacy-Focused Multi-Factor Authentication

8.3.1 Problem Addressed

- Traditional MFA depends on centralized servers, which are vulnerable to outages and breaches
- Zero Trust architectures require continuous, strong identity verification, but current MFA approaches are limited in resilience and privacy

The study addresses these issues by designing a decentralized, privacy-focused MFA mechanism that eliminates single points of failure and ensures secrets remain private.

8.3.2 Research Goals

- Build a decentralized authentication system aligned with Zero Trust principles
- Ensure privacy-preserving verification so users never expose OTPs
- Provide auditability and traceability of authentication events
- Deliver a realistic proof-of-concept that demonstrates feasibility and performance

8.3.3 Proposed System

- **Distributed Authentication Mechanism (DAM):** validator nodes collectively handle authentication
- **Distributed OTP Generation:** each validator contributes a random partial secret; combined into full OTP
- **Privacy via zk-SNARKs:** users prove they know the OTP without revealing it
- **Authentication Token:** successful proof leads to issuance of a non-transferable NFT (digital badge) valid for a period

8.3.4 Experimental Results

- **Performance:** comparable to real-world MFA timings ($\sim$20 seconds average)
- **Security:** analysis shows probability of attack success is negligible due to distribution, cryptographic verification, and non-transferability

8.3.5 Key Findings and Contributions

- Decentralized authentication reduces single points of failure
- OTP secrets are never exposed; verification occurs through zk-SNARK proofs
- Immutable, auditable on-chain logs improve accountability
- Non-transferable NFTs prevent token theft or resale

8.3.6 Real-World Applications

- **Banking & Fintech:** customers receive distributed OTPs and prove knowledge via zk-SNARK, receiving session tokens for access
- **Corporate IT (Zero Trust):** employees authenticate and receive short-lived NFTs, ensuring continuous verification without exposing passwords
- **Developer Platforms:** integration with existing blockchain and web authentication frameworks for higher resilience

8.3.7 Relevance to Our Project

This research is directly relevant because it shows how decentralized, privacy-preserving multi-factor authentication can be integrated into a Zero Trust architecture; a blueprint for secure identity verification that we can adapt for controlling database access in our Zero Trust Gateway.

8.3.8 Conclusion

This research provides a comprehensive, innovative, and feasible approach to MFA in Zero Trust environments. By using distributed OTP generation, zk-SNARK verification, and non-transferable authentication tokens, it resolves critical weaknesses in traditional MFA. While some limitations exist (cost, setup trust, validator reputation), the overall contribution strongly supports the feasibility of implementing privacy-focused, decentralized identity verification in real-world systems.

8.4 Drivolution: Rethinking the Database Driver Lifecycle

8.4.1 Problem Addressed

Traditional database driver management creates significant operational burdens in large production environments. The study identifies four major challenges:

- **Distribution complexity:** Drivers are distributed separately from database engines, leading to version mismatches and incompatibilities
- **Manual deployment:** Driver installation requires manual operations on each client machine, which doesn't scale well
- **Disruptive upgrades:** Updating drivers requires stopping applications, reconfiguring them, and restarting—causing downtime
- **Security vulnerabilities:** Malicious applications can exploit outdated drivers or use crafted drivers to attack database servers

These issues are amplified in heterogeneous environments where multiple database versions, platforms (63+ for MySQL alone), and client applications coexist. The problem becomes even more acute in replicated database environments where upgrades must account for the Cartesian product of drivers and databases.

8.4.2 Research Methodology

The authors propose and implement Drivolution, an alternative architecture for database driver management. The methodology includes:

- Design of a new driver lifecycle model inspired by OS bootloaders and DHCP protocols
- Implementation for JDBC API integrated with Sequoia database clustering middleware
- Multiple case studies demonstrating real-world applicability
- Performance evaluation in simulated production environments

8.4.3 Drivolution Architecture

The proposed system fundamentally reimagines driver management through several key innovations:

Core Components

- **Driver Storage:** Drivers are stored in the database itself (in regular tables) or in standalone Drivolution servers, treating them as integral parts of the database schema
- **Bootloader:** A small, stable client-side component that rarely needs updating—it downloads and executes driver code from the database
- **Drivolution Protocol:** A DHCP-inspired protocol with three messages (REQUEST, OFFER, ERROR) for driver negotiation
- **Lease System:** Time-limited driver validity periods that enable automatic updates

Technical Implementation

The system uses two key database tables in the information schema:

- **Drivers table:** Stores driver binaries (as BLOBs), API versions, platform specifications, and version information
- **Driver_permission table:** Defines access rights, update policies, lease times, and expiration policies per user/client/database combination

The bootloader intercepts API connection calls, contacts the Drivolution server, downloads appropriate drivers based on client platform and requirements, and dynamically loads them into application memory—all transparently to the application.

8.4.4 Key Innovations

Simplified Lifecycle

Traditional approach (10 steps for upgrade per client):

1. Stop application
2. Uninstall old driver
3. Download new driver package
4. Install driver
5. Configure application
6. Restart application
7. (Plus 4 more verification steps)

Drivolution approach (1 step for all clients):

1. Insert new driver into Drivolution server database

Transparent Updates

Three update policies accommodate different operational needs:

- **AFTER_CLOSE:** Wait for application to close connections naturally
- **AFTER_COMMIT:** Close connections after current transactions complete
- **IMMEDIATE:** Force immediate termination and upgrade

Security Features

- SSL-encrypted, authenticated transfer channels prevent man-in-the-middle attacks
- Digital signature verification ensures driver authenticity
- Standard database access controls limit who can retrieve which drivers
- Centralized management reduces risk of outdated, vulnerable drivers

8.4.5 Case Studies and Results

Heterogeneous DBMS Administration

For DBAs managing multiple database versions, Drivolution reduced:

- Accessing new database: from 6 manual steps to 1 automatic connection
- Driver upgrade: from 6 steps per DBA workstation to 2 centralized operations

Master/Slave Failover

Drivolution enables transparent client reconfiguration during failover:

- Pre-configured drivers (DB_master, DB_slave) distributed based on current topology
- Failover accomplished by marking old driver expired and offering new one
- All clients automatically reconfigure without manual intervention

Sequoia Clustering Middleware

Multiple deployment configurations demonstrated:

- **Standalone server:** Centralized control with hot-standby replication for availability
- **Embedded in controllers:** Drivolution servers replicated across cluster nodes, eliminating single point of failure
- Seamless upgrades of both Sequoia drivers and backend database drivers

Customized Driver Delivery

- **On-demand assembly:** Delivers only required components (e.g., NLS packages for specific languages, GIS extensions only to geographic applications)
- **License management:** Dynamic distribution of per-user licenses (e.g., IBM DB2 licensing model)

8.4.6 Performance Characteristics

- Bootloader overhead: minimal (simple connection interception)
- One-time download per lease period (hours to days)
- No performance impact on queries after driver loaded
- Drivolution server can be replicated for availability without complex consistency requirements (infrequent updates)

8.4.7 Advantages and Contributions

- **Operational simplicity:** Centralized management reduces complexity exponentially in large deployments
- **Zero downtime:** Applications continue running during driver upgrades
- **Version consistency:** Guaranteed compatibility between drivers and databases
- **Security improvement:** Faster deployment of security patches, elimination of forgotten/outdated drivers
- **Legacy compatibility:** Works with existing databases and applications without modifications

- **Platform neutrality:** Single bootloader implementation per API works across all databases
- **Flexibility:** Supports multiple deployment models (in-database, external, standalone service)

8.4.8 Limitations and Considerations

- **Bootloader dependency:** Initial bootloader installation still required (though this is one-time per API/platform)
- **Dynamic loading requirement:** Not all languages/platforms support secure dynamic code loading
- **API stability:** Bootloaders are API-specific; major API changes require new bootloaders
- **Testing discipline:** Ease of updates might tempt skipping rigorous testing (though the paper notes testing is actually easier with short leases for staged rollouts)
- **Drivolution server availability:** Becomes a dependency, though this is mitigated through replication
- **Trust requirements:** Initial bootloader and SSL certificates must be established securely

8.4.9 Relevance to Our Zero Trust Database Access Project

This research provides several valuable insights for our project:

Zero Trust Alignment

- **Continuous verification:** The lease system embodies "never trust, always verify"—clients must regularly re-authenticate to receive driver updates
- **Least privilege:** The driver_permission table enables fine-grained control over which clients access which databases with which drivers
- **Explicit trust zones:** Each client-database interaction is mediated through the Drivolution server, creating an explicit trust boundary

Practical Implementation Lessons

- **Centralized policy enforcement:** Storing access policies in the database (driver_permission table) parallels our need for centralized Zero Trust policy management
- **Transparent security:** The bootloader approach shows how security enhancements can be added without modifying applications—applicable to our gateway design
- **Gradual rollout:** Short initial leases with gradual expansion demonstrate safe deployment of security updates—relevant for our anomaly detection model updates
- **Backward compatibility:** Supporting both Drivolution and legacy connections shows how to introduce Zero Trust incrementally

Architectural Patterns

- **Interception layer:** The bootloader pattern (intercept, verify, mediate) directly parallels our Zero Trust gateway architecture
- **Dynamic configuration:** Delivering pre-configured drivers is analogous to delivering context-aware access policies
- **Lease-based validity:** Time-limited driver validity maps to session tokens with expiration in Zero Trust
- **Distributed servers:** Replicating Drivolution servers across cluster nodes provides a model for high-availability Zero Trust policy enforcement

Security Considerations

- **Encrypted channels:** SSL/TLS requirements reinforce the importance of encrypting all database traffic in Zero Trust
- **Signature verification:** Driver signing parallels the need to verify integrity of all components in our system
- **Audit trails:** The ability to track which drivers were distributed to which clients informs our logging and compliance requirements
- **Attack surface reduction:** Preventing outdated drivers parallels preventing outdated/vulnerable access patterns in Zero Trust

8.4.10 Implications for zGate Proxy Architecture

While Drivolution addresses driver lifecycle management through client-side bootloaders and centralized driver distribution, our Zero Trust gateway requires a fundamentally different architectural approach. The Drivolution model operates as a transparent intermediary that facilitates driver delivery but does not actively mediate database communication protocols.

For zGate proxy to effectively implement Zero Trust principles—including continuous authentication, fine-grained access control, query-level authorization, and real-time anomaly detection—we must adopt a protocol translation architecture. This necessitates implementing native database protocol handlers for each supported database technology (PostgreSQL, MySQL, Oracle, SQL Server, etc.), enabling zGate to function as a bidirectional protocol mediator.

In this architecture, zGate presents itself as a database server to client applications, accepting connections through database-native protocols and performing authentication, authorization, and security checks. Simultaneously, zGate maintains authenticated connection pools to backend database servers, acting as a client that forwards validated queries and returns results. This dual-role architecture provides several critical capabilities:

- **Deep packet inspection:** Full visibility into query content enables semantic analysis and policy enforcement at the query level
- **Protocol-level security:** Independent verification of authentication credentials without relying on client-provided drivers
- **Centralized policy enforcement:** All database traffic traverses zGate, ensuring no requests bypass security controls
- **Connection pooling and optimization:** Backend connection management independent of client connection lifecycle
- **Multi-database support:** Protocol handlers for different database technologies enable consistent security across heterogeneous environments
- **Audit and compliance:** Complete transaction logs captured at the protocol level with full context

The implementation of database-specific protocol handlers represents significant engineering complexity, as each database vendor implements proprietary wire protocols with distinct authentication mechanisms, query formats, and result set encodings. However, this approach is essential for achieving the granular control and visibility required in a

Zero Trust architecture, where trust must be continuously verified and access decisions made based on comprehensive contextual analysis.

8.4.11 Conclusion

Drivolution demonstrates that fundamental database infrastructure components can be redesigned to reduce operational complexity, improve security, and enable zero-downtime operations. The architecture's emphasis on centralized management, transparent operation, and compatibility with legacy systems provides a valuable blueprint for introducing Zero Trust principles into database access patterns. The successful implementation in production middleware (Sequoia) validates that such architectural changes are not merely theoretical but practically achievable in real-world systems.

However, the requirements of Zero Trust security demand more than transparent driver management—they necessitate active protocol mediation. This insight directly motivates the zGate proxy architecture, where implementing backend drivers and connection handlers for each database technology enables the system to serve as an intelligent intermediary, presenting a server interface to clients while maintaining authenticated client connections to database backends. This bidirectional translation capability forms the foundation upon which comprehensive Zero Trust security controls can be built.

8.5 Paper 5

8.6 Paper 6

8.7 Paper 7

8.8 Research References

- **Paper 1:** Generative AI-Enhanced Cybersecurity Framework for Enterprise Data Privacy Management
  https://www.mdpi.com/2073-431X/14/2/55
- **Paper 2:** The Significance of Artificial Intelligence in Zero Trust Technologies: A Comprehensive Review
  https://link.springer.com/article/10.1186/s43067-024-00155-z
- **Paper 3:** Securing Digital Identity in the Zero Trust Architecture: A Blockchain Approach to Privacy-Focused Multi-Factor Authentication
  https://ieeexplore.ieee.org/abstract/document/10505915

- **Paper 4:** Drivolution: Rethinking the Database Driver Lifecycle
  https://www.researchgate.net/publication/43651762_Drivolution_Rethinking_the_Database_Driver_Lifecycle

9. Technical Background

This chapter provides an in-depth technical overview of all technologies, protocols, and architectural concepts forming the foundation of the zGate Zero Trust Database Access Proxy. The discussion is intentionally extensive to support academic rigor and enable future researchers or developers to extend the project.

9.1 Systems Programming in Go

The zGate gateway is implemented entirely in Go (Golang) version 1.25.4. Go is selected due to its strong concurrency model, built-in memory safety guarantees, and first-class support for networked systems.

9.1.1 Introduction to Go Programming Language

Go, also known as Golang, is a statically typed, compiled programming language designed at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. First released publicly in 2009, Go was created to address shortcomings in other languages used for systems programming, particularly in the context of multicore processors, networked systems, and large codebases.

Design Philosophy Go emphasizes simplicity, readability, and pragmatism. Key design principles include:

- **Simplicity**: Minimalist syntax with only 25 keywords
- **Explicit over implicit**: No hidden control flow or magic behaviors
- **Composition over inheritance**: Interfaces and struct embedding instead of class hierarchies
- **Fast compilation**: Designed for rapid build times even in large projects
- **Built-in concurrency**: First-class language support for concurrent programming

Memory Safety Go provides automatic memory management through garbage collection, eliminating entire classes of vulnerabilities:

- **No manual memory management**: Prevents use-after-free and double-free errors
- **Bounds checking**: Array and slice accesses are automatically validated
- **No pointer arithmetic**: Prevents buffer overflows and memory corruption
- **Type safety**: Strong static typing prevents type confusion attacks

Standard Library Go's extensive standard library includes production-ready packages for:

- Network programming (`net`, `net/http`)
- Cryptography (`crypto/*`)
- Encoding/decoding (`encoding/json`, `encoding/xml`)
- Testing and benchmarking (`testing`)
- Concurrent programming (`sync`, `context`)

9.1.2 Go Runtime Model

Go employs a sophisticated user-space thread management architecture based on the G–M–P scheduling model, which enables efficient concurrency without the overhead of traditional OS threads.

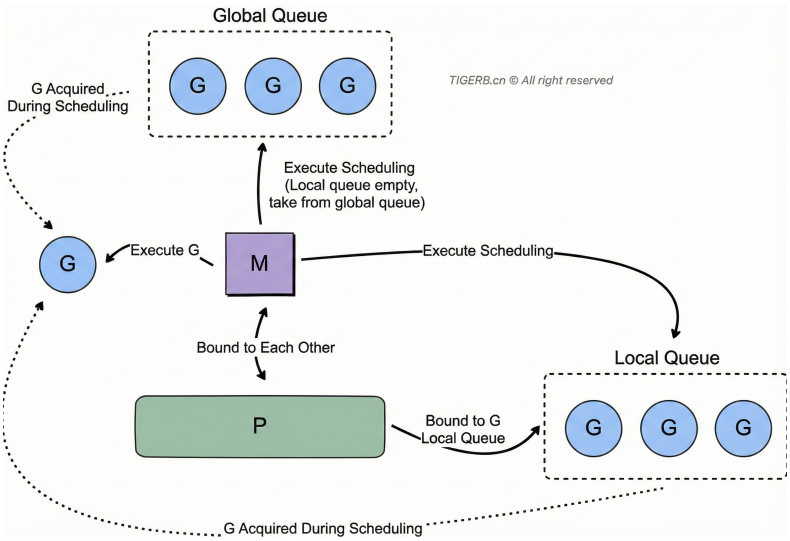


Figure 9.1: Go concurrency execution model illustrating goroutines, OS threads, the Go scheduler (G-M-P model).

The G–M–P Model Explained

- **G (Goroutine)**: A goroutine is a lightweight cooperative thread with a dynamically-sized stack that starts at 2KB and can grow to several megabytes. Unlike OS threads, goroutines are managed entirely in user space by the Go runtime. Goroutines use cooperative scheduling, meaning they yield control at specific points (channel operations, system calls, function calls) rather than being preemptively interrupted.
- **M (Machine)**: An M represents an OS thread managed by the operating system kernel. The Go runtime creates a pool of Ms (typically matching the number of CPU cores) that execute goroutines. When a goroutine performs a blocking system call, the M is detached and a new M may be created to continue executing other goroutines.
- **P (Processor)**: A P is a scheduling context that maintains a local run queue of goroutines. The number of Ps is typically set to the number of available CPU cores (controlled by GOMAXPROCS). Each M must be associated with a P to execute goroutines. When a goroutine blocks, the P can be handed off to another M, allowing other goroutines to continue execution.

Scheduling Mechanism The scheduler implements work-stealing to balance load:

1. Each P maintains a local queue of runnable goroutines
2. When a P's queue is empty, it attempts to steal work from other Ps
3. A global run queue handles goroutines that don't fit in local queues
4. Network poller integration enables efficient I/O multiplexing

Why This Matters for zGate This model enables thousands of goroutines to execute concurrently with negligible overhead. zGate relies on this feature because each client session, backend connection, interceptor callback, and logging pipeline runs as its own goroutine. A typical deployment might handle 10,000+ concurrent database connections, each requiring multiple goroutines, which would be impossible with traditional thread-per-connection models.

9.1.3 Concurrency Primitives

Go provides several built-in primitives for concurrent programming that form the foundation of zGate's concurrent architecture.

Goroutines in Detail Goroutines are created using the `go` keyword followed by a function call. They provide several advantages:

- **Low memory overhead**: Each goroutine starts with only 2KB stack space vs 1-2MB for OS threads
- **Fast creation**: Creating a goroutine takes microseconds vs milliseconds for threads
- **Efficient scheduling**: Context switching between goroutines is faster than kernel thread switches
- **Scalability**: Applications can easily spawn millions of goroutines

In zGate, goroutines are used for:

- **Frontend packet reader**: Continuously reads MySQL packets from client connections
- **Backend packet writer**: Forwards packets to the database server
- **Audit logger**: Asynchronously writes audit entries without blocking query processing
- **TLS handshake worker**: Handles cryptographic handshakes in parallel
- **Interceptor orchestrators**: Executes policy enforcement logic concurrently

Channels in Depth Channels are Go's primary mechanism for communication between goroutines, implementing the CSP (Communicating Sequential Processes) model. Channels are typed, thread-safe queues that can be buffered or unbuffered.

Channel Types:

- **Unbuffered channels**: Synchronous - sender blocks until receiver is ready
- **Buffered channels**: Asynchronous up to buffer size - sender blocks only when buffer is full
- **Directional channels**: Can be send-only (`chan<-`) or receive-only (`<-chan`)

Channel Operations:

- **Send**: `ch <- value`
- **Receive**: `value := <-ch`
- **Close**: `close(ch)`

- **Select**: Multiplexing over multiple channel operations

In zGate, channels provide synchronization for:

- **Session-level error propagation**: When a critical error occurs in any goroutine handling a session
- **Asynchronous event forwarding**: Audit events, metrics, and alerts
- **Administrative operation coordination**: Graceful shutdown, configuration reloads
- **Work distribution**: Distributing query processing tasks across worker pools

Mutexes and Atomic Operations Mutex (Mutual Exclusion): A mutex is a synchronization primitive that protects shared data from concurrent access:

- **sync.Mutex**: Provides exclusive locking - only one goroutine can hold the lock
- **sync.RWMutex**: Reader-writer mutex - allows multiple readers OR one writer
- **Lock/Unlock pattern**: Must be paired, typically using `defer` to ensure unlock

Atomic Operations: The `sync/atomic` package provides lock-free operations for simple data types:

- **Atomic integers**: Add, Load, Store, Swap, CompareAndSwap operations
- **Performance**: Much faster than mutex-based protection for simple counters
- **Memory ordering**: Provides happens-before guarantees

Critical shared resources in zGate use:

- **sync.Mutex / sync.RWMutex**: For metadata caches (user sessions, prepared statements)
- **sync.Once**: For one-time initialization of secrets, certificates, and database connections
- **sync/atomic**: For high-throughput metrics (query counts, error rates, latency tracking)

9.1.4 Low-Level TCP Socket Programming

Unlike typical database clients that rely on high-level drivers, zGate communicates directly using the MySQL wire protocol over raw TCP sockets. This low-level approach provides complete control over the communication pipeline.

TCP/IP Socket Fundamentals **TCP (Transmission Control Protocol):**

- **Connection-oriented**: Requires handshake (SYN, SYN-ACK, ACK) before data transfer
- **Reliable**: Guarantees in-order delivery with automatic retransmission
- **Flow control**: Prevents sender from overwhelming receiver
- **Congestion control**: Adapts sending rate based on network conditions

Socket Operations in Go:

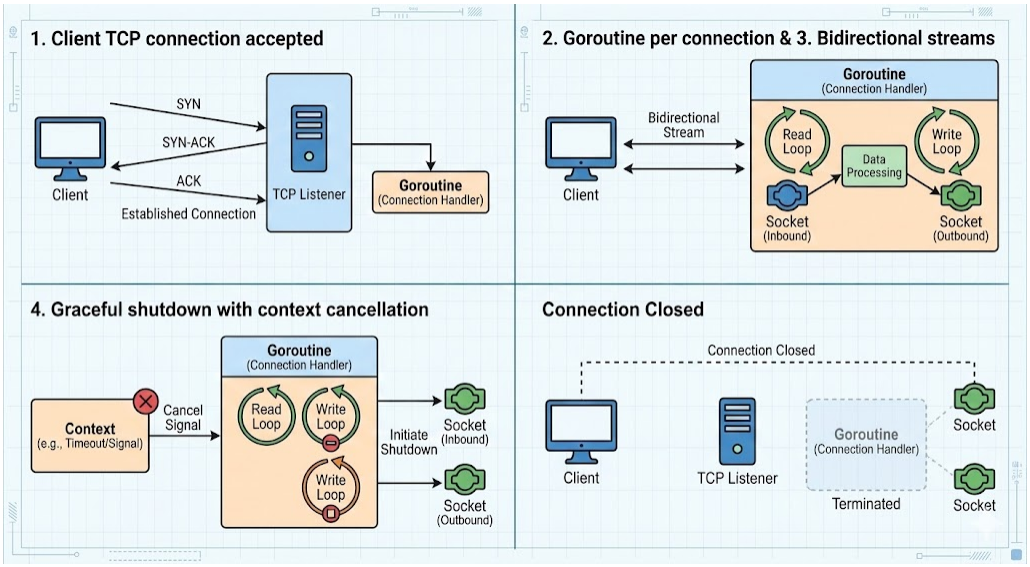


Figure 9.2: Lifecycle of a TCP connection in a Go-based server, showing connection acceptance, goroutine spawning, bidirectional data flow, and graceful shutdown via context cancellation.

- **net.Dial()**: Establishes outbound TCP connection
- **net.Listen()**: Creates listening socket for inbound connections
- **Accept()**: Accepts incoming connection, returns new socket
- **Read()/Write()**: Transfer data over established connection
- **Close()**: Terminates connection gracefully

MySQL Protocol Socket Management Important responsibilities in zGate include:

- **Manual packet header reading**: Every MySQL packet begins with a 4-byte header:
  - Bytes 0-2: Payload length (24-bit little-endian integer, max 16MB)

  - Byte 3: Sequence ID (increments with each packet, wraps at 255)

- **Deadline management**: Network timeouts prevent hung connections:
  - `SetReadDeadline(time.Now().Add(timeout))`: Aborts read if no data arrives
  - `SetWriteDeadline(time.Now().Add(timeout))`: Aborts write if socket buffer is full
  - Deadlines are per-operation, not absolute timeouts
- **Zero-copy packet forwarding**: When packets don't require inspection or modification:
  - Direct buffer passing between client and server sockets
  - Avoids serialization/deserialization overhead
  - Reduces memory allocations and GC pressure
  - Uses `io.Copy()` or `io.CopyN()` for efficient transfer
- **Full connection lifecycle handling**:
  - **Handshake phase**: Initial authentication and capability negotiation
  - **Command phase**: Processing client commands (queries, prepared statements)
  - **Result phase**: Streaming result sets back to client
  - **Cleanup**: Proper resource release on connection termination

Buffer Management Efficient buffer handling is critical for performance:

- **Buffer pools**: Pre-allocated buffers using `sync.Pool` to reduce GC
- **Slice capacity management**: Careful pre-allocation to avoid repeated resizing
- **Memory reuse**: Buffers are reset and returned to pool after use
- **Large packet handling**: Special handling for packets exceeding 16MB (split across multiple packets)

9.1.5 Context Propagation

The `context.Context` package provides a standardized way to carry deadlines, cancellation signals, and request-scoped values across API boundaries and between goroutines.

Context Fundamentals Context Interface:

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}
```

Context Creation Functions:

- **context.Background()**: Root context, never cancelled
- **context.TODO()**: Placeholder when context is unclear
- **context.WithCancel()**: Returns context with cancel function
- **context.WithDeadline()**: Cancels at specific time
- **context.WithTimeout()**: Cancels after duration
- **context.WithValue()**: Carries request-scoped data

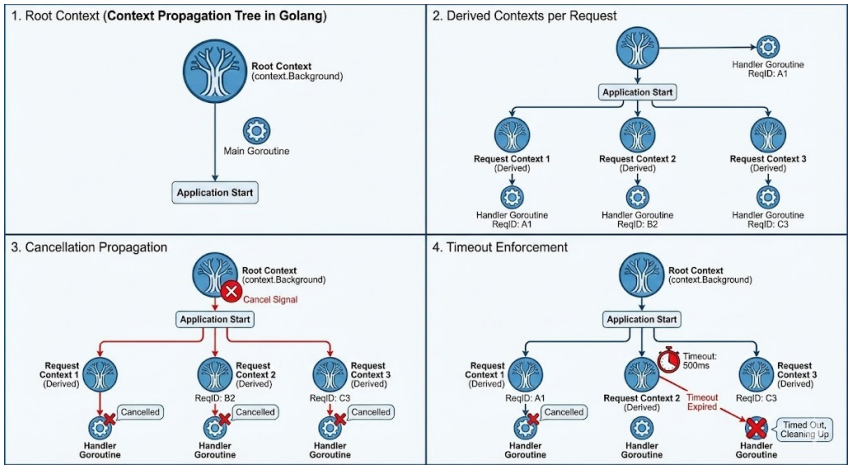


Figure 9.3: Hierarchical propagation of `context.Context` objects across goroutines, demonstrating timeout enforcement and cancellation signaling.

Context in zGate The proxy uses `context.Context` to ensure bounded execution and clean cancellation. Each incoming query obtains a context with:

- **Query ID**: Unique identifier for request tracing and correlation
- **Deadline and timeout settings**:
  - Default query timeout (e.g., 30 seconds)

  - User-specific timeout overrides
  - Inherited from client connection timeout if shorter

- **User identity and role information**:
  - Authenticated username
  - Active role assignments
  - Permission set
  - Session token information

- **Logging metadata**:
  - Source IP address
  - Client application identifier
  - Connection ID
  - Request timestamp

Cancellation Propagation Context cancellation terminates goroutines cleanly, avoiding resource leakage:

1. Client disconnects → context cancelled → all related goroutines notified
2. Query timeout exceeded → context cancelled → database connection interrupted
3. Admin kills session → context cancelled → graceful cleanup initiated
4. Shutdown signal received → root context cancelled → all sessions terminated

Best Practices in zGate:

- Always pass context as first parameter to functions
- Check `ctx.Done()` in long-running loops
- Use `select` to multiplex context cancellation with other operations
- Never store contexts in structs (pass explicitly)
- Defer cancellation function calls to prevent leaks

9.2 Database Wire Protocols (MySQL/MariaDB)

A distinguishing feature of zGate is its ability to ”speak” the MySQL protocol directly, acting as a full proxy rather than a driver or middleware within the application layer.

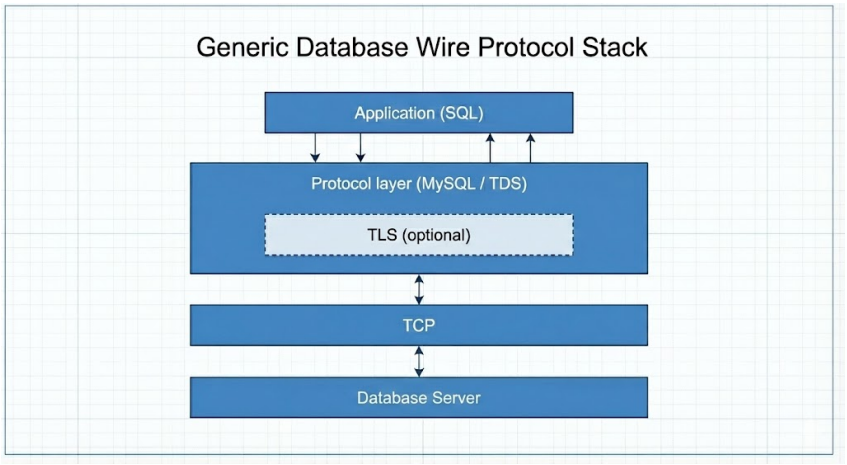


Figure 9.4: Layered view of database communication showing SQL semantics encapsulated within protocol-specific frames over optional TLS and TCP/IP.

9.2.1 MySQL Protocol Overview

The MySQL Client/Server Protocol is a binary protocol used for communication between MySQL clients and servers. It was originally designed in the 1990s and has evolved through multiple versions while maintaining backward compatibility.

Protocol Characteristics

- **Binary protocol**: Data is transmitted in compact binary format, not ASCII
- **Stateful**: Server and client maintain session state across multiple packets
- **Packet-oriented**: All communication happens in discrete packets
- **Sequential**: Packets within a command are numbered sequentially
- **Bidirectional**: Both client and server can initiate certain communications

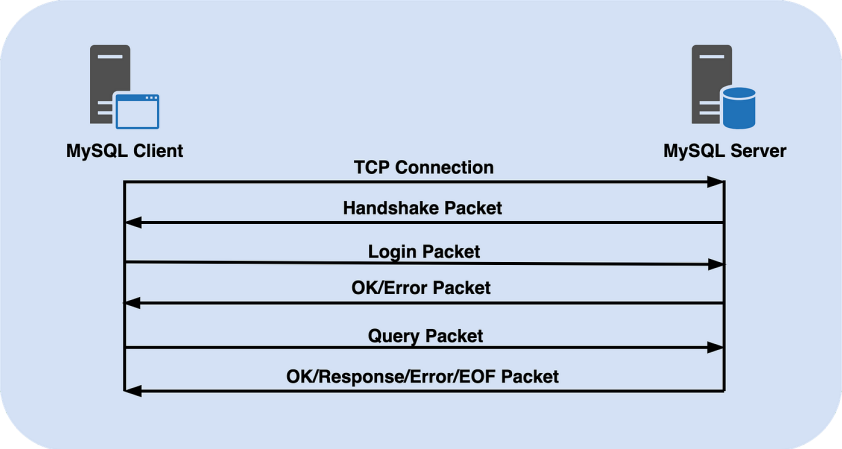


Figure 9.5: Connection and command phases of the MySQL protocol.

Protocol Phases

1. **Connection Phase**:
   - Server sends initial handshake packet with capabilities and auth challenge
   - Client responds with handshake response containing credentials
   - Server sends OK or ERR packet
2. **Command Phase**:
   - Client sends command packets (queries, prepared statements, etc.)
   - Server responds with result sets, OK, or ERR packets
   - Multiple commands can be sent over same connection
3. **Termination Phase**:
   - Client sends COM_QUIT or closes connection
   - Server releases resources and closes socket

9.2.2 MySQL Packet Structure

Each MySQL packet contains a precisely defined structure that must be correctly parsed and reconstructed by zGate.

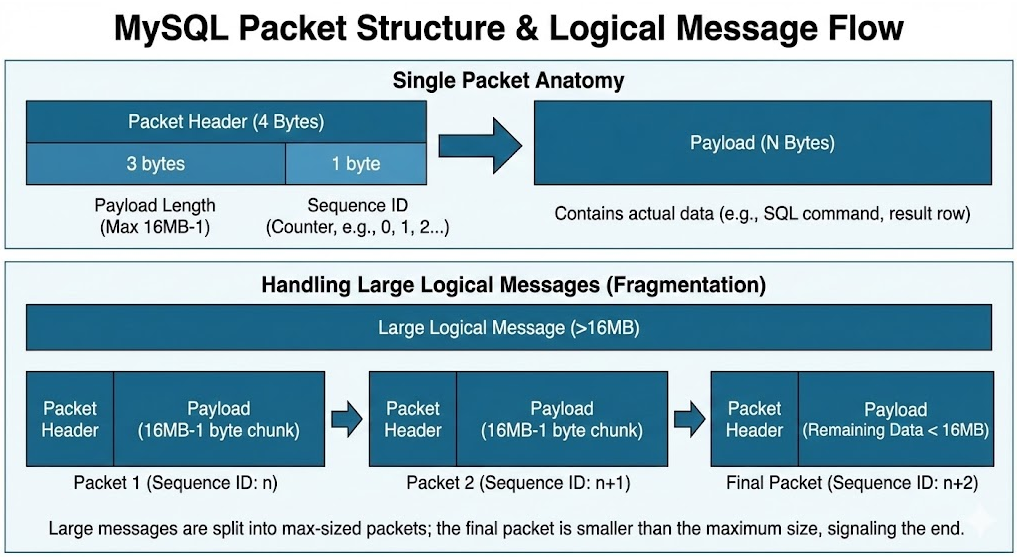


Figure 9.6: Structure of a MySQL protocol packet including payload length, sequence identifier, and payload data.

Packet Header Format

- **3-byte payload length** (little-endian):
  - Represents length of packet payload in bytes
  - Maximum value: 0xFFFFFF (16,777,215 bytes = 16MB - 1)
  - Length does NOT include the 4-byte header itself
  - Packets exceeding 16MB-1 are split into multiple packets
- **1-byte sequence ID**:
  - Starts at 0 for each new command
  - Increments by 1 for each packet in the sequence
  - Wraps around after 255 (rare in practice)
  - Used to detect packet loss or out-of-order delivery
  - Server and client must maintain synchronized counters
- **N-byte payload**:
  - Actual packet content (commands, result data, etc.)
  - Format depends on packet type
  - Can contain binary or text data

Large Packet Handling When payload exceeds 16MB-1 bytes:

1. First packet contains maximum payload (0xFFFFFF bytes)
2. Sequence ID increments for continuation packet(s)
3. Last packet contains remaining data (length < 0xFFFFFF)
4. Empty packet (length=0) sent if payload is exact multiple of 16MB-1

Packet Validation in zGate The gateway must validate and reconstruct packets precisely:

- **Length validation**: Ensure payload length matches actual bytes read
- **Sequence synchronization**: Track and validate sequence IDs on both sides
- **Buffer management**: Allocate appropriate buffers based on payload length
- **Error detection**: Identify corrupted or malformed packets
- **Large packet assembly**: Correctly reassemble multi-packet messages

9.2.3 Command Phase Processing

The MySQL protocol defines numerous command types that clients can send to servers. zGate must understand and properly handle each command type.

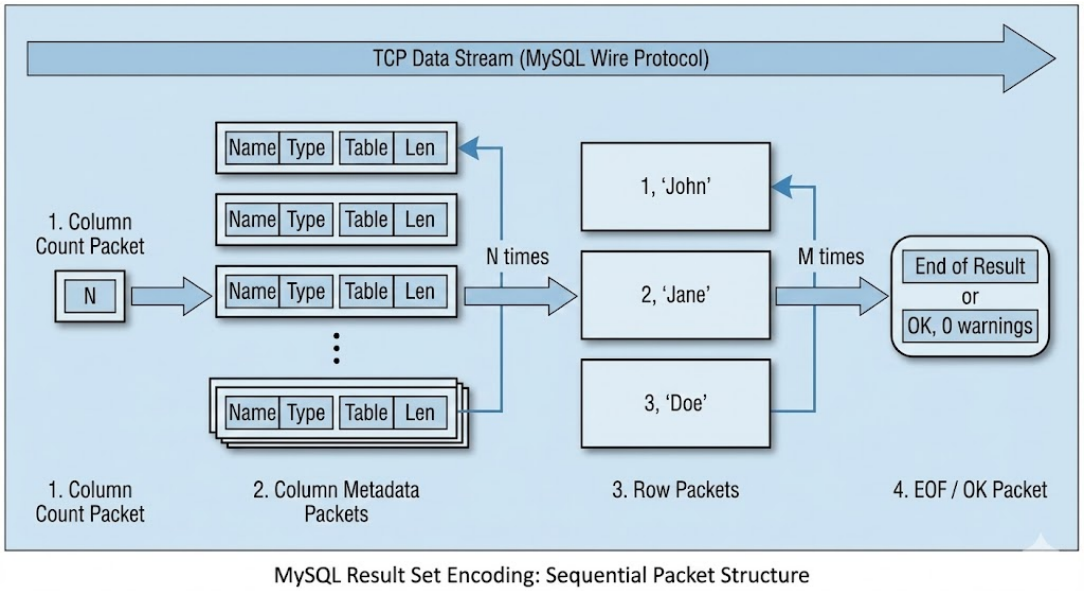


Figure 9.7: Logical structure of a MySQL result set including column count, column metadata packets, row packets, and termination packets.

Command Packet Format

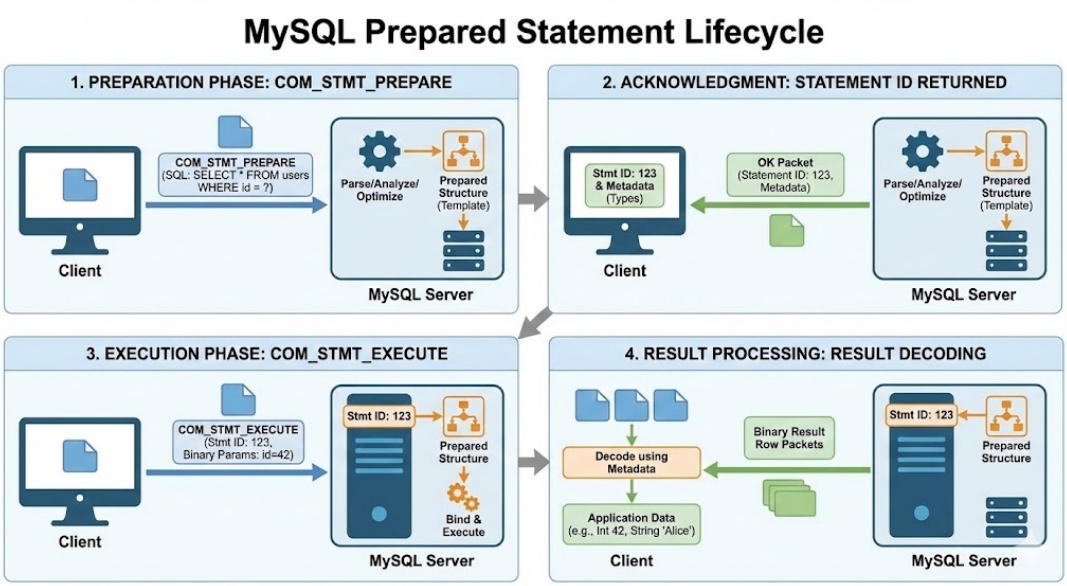


Figure 9.8: Lifecycle of a MySQL prepared statement showing SQL transmission during preparation and binary parameter binding during execution.

Supported Commands in zGate The proxy supports various MySQL commands:

- **COM_QUERY (0x03)**: Execute textual SQL statement
  - Payload: SQL string (NOT null-terminated)
  - Most common command type
  - Response: Result set, OK, or ERR packet
  - Example: `SELECT * FROM users WHERE id=1`
- **COM_INIT_DB (0x02)**: Change default database
  - Payload: Database name string
  - Similar to SQL: `USE database_name`
  - Response: OK or ERR packet
  - Updates session state in both client and server
- **COM_STMT_PREPARE (0x16)**: Prepare SQL statement
  - Payload: SQL statement with ? placeholders
  - Server parses and returns statement ID and parameter metadata
  - Enables binary protocol for faster execution
  - Statement cached on server until explicitly closed
- **COM_STMT_EXECUTE (0x17)**: Execute prepared statement

  - Payload: Statement ID + flags + parameter values (binary encoded)
  - Uses efficient binary protocol instead of text
  - Parameters sent as native types (integers, dates, etc.)
  - Response: Result set in text or binary protocol
- **COM_STMT_CLOSE (0x19)**: Deallocate prepared statement
  - Payload: Statement ID (4 bytes)
  - No response packet (fire-and-forget)
  - Frees server resources
- **COM_QUIT (0x01)**: Close connection
  - No payload
  - No response from server
  - Graceful connection termination
- **COM_PING (0x0E)**: Test connection liveness
  - No payload
  - Response: OK packet
  - Used for keepalive and health checks
- **COM_FIELD_LIST (0x04)**: List table columns (deprecated)
- **COM_STMT_RESET (0x1A)**: Reset prepared statement
- **COM_SET_OPTION (0x1B)**: Set connection options

Command Processing in zGate Each command type has unique processing requirements:

1. **Parse command byte**: Identify command type from first payload byte
2. **Extract command payload**: Read remaining packet data
3. **Policy enforcement**: Check RBAC permissions for this command
4. **SQL rewriting**: Modify query if needed (COM_QUERY only)
5. **Forward to backend**: Send modified or original packet to database
6. **Process response**: Intercept and potentially modify result
7. **Audit logging**: Record command execution details

9.2.4 Result Set Encoding

Result sets in MySQL protocol follow a specific multi-packet format that zGate must parse to enable data masking and filtering.

Result Set Packet Sequence A typical result set consists of:

1. **Column count packet**:
    - Length-encoded integer indicating number of columns
    - Example: 0x03 indicates 3 columns in result
2. **Column definition packets** (one per column):
    - Catalog name (usually ”def”)
    - Schema (database) name
    - Table alias and original table name
    - Column alias and original column name
    - Character set encoding
    - Column display width
    - Column type (INT, VARCHAR, DATE, etc.)
    - Column flags (NOT NULL, PRIMARY KEY, etc.)
    - Decimal precision (for numeric types)
3. **EOF packet** (if CLIENT_DEPRECATE_EOF not set):
    - Marks end of column definitions
    - Contains server status flags and warning count
4. **Row data packets**:
    - One packet per row
    - Values encoded as length-encoded strings (text protocol) or native types (binary protocol)
    - NULL represented as 0xFB byte in text protocol
5. **EOF or OK packet**:
    - Marks end of result set
    - Contains affected rows, last insert ID, status flags, warnings

Length-Encoded Integers MySQL uses a variable-length encoding for integers:

- **Value $<$ 251**: Single byte containing the value
- **Value = 251 (0xFB)**: NULL value marker
- **Value = 252 (0xFC)**: Next 2 bytes contain value (little-endian)
- **Value = 253 (0xFD)**: Next 3 bytes contain value (little-endian)
- **Value = 254 (0xFE)**: Next 8 bytes contain value (little-endian)

Text vs Binary Protocol **Text Protocol (COM_QUERY):**

- All values encoded as strings
- Dates, numbers, etc. formatted as text
- Less efficient but human-readable

Binary Protocol (COM_STMT_EXECUTE):

- Values in native binary format
- NULL bitmap at start of row
- Type-specific encoding (4-byte int, 8-byte double, etc.)
- More efficient, less parsing overhead

zGate Row Interception zGate intercepts row-level data for masking and policy enforcement, requiring:

- **Parsing length-encoded integers**: To determine string/value lengths
- **Reconstructing text and binary rows**: After applying masking rules
- **Preserving column metadata**: To understand data types for correct parsing
- **Maintaining packet integrity**: Recalculating payload lengths and sequence IDs
- **Streaming processing**: Handling large result sets without buffering all rows

Example Masking Scenario:

1. Client queries: `SELECT ssn, name FROM employees`
2. zGate parses column definitions, identifies ”ssn” column

3. For each row packet:
    - Parse length-encoded string for ssn value
    - Apply masking: "123-45-6789" → "XXX-XX-6789"
    - Reconstruct row packet with masked value
    - Recalculate payload length
    - Forward modified packet to client

9.2.5 SQL Parsing and AST Manipulation

To safely inject or modify SQL logic, zGate uses an SQL Abstract Syntax Tree (AST) approach rather than string manipulation.

SQL Abstract Syntax Trees An Abstract Syntax Tree is a tree representation of the syntactic structure of SQL code. Each node in the tree represents a construct in the SQL grammar.

Why AST vs String Manipulation:

- **Semantic understanding**: Parser understands SQL structure, not just text
- **Safe modification**: Changes preserve syntax validity
- **SQL injection prevention**: Prevents introduction of new SQL commands
- **Context awareness**: Distinguishes between identifiers, literals, keywords
- **Complexity handling**: Correctly processes nested queries, subqueries, CTEs

Parsing Pipeline

1. **Lexical Analysis (Tokenization)**:
    - Input: Raw SQL string
    - Process: Break into tokens (keywords, identifiers, operators, literals)
    - Output: Token stream
    - Example: `SELECT name FROM users` → [SELECT][name][FROM][users]
2. **Syntax Analysis (Parsing)**:
    - Input: Token stream
    - Process: Apply grammar rules to build AST
    - Output: AST root node

- Detects syntax errors

3. **AST Manipulation**:
   - Traverse tree using visitor pattern
   - Inspect and modify nodes
   - Add new nodes (e.g., WHERE clauses)
   - Remove or replace nodes
4. **Code Generation (Serialization)**:
   - Input: Modified AST
   - Process: Traverse tree and generate SQL text
   - Output: Valid SQL string
   - Preserves formatting where possible

AST Node Types Common node types in SQL AST:

- **SelectStmt**: SELECT query root
- **SelectExprList**: Target list (columns to return)
- **FromClause**: Table references
- **WhereClause**: Filter conditions
- **JoinExpr**: JOIN operations
- **BinaryExpr**: Binary operations (AND, OR, =, <, etc.)
- **FuncCall**: Function calls (COUNT, MAX, etc.)
- **Identifier**: Table and column names
- **Literal**: String, numeric, date literals
- **Subquery**: Nested SELECT statement

AST Manipulation in zGate zGate performs node-by-node inspection for:

- **Target list modification**:
  - Removing restricted columns from SELECT list
  - Adding computed columns for audit purposes
  - Replacing sensitive columns with masked expressions
  - Example: `SELECT ssn` → `SELECT CONCAT('XXX-XX-', SUBSTR(ssn, 8)) AS ssn`
- **WHERE clause enforcement**:
  - Injecting row-level security predicates
  - Adding tenant isolation filters
  - Enforcing time-based access controls
  - Example: User can only see their own records
  - Original: `SELECT * FROM orders`
  - Modified: `SELECT * FROM orders WHERE user_id = 'alice'`
- **Table-level permission checks**:
  - Identify all referenced tables
  - Verify user has access to each table
  - Check for column-level permissions
  - Reject queries accessing forbidden tables
- **Subquery processing**:
  - Recursively process nested queries
  - Apply same policies to subqueries
  - Handle correlated subqueries correctly
- **JOIN analysis**:
  - Check permissions on all joined tables
  - Inject filters on joined tables if needed
  - Prevent information leakage through joins

Safe Rewriting Examples **Example 1: Column Masking**

```
Original SQL: SELECT ssn, name, salary FROM employees
AST Modification:
  - Locate "ssn" in SelectExprList
  - Replace with FuncCall node: mask_ssn(ssn)
Generated SQL: SELECT mask_ssn(ssn), name, mask_salary(salary)
               FROM employees
```

Example 2: Row Filtering

```
Original SQL: SELECT * FROM medical_records
AST Modification:
  - Navigate to WhereClause (or create if absent)
  - Add BinaryExpr: department = 'cardiology'
Generated SQL: SELECT * FROM medical_records
               WHERE department = 'cardiology'
```

Example 3: Table Access Control

```
Original SQL: SELECT * FROM hr.salaries
AST Modification:
  - Traverse tree, find table reference "hr.salaries"
  - Check user's table permissions
  - If denied: Return error before forwarding to database
  - If allowed: Forward unmodified or with row filters
```

Security Benefits This AST approach avoids string-based manipulation, preventing:

- **SQL injection vulnerabilities**: Cannot introduce new SQL commands
- **Malformed query errors**: Generated SQL is always syntactically valid
- **Escaping issues**: No need to manually escape quotes and special characters
- **Context confusion**: Parser understands string literals vs identifiers
- **Logic errors**: Changes preserve query semantics

9.3 Database Wire Protocols (Microsoft SQL Server - TDS)

In addition to MySQL and MariaDB, modern enterprise environments frequently rely on Microsoft SQL Server. SQL Server communicates using the Tabular Data Stream

(TDS) protocol, a proprietary, binary, application-layer protocol designed by Microsoft. Understanding TDS is essential for building protocol-aware proxies, intrusion detection systems, or database firewalls that operate transparently at the network level.

Unlike MySQL, which exposes relatively straightforward packet structures, TDS is more complex, stateful, and tightly coupled with SQL Server's execution engine.

9.3.1 Overview of the TDS Protocol

Tabular Data Stream (TDS) is a message-oriented protocol layered directly on top of TCP. It defines how SQL Server clients and servers exchange authentication data, SQL batches, procedure calls, metadata, and result sets.

Key Characteristics

- **Binary protocol**: All messages are encoded in binary, not text
- **Token-based**: Responses consist of a stream of typed tokens
- **Stateful**: Session context persists across requests
- **Versioned**: Multiple protocol versions (TDS 4.2 – 7.4+)
- **Tightly integrated**: Closely coupled to SQL Server execution semantics

Protocol Versions Common TDS versions include:

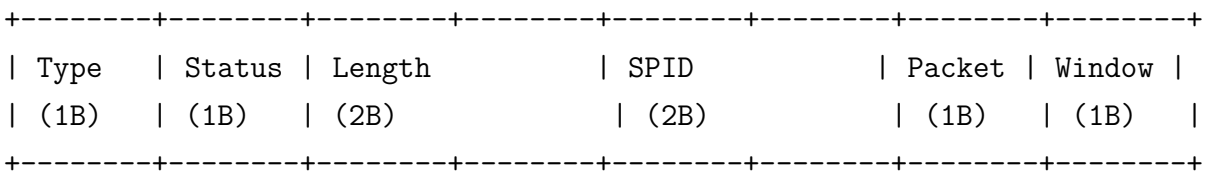
- **TDS 4.2**: Legacy Sybase / early SQL Server
- **TDS 7.0**: SQL Server 7.0
- **TDS 7.1**: SQL Server 2000
- **TDS 7.2**: SQL Server 2005
- **TDS 7.3**: SQL Server 2008 / 2012
- **TDS 7.4+**: SQL Server 2014+

Modern SQL Server installations primarily use TDS 7.4.

9.3.2 TDS Packet Structure

All TDS communication occurs as a sequence of packets, each with a fixed-size header followed by a variable-length payload.

TDS Packet Header



- **Type (1 byte)**:
  - Identifies message category
  - Examples:
    - 0x01 – SQL Batch
    - 0x02 – Pre-login
    - 0x04 – RPC
    - 0x10 – Login
    - 0x12 – Response
- **Status (1 byte)**:
  - Bit flags indicating packet role
  - End-of-message (EOM) flag
  - Indicates whether more packets follow
- **Length (2 bytes)**:
  - Total packet length including header
  - Big-endian integer
  - Max size typically 4KB or 8KB (configurable)
- **SPID (2 bytes)**:
  - Server Process ID
  - Identifies server-side session
- **Packet ID (1 byte)**:
  - Sequence number for packet ordering
- **Window (1 byte)**:
  - Legacy field (unused in modern TDS)

Packet Fragmentation If a message exceeds the negotiated packet size:

- The message is split across multiple TDS packets
- Status byte marks continuation or end-of-message
- Receiver must reassemble payload before processing

9.3.3 Pre-Login and Login Process

Before authentication, TDS performs a pre-login negotiation phase.

Pre-Login Phase

The pre-login packet negotiates session parameters:

- Encryption support (required, optional, disabled)
- TDS protocol version
- Packet size
- Instance name
- Thread ID

The server responds with its supported options. If encryption is required or requested, TLS negotiation begins immediately after pre-login.

TLS Integration Unlike MySQL, SQL Server embeds TLS negotiation inside the TDS flow:

- TLS handshake occurs after pre-login
- All subsequent TDS packets are encrypted
- Proxy must detect transition from plaintext to TLS
- Requires full TLS interception or passthrough

Login Phase

The LOGIN7 packet contains authentication information:

- Username
- Password (obfuscated, not encrypted unless TLS active)
- Database name
- Client hostname
- Application name
- Language and collation

Password Obfuscation Without TLS:

- Password bytes are XOR-obfuscated
- Each byte rotated and XORed with 0xA5
- This is **not encryption** and offers no real security

Therefore, TLS is mandatory in secure deployments.

9.3.4 TDS Message Types

TDS supports multiple message categories:

- **SQL Batch**:
  - Contains raw SQL text
  - Similar to MySQL COM_QUERY
  - Executed as a single batch
- **RPC (Remote Procedure Call)**:
  - Used for stored procedure execution
  - Includes procedure name or ID
  - Parameters encoded in binary
- **Attention**:
  - Cancels currently executing query
- **Bulk Load**:
  - High-speed data import
  - Streams row data efficiently

9.3.5 TDS Token-Based Response Model

Unlike MySQL's fixed result-set structure, TDS responses consist of a stream of typed tokens.

Token Stream Concept A server response is a sequence of tokens:

- Each token begins with a 1-byte token type
- Followed by token-specific payload
- Tokens are processed sequentially

Common TDS Tokens

- **COLMETADATA**:
  - Column count
  - Column names
  - Data types
  - Precision, scale, collation
- **ROW**:
  - Row data in binary format
  - Values encoded according to column metadata
- **DONE / DONEPROC / DONEINPROC**:
  - Indicates completion of batch or procedure
  - Includes affected row count
  - Includes status flags
- **ERROR**:
  - Error number
  - Severity
  - Message text
  - Line number
- **INFO**:
  - Informational messages
  - PRINT output
  - Warnings

9.3.6 Data Type Encoding

SQL Server uses strongly typed binary encodings.

Fixed-Length Types

- INT: 4 bytes
- BIGINT: 8 bytes
- FLOAT: 8 bytes
- BIT: 1 byte

Variable-Length Types

- VARCHAR / NVARCHAR:
  - Length prefix
  - UTF-16LE encoding for NVARCHAR
- VARBINARY:
  - Length-prefixed byte array

NULL Representation

- NULL values represented by length = 0xFFFF or special markers
- Requires metadata awareness to parse correctly

9.3.7 Implications for Proxy Design

Building a TDS-aware proxy introduces significant complexity:

- Stateful token stream parsing
- Precise metadata tracking
- Binary data manipulation
- TLS-in-band protocol switching
- Procedure call interception
- Multi-result-set handling

Comparison with MySQL

- MySQL: Command-based, simpler framing
- TDS: Token-stream-based, execution-aware
- MySQL: Text-heavy protocol
- TDS: Fully binary and metadata-driven

This complexity makes TDS significantly harder to intercept, modify, or rewrite safely, and explains why many database security solutions operate only at the SQL Server driver or API layer rather than at the wire protocol level.

9.3.8 Challenges of Abstract Syntax Tree (AST) Parsing in TDS

While parsing raw TDS packets enables visibility into SQL Server traffic, constructing an Abstract Syntax Tree (AST) from intercepted queries presents a significantly deeper technical challenge. AST parsing is required for fine-grained policy enforcement, such as restricting specific SQL operations, table access, or predicate-level filtering.

Unlike traditional SQL parsing from application logs, TDS introduces several protocol-level obstacles that complicate AST construction.

Absence of a Canonical SQL Text Representation

In TDS, SQL commands may be transmitted in different forms:

- Raw SQL batches (SQL Batch packets)
- Remote Procedure Calls (RPCs)
- Parameterized prepared statements

In the case of RPC execution:

- SQL text may not be present at all
- Only a procedure identifier and binary-encoded parameters are transmitted
- Logical intent must be inferred from metadata and procedure definitions

This breaks the assumption that a proxy always has access to a full SQL string suitable for lexical analysis.

Binary Parameter Encoding and Late Binding

TDS encodes parameters in binary form according to their SQL Server data types. Unlike text-based SQL where literals appear inline, TDS separates query structure from values.

Consequences include:

- Parameters are position-based, not name-based
- Parameter types are defined by metadata tokens
- Values are resolved only at execution time

As a result, an AST parser must:

- Reconstruct the query template
- Bind parameters to placeholders
- Track type information across token boundaries

This is closer to compiler intermediate representation reconstruction than traditional SQL parsing.

Multiple Result Sets and Control Flow Semantics

SQL Server allows complex batches containing:

- Multiple SELECT statements
- Conditional logic (IF, WHILE)
- Temporary tables
- Stored procedure calls returning nested result sets

In TDS, these constructs produce:

- Interleaved COLMETADATA and ROW tokens
- DONE, DONEPROC, and DONEINPROC tokens signaling execution stages

From an AST perspective:

- Execution flow is non-linear
- Query boundaries are not explicitly marked
- Semantic meaning emerges only after full token stream analysis

This requires stateful parsing and execution-context tracking.

Encrypted Sessions and Visibility Loss

Once TLS is negotiated:

- All TDS payloads are encrypted
- AST parsing becomes impossible without TLS termination

This introduces architectural trade-offs:

- TLS passthrough preserves security but eliminates visibility
- TLS interception enables parsing but expands the trusted computing base

From a technical standpoint, AST parsing in TDS is inseparable from cryptographic session management.

Lack of Official Grammar Specifications

Microsoft does not provide a complete, formal SQL grammar aligned with TDS execution semantics.

Challenges include:

- SQL Server-specific extensions (TOP, MERGE, APPLY)
- Version-dependent behavior
- Undocumented token combinations

Therefore, AST parsers must rely on:

- Reverse engineering
- Empirical testing
- Partial grammars adapted from T-SQL documentation

This increases maintenance complexity and version fragility.

Implications for Security Policy Enforcement

Due to the challenges above, enforcing policies at the AST level in TDS requires:

- Hybrid parsing strategies (syntax + metadata)
- Conservative fallbacks (deny on ambiguity)
- Awareness of false positives and negatives

This explains why many commercial database security products:

- Operate at the driver level
- Rely on SQL Server auditing APIs
- Avoid full wire-level AST parsing

9.4 Cryptography and Security Engineering

Security is the cornerstone of the system. The implementation combines modern cryptographic standards with secure design principles to protect data in transit, at rest, and during processing.

9.4.1 TLS / SSL Transport Layer Security

Transport Layer Security (TLS) and its predecessor Secure Sockets Layer (SSL) are cryptographic protocols designed to provide secure communication over a computer network.

TLS Protocol Overview Historical Context:

- **SSL 1.0**: Never publicly released (Netscape, 1994)
- **SSL 2.0**: Released 1995, deprecated 2011 (security flaws)
- **SSL 3.0**: Released 1996, deprecated 2015 (POODLE attack)
- **TLS 1.0**: Released 1999, deprecated 2020
- **TLS 1.1**: Released 2006, deprecated 2020
- **TLS 1.2**: Released 2008, still widely used
- **TLS 1.3**: Released 2018, current standard

Security Properties:

- **Confidentiality**: Data encrypted using symmetric cipher
- **Integrity**: MAC (Message Authentication Code) prevents tampering
- **Authentication**: X.509 certificates verify server/client identity
- **Forward secrecy**: Session keys not derivable from long-term keys

TLS Handshake Process The TLS handshake establishes a secure session:

1. **Client Hello**:
    - Supported TLS versions
    - Cipher suites (encryption algorithms)
    - Random nonce
    - Supported extensions
2. **Server Hello**:
    - Selected TLS version
    - Selected cipher suite
    - Server random nonce
    - Server certificate (X.509)
3. **Key Exchange**:
    - Client verifies server certificate
    - Ephemeral Diffie-Hellman key exchange (TLS 1.3)
    - Or RSA key exchange (TLS 1.2)
    - Derive master secret
4. **Finished Messages**:
    - Both sides send encrypted "Finished" message
    - Proves they derived same keys
    - Handshake complete, application data can flow

TLS in zGate TLS is used to secure:

- **Client–Gateway communication**:
    - MySQL clients connect via TLS-encrypted connections
    - Optional mutual TLS (mTLS) for client authentication
    - Certificate-based authentication instead of passwords
- **Gateway–Database communication**:
    - Backend connections always use TLS
    - Verifies database server certificate

  - Protects credentials and query data in transit

- **Administrative API (HTTPS)**:
  - REST API served over HTTPS
  - Admin credentials encrypted in transit
  - API tokens protected from network sniffing

Technical Implementation Details Technical aspects in zGate include:

- **Loading X.509 certificates from PEM**:
  - PEM (Privacy Enhanced Mail) format: Base64-encoded DER certificates
  - Private keys protected with passphrase encryption
  - Certificate chain loading (intermediate + root CAs)
  - Key pair validation (public key matches private key)
- **Enforcing TLS 1.2+**:
  - Minimum version set in `tls.Config`
  - Rejects connections using SSL 3.0, TLS 1.0, TLS 1.1
  - Prevents downgrade attacks
- **Certificate chain validation**:
  - Using Go's `crypto/x509` package
  - Verifies certificate signature chain to trusted root CA
  - Checks expiration dates
  - Validates hostname/IP against certificate SAN (Subject Alternative Names)
  - Checks certificate revocation status (OCSP or CRL)
- **Cipher suite selection**:
  - Prefer AEAD ciphers (AES-GCM, ChaCha20-Poly1305)
  - Disable weak ciphers (RC4, 3DES, export ciphers)
  - Enable perfect forward secrecy (ECDHE key exchange)
  - Example strong cipher: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- **Optional client certificate verification (mTLS)**:
  - Server requests client certificate during handshake

  - Client presents certificate signed by trusted CA
  - Server validates certificate and extracts identity (CN or SAN)
  - Used for passwordless authentication
  - Common in service-to-service authentication

9.4.2 Symmetric Encryption (AES-256)

Advanced Encryption Standard (AES) is a symmetric block cipher adopted as a standard by NIST in 2001, replacing the older DES algorithm.

AES Fundamentals Algorithm Properties:

- **Block cipher**: Encrypts fixed-size blocks (128 bits)
- **Symmetric**: Same key for encryption and decryption
- **Key sizes**: 128, 192, or 256 bits
- **Rounds**: 10 (AES-128), 12 (AES-192), 14 (AES-256)
- **Speed**: Hardware-accelerated on modern CPUs (AES-NI instructions)

AES Operations:

1. **SubBytes**: Substitute each byte using S-box
2. **ShiftRows**: Rotate rows of state array
3. **MixColumns**: Linear transformation of columns
4. **AddRoundKey**: XOR with round key

AES Modes of Operation Block ciphers require a ”mode” to encrypt data longer than one block:

- **ECB (Electronic Codebook)**: INSECURE, never use
  - Each block encrypted independently
  - Identical plaintexts produce identical ciphertexts
  - Reveals patterns in data
- **CBC (Cipher Block Chaining)**: Legacy, requires padding
  - Each block XORed with previous ciphertext

  - Requires padding to block boundary
  - Vulnerable to padding oracle attacks if not careful
  - Needs separate MAC for authentication
- **GCM (Galois/Counter Mode)**: Recommended
  - AEAD (Authenticated Encryption with Associated Data)
  - Provides both confidentiality and authenticity
  - No padding required
  - Parallelizable (fast)
  - Includes authentication tag to detect tampering
- **CCM, EAX, OCB**: Alternative AEAD modes

AES-256-GCM in zGate The internal SQLite datastore is encrypted using AES-256-GCM. The system uses:

- **32-byte (256-bit) encryption keys**:
  - Derived from master password using key derivation function
  - Or generated randomly for data-at-rest encryption
  - Stored in secure key management system or HSM
  - Never logged or exposed in API responses
- **CSPRNG-generated nonces**:
  - Nonce (Number Used Once) = Initialization Vector (IV)
  - 12 bytes (96 bits) for GCM mode
  - Must be unique for every encryption operation with same key
  - Generated using cryptographically secure random number generator
  - Stored alongside ciphertext (not secret)
  - Nonce reuse catastrophically breaks GCM security
- **Authentication tag**:
  - 16-byte tag appended to ciphertext
  - Verifies data integrity and authenticity
  - Prevents tampering and bit-flipping attacks

  - Decryption fails if tag doesn't match

- **Associated Data (AD)**:
  - Additional authenticated but unencrypted data
  - Example: database record ID, timestamp, version
  - Binds ciphertext to specific context
  - Prevents ciphertext from being moved/reused elsewhere

- **Key rotation support**:
  - Periodic key changes (e.g., quarterly)
  - Re-encrypt data with new keys
  - Multiple active keys during transition period
  - Track which key version encrypted each record
  - Old keys archived securely for decryption of historical data

10. System Architecture

10.1 High-Level Architecture Diagram

10.2 Main System Components

10.3 Component Communication Flow

10.4 Tech Stack Summary

11. Detailed Architecture of the Proxy

Overview

The zGate proxy is the core component that provides secure, zero-trust database access through dynamic port allocation and temporary user management. It acts as an intelligent intermediary between users and backend databases, enforcing policy, managing credentials, and monitoring all database traffic.

Key Components

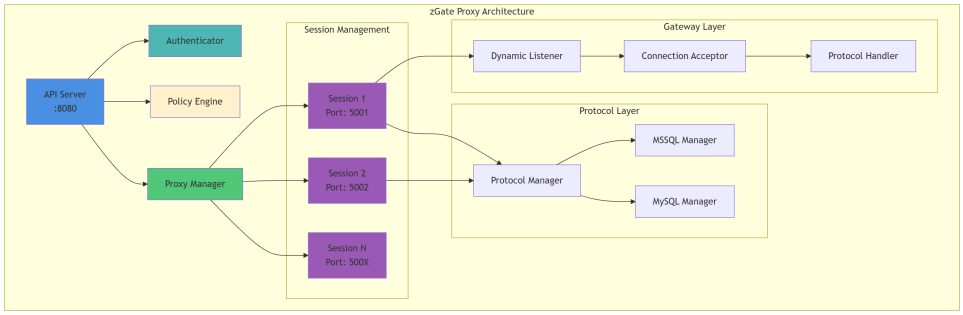


Figure 11.1: system components

11.1 Connection Lifecycle

The connection lifecycle encompasses the complete journey from user authentication to database disconnection, including all intermediate states and transitions.

11.1.1 Lifecycle States

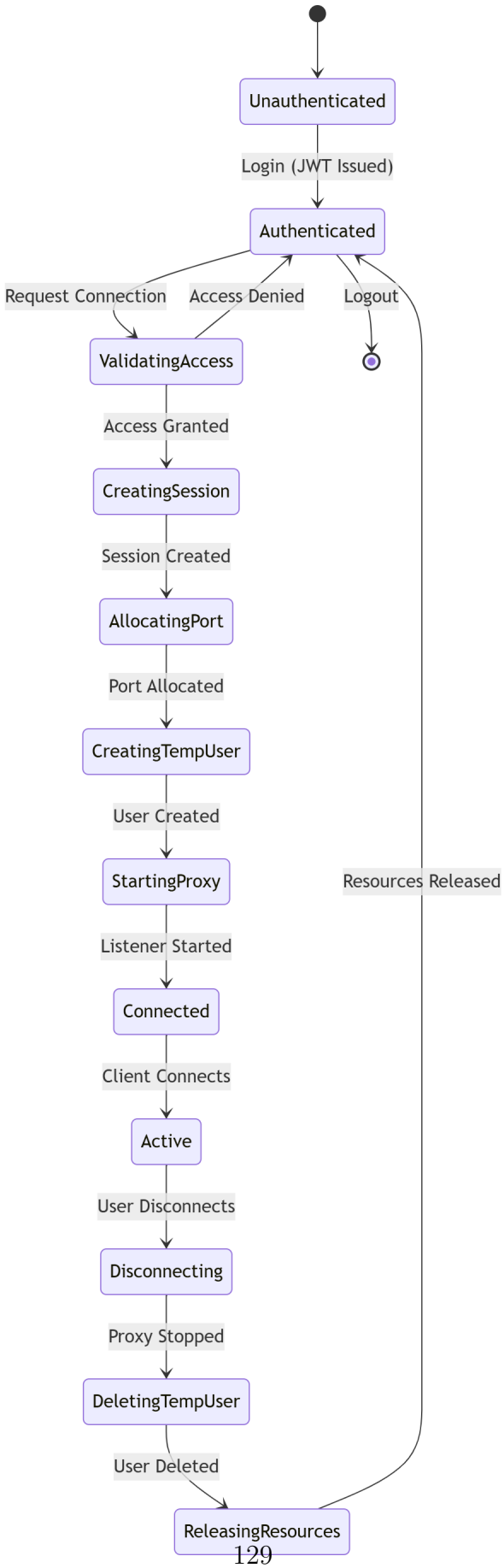


Figure 11.2: connection lifecycle states

11.1.2 Detailed Lifecycle Flow

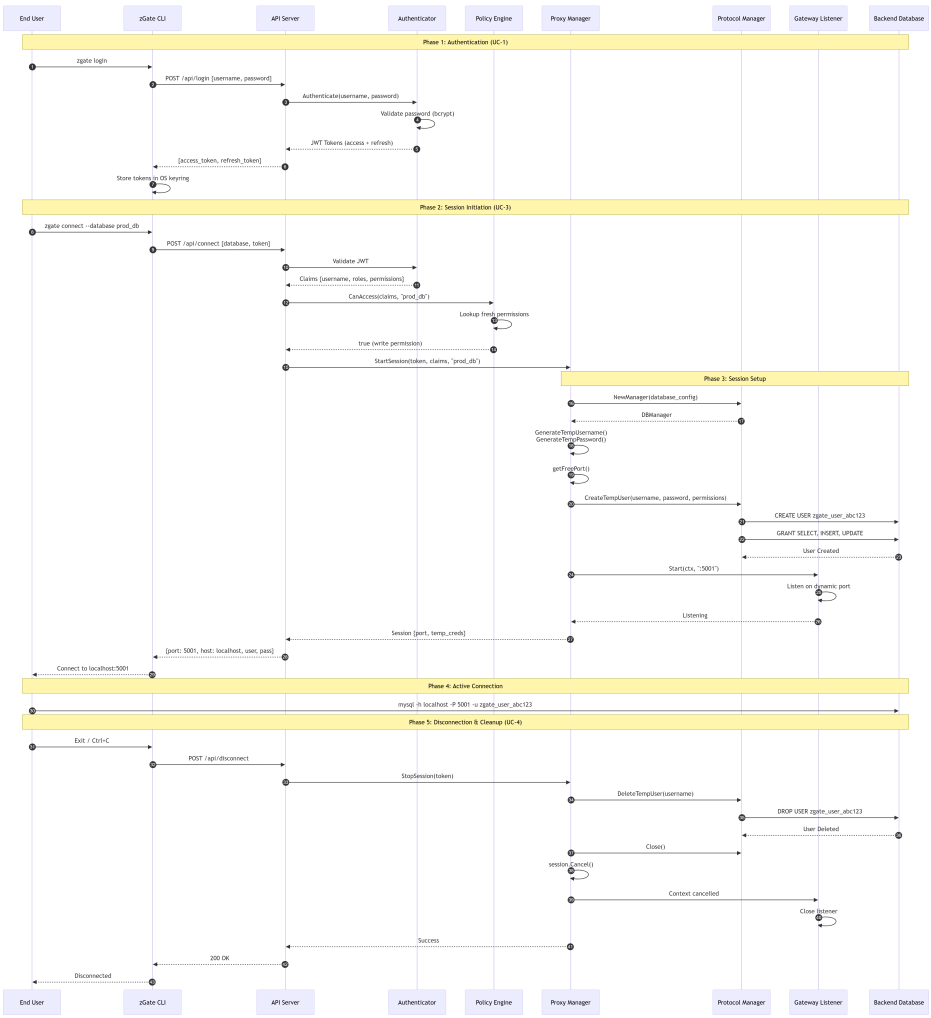


Figure 11.3: Detailed connection lifecycle flow

11.2 Authentication Flow

Authentication is a multi-layered process involving credential validation, token generation, and permission resolution.

11.2.1 Authentication Architecture

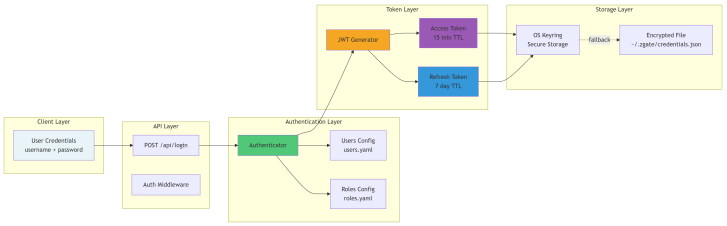


Figure 11.4: Authentication architecture

11.2.2 Token Structure

Access Token Claims

```
{
  key"sub": val"kemo@company.com",
  key"username": val"kemo@company.com",
  key"roles": [val"data_analyst"],
  key"permissions":
     [key"database": val"azure\_mssql\_prod",key"level": val"read",],
```

Refresh Token Claims

```
{
  key"sub": val"kemo@company.com",
  key"username": val"kemo@company.com",
  key"iat": 1702345678,key"exp": 1702950478,
  key"type": val"refresh"
}
```

11.2.3 Token Refresh Flow

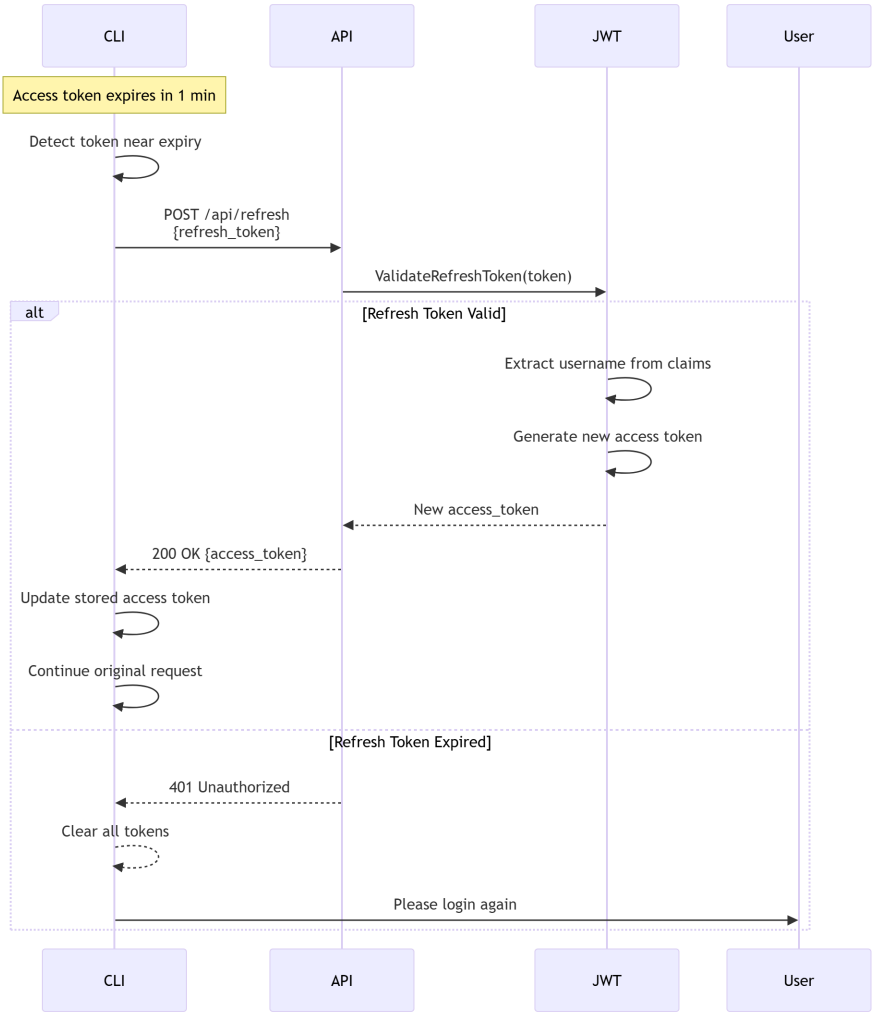


Figure 11.5: Token refresh flow

11.3 Query Filtering Flow

Query filtering ensures that users can only execute operations within their granted permission levels. While the current implementation relies on database-level permissions, the architecture supports query-level filtering.

Permission Levels

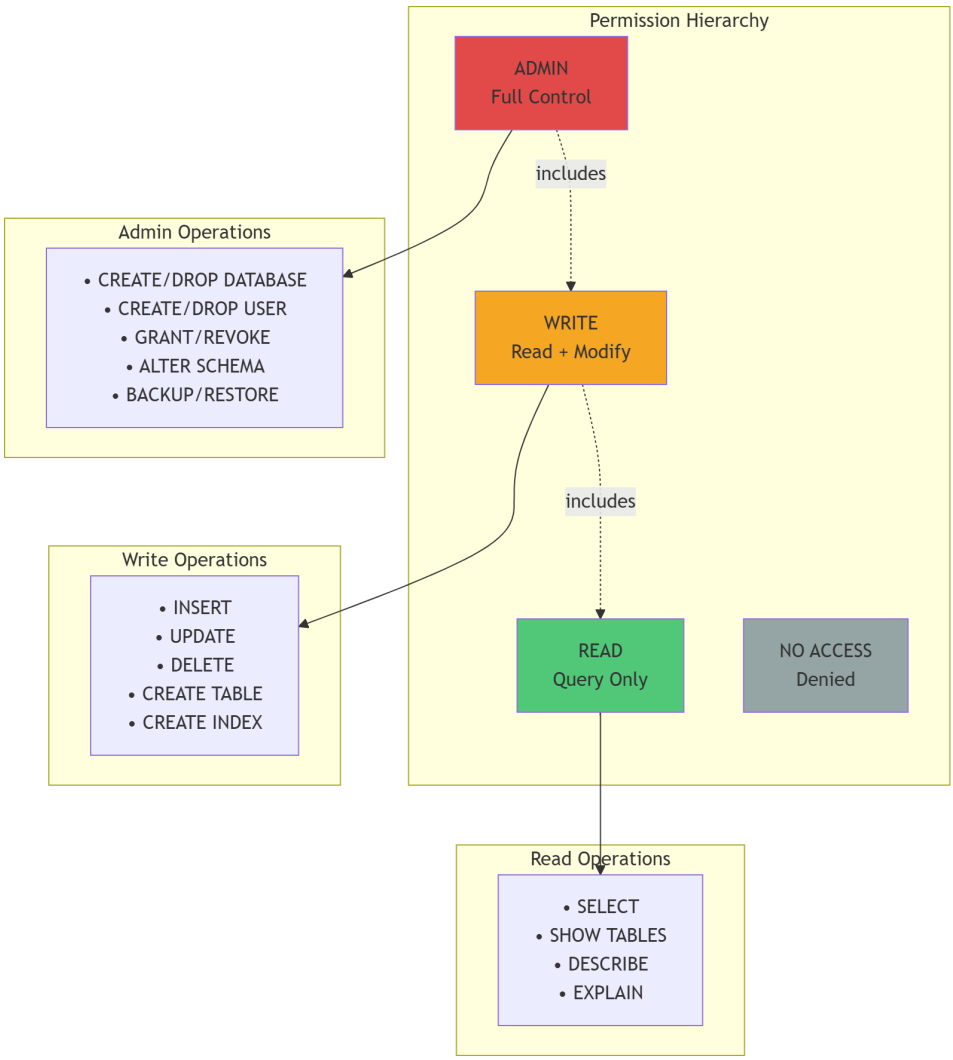


Figure 11.6: Permission levels: read, write, admin

Permission Enforcement Flow

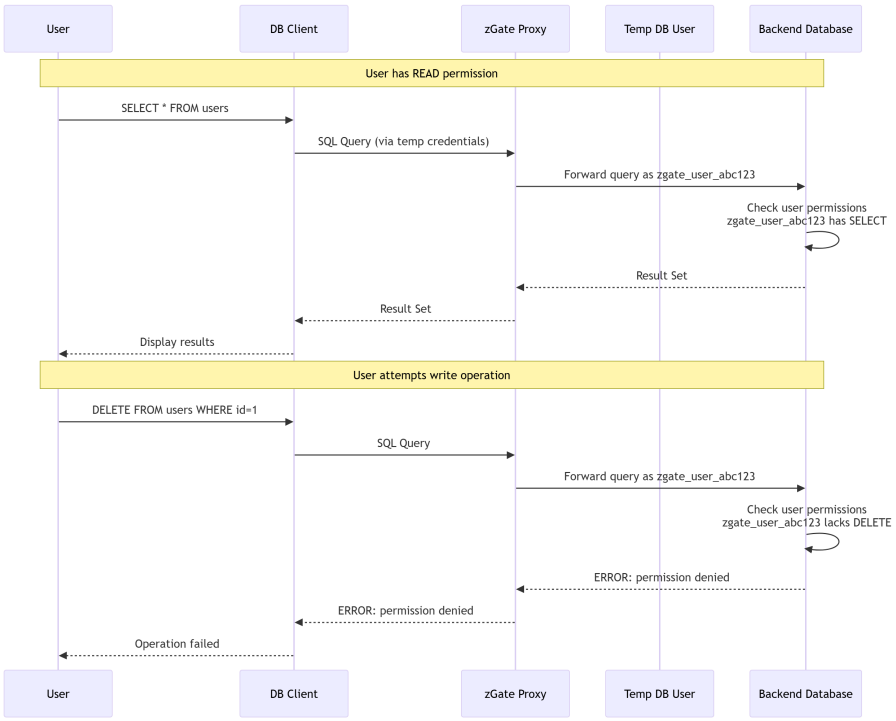


Figure 11.7: Permission enforcement flow

Permission Mapping

The proxy maps zGate permission levels to database-specific grants:

MSSQL Permission Mapping

zGate Level	MSSQL Grants
read	SELECT, VIEW DEFINITION
write	SELECT, INSERT, UPDATE, DELETE
admin	CONTROL, ALTER, CREATE TABLE, DROP TABLE

Table 11.1: MSSQL permission mapping

MySQL Permission Mapping

zGate Level	MySQL Grants
read	SELECT, SHOW VIEW
write	SELECT, INSERT, UPDATE, DELETE
admin	ALL PRIVILEGES

Table 11.2: MySQL permission mapping

11.3.1 Temporary User Creation

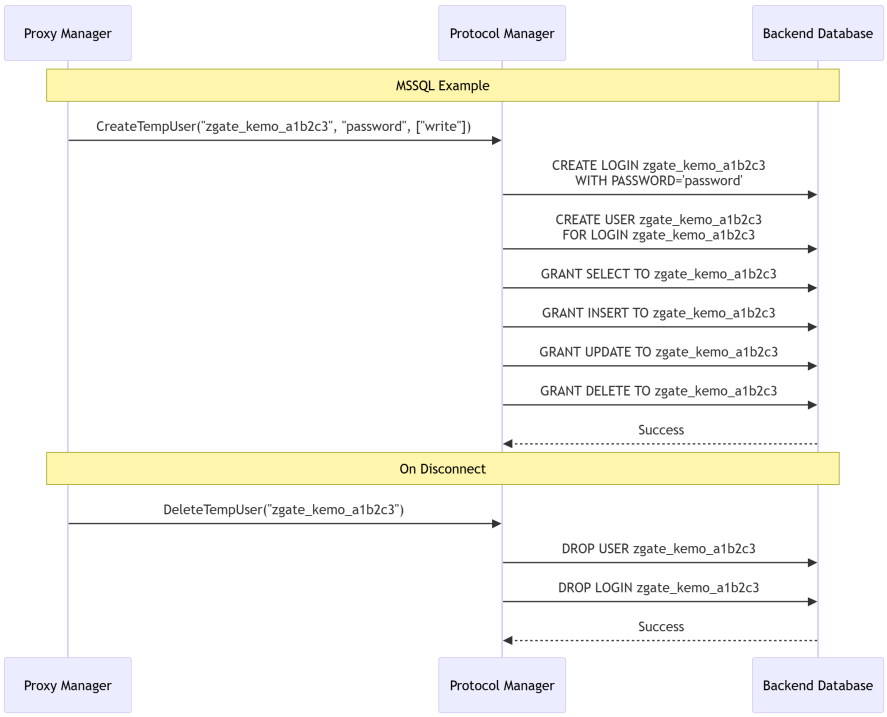


Figure 11.8: Temporary user creation flow

11.4 Policy Enforcement Flow

Policy enforcement is the critical security layer that determines whether a user can access a specific database based on real-time configuration evaluation.

11.4.1 Policy Architecture

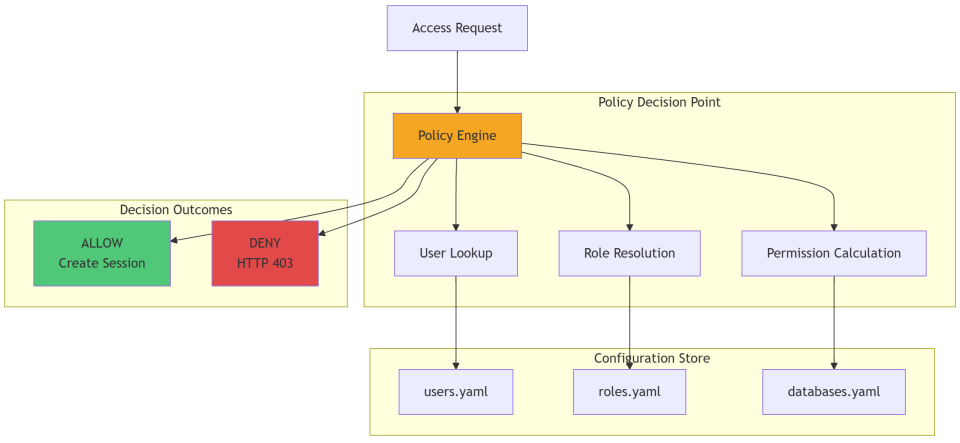


Figure 11.9: Policy enforcement architecture

11.4.2 Real-Time Policy Evaluation

The policy engine always evaluates permissions using the **current** configuration, not the permissions embedded in the JWT token. This ensures immediate effect when permissions are changed.

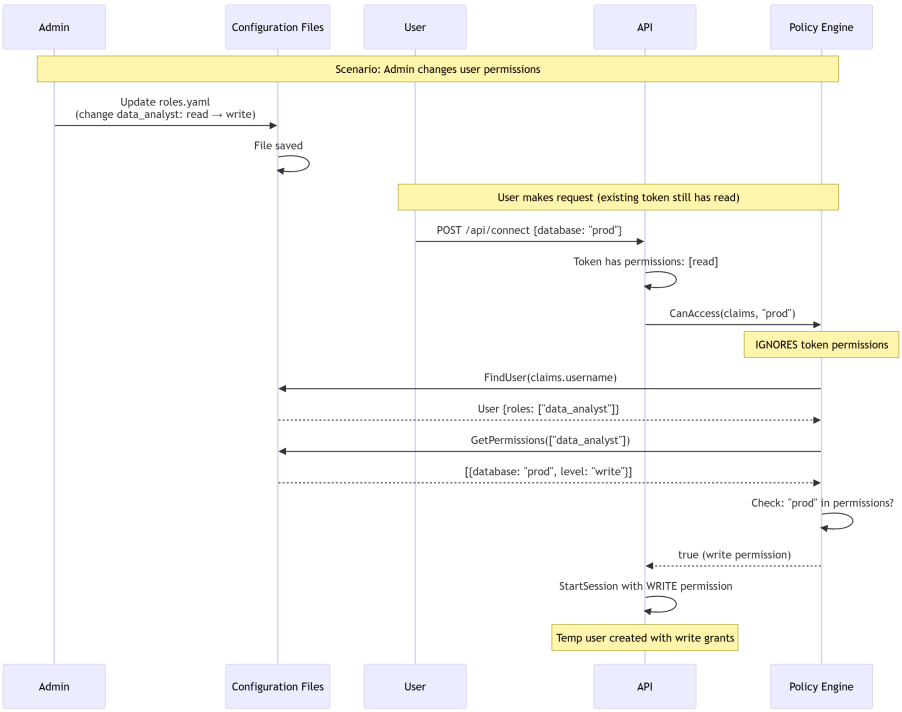


Figure 11.10: Real-time policy evaluation

11.4.3 Policy Enforcement Points

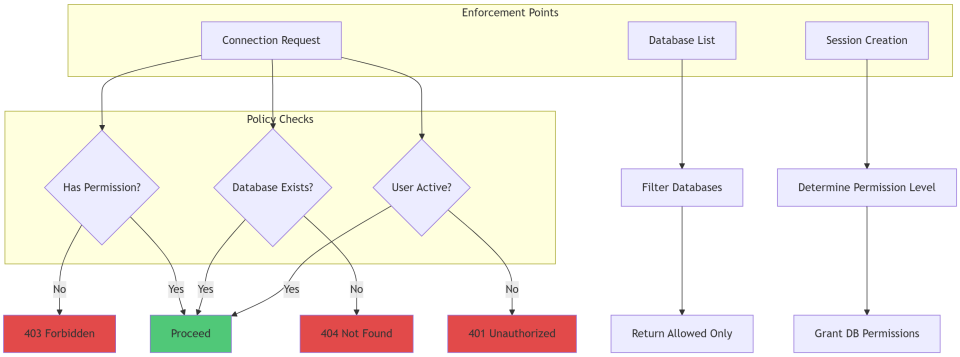


Figure 11.11: Policy enforcement points in the connection lifecycle

11.4.4 End-to-End Flow Narrative

Phase 1: Authentication & Authorization

1. **User** executes `zgate login` via **CLI**

2. **CLI** sends credentials to **LoginAPI**
3. **Authenticator** validates against `users.yaml`
4. Password verified using bcrypt comparison
5. **Authenticator** resolves permissions from `roles.yaml`
6. JWT tokens generated and returned to **CLI**
7. **CLI** stores tokens in OS keyring

Phase 2: Database Discovery

1. **User** executes `zgate list` via **CLI**
2. **CLI** sends request with token to **ListAPI**
3. **AuthMiddleware** validates JWT token
4. **PolicyEngine** looks up fresh permissions from `users.yaml` + `roles.yaml`
5. **PolicyEngine** filters `databases.yaml` based on permissions
6. Accessible databases returned to **CLI**
7. **CLI** displays formatted list to **User**

Phase 3: Connection Establishment

1. **User** executes `zgate connect --database prod_db` via **CLI**
2. **CLI** sends request to **ConnectAPI** with token
3. **AuthMiddleware** validates token, extracts claims
4. **PolicyEngine** performs real-time permission check
5. **ConnectAPI** calls **ProxyManager**.StartSession()
6. **ProxyManager** allocates dynamic port (e.g., 5001)
7. **ProtocolFactory** creates appropriate manager (MSSQL/MySQL)
8. **MSSQLMgr/MySQLMgr** generates temp credentials
9. **Manager** creates temporary user in **Backend Database**
10. **Manager** grants permissions based on user's level

11. **ProxyManager** creates **Session** object
12. **Session** starts **DynamicListener** on allocated port
13. Connection details returned to **CLI**
14. **CLI** displays connection instructions to **User**

Phase 4: Active Database Usage

1. **User** uses **DBClient** (e.g., `mysql -h localhost -P 5001 -u zgate_kemo_abc123`)
2. **DBClient** connects to **DynamicListener** on port 5001
3. **DynamicListener** accepts connection via **ConnectionAcceptor**
4. **ProtocolHandler** establishes backend connection to **MSSQL/MySQL**
5. Database traffic proxied bidirectionally
6. All queries executed as **TempUser** with restricted permissions
7. **Backend Database** enforces grants (read/write/admin)
8. Results returned through proxy to **DBClient**

Phase 5: Session Termination

1. **User** exits **DBClient** or executes `zgate disconnect`
2. **CLI** sends disconnect request to **DisconnectAPI**
3. **ProxyManager**.StopSession() called
4. **Session** context cancelled
5. **DynamicListener** stops accepting connections
6. **Manager** drops temporary user from **Backend Database**
7. **Manager** closes database connection
8. Port released for reuse
9. **Session** removed from **ProxyManager** map
10. Success response returned to **CLI**

12. High-Level Data Flow Diagrams

12.1 Authentication Flow Diagram

12.2 Query Filtering Diagram

12.3 Logging & Auditing Flow Diagram

13. Technology Justification

13.1 Why Go

The selection of Go (Golang) as the primary language for implementing the zGate Gateway proxy was driven by several technical requirements unique to high-performance, security-critical network infrastructure. Unlike interpreted languages, Go provides the performance characteristics and concurrency model essential for building a production-grade database access proxy.

13.1.1 Performance & Execution Model

Go is a compiled language that translates source code directly into machine code, unlike interpreted languages such as Python that execute code line-by-line at runtime. This fundamental architectural difference results in significantly lower latency and higher throughput—critical metrics for a proxy that sits between clients and database servers, where every millisecond of added latency compounds across thousands of queries.

Additionally, Go applications exhibit substantially lower memory overhead compared to equivalent implementations in interpreted languages. This efficiency allows the gateway to handle significantly more concurrent traffic on the same hardware resources, directly impacting the scalability and cost-effectiveness of the system.

13.1.2 Goroutine Concurrency Model

The most significant advantage of Go for proxy server implementation lies in its concurrency model based on goroutines. This feature is crucial for handling the massive number of simultaneous database connections that a production gateway must support.

Traditional threading models impose significant memory overhead, as each thread typically consumes megabytes of stack space. In contrast, goroutines are extremely lightweight, consuming only approximately 2KB of initial stack space. This efficiency enables the gateway to spawn tens of thousands of goroutines to handle concurrent proxy connections without exhausting system resources—a capability that would be impractical with traditional threading approaches.

Furthermore, Go provides *channels* as a first-class language construct for safely communicating between concurrent processes. This built-in mechanism eliminates the complex locking patterns and race conditions that plague concurrent programming in other languages, making the codebase more maintainable and less prone to subtle concurrency bugs.

13.1.3 Production-Grade Standard Library

Go was designed by Google specifically for building networked systems and internet services. The standard library's `net` and `net/http` packages are robust, secure, and production-ready out of the box, eliminating the need for heavy external frameworks.

The `net` library provides fine-grained control over TCP socket behavior, including precise management of timeouts, deadlines, and keep-alive settings. This low-level control is essential for a custom proxy that must intelligently manage traffic flow, implement connection pooling, and enforce session policies at the transport layer.

13.1.4 Security & Cryptography Ecosystem

Security is paramount for a Zero Trust database access gateway, and Go's cryptography ecosystem is exceptionally well-suited to this requirement. The `crypto/tls` package in Go is considered one of the industry's best implementations of SSL/TLS protocols, with built-in support for modern standards including TLS 1.3 by default.

Critically, Go is a memory-safe language despite its performance characteristics comparable to C or C++. The language's garbage collector prevents common security vulnerabilities such as buffer overflows, use-after-free errors, and memory leaks—all of which would be catastrophic in a security gateway positioned between untrusted clients and sensitive database systems.

13.1.5 Deployment Simplicity

Go's compilation model produces a single static binary that bundles all dependencies and libraries. This characteristic eliminates "dependency hell" and dramatically simplifies deployment operations.

To deploy the zGate Gateway to a server or container, operators simply copy a single executable file—no runtime installation, no package manager invocations, and no version conflict resolution required. This simplicity reduces the attack surface, minimizes deployment complexity, and ensures consistent behavior across different target environments.

13.1.6 Developer Experience & Code Maintainability

Go's minimalist design philosophy emphasizes simplicity and readability, featuring a deliberately small syntax with no complex inheritance hierarchies or implicit behaviors. This characteristic is particularly valuable for security-critical software, where code auditability is essential. If code is easy to read, it is correspondingly easier to audit for security flaws and verify correctness.

As a statically typed language with a powerful type system, Go enables the compiler to catch many classes of bugs—including type mismatches, null pointer dereferences, and interface violations—before code execution. This compile-time validation substantially reduces the likelihood of runtime errors in production environments, contributing to overall system reliability.

13.2 Why Node.js / TypeScript / React

The zGate Web Administration Dashboard serves as the centralized control plane for the entire gateway infrastructure. While the Go backend handles the computationally intensive proxy operations, the WebUI provides administrators with real-time visibility, configuration management, and audit capabilities. The selection of Node.js, TypeScript, and React for this interface was driven by the need for a responsive, type-safe, and maintainable frontend that complements the Go backend architecture.

13.2.1 React: Dynamic and Real-Time Dashboard Capabilities

The administrative dashboard for a database access proxy presents unique user interface challenges. Administrators require real-time visibility into active connections, traffic patterns, policy violations, and system health metrics—all of which change continuously during normal operations.

Component-Based Architecture

React's component-based architecture aligns naturally with the modular nature of a proxy gateway dashboard. The interface comprises numerous repeating elements: toggle switches for policy rules, status cards for connection statistics, table rows for audit logs, and configuration forms for database endpoints. React enables these elements to be built as reusable, self-contained components that encapsulate their own logic and styling. This approach yields a clean, organized codebase where modifications to one component do not cascade unpredictably throughout the application.

Virtual DOM for Live Monitoring

A production proxy processes database traffic continuously, generating metrics and events at high frequency. Displaying live traffic graphs, active connection counts, and streaming audit logs requires the UI to update rapidly without degrading user experience. React's Virtual DOM reconciliation algorithm addresses this challenge by computing the minimal set of DOM mutations required to reflect state changes. Rather than re-rendering entire page sections, React surgically updates only the elements that have actually changed, preventing the lag and flickering that would otherwise occur with frequent data updates.

Single Page Application Model

The dashboard implements a Single Page Application (SPA) architecture, providing administrators with a fluid, application-like experience. Navigation between views—such as switching from the Dashboard to Settings to Audit Logs—occurs instantaneously without full page reloads. This responsiveness is critical for operational scenarios where administrators must rapidly investigate security events or modify policies under time pressure.

13.2.2 TypeScript: Type Safety and Contract Enforcement

The selection of TypeScript over plain JavaScript reflects a deliberate architectural decision to extend the type safety guarantees of the Go backend into the frontend layer.

Consistency with Backend Type Discipline

Go was selected for the backend partly due to its static type system, which catches entire categories of bugs at compile time. Using TypeScript brings equivalent discipline to the frontend codebase. Common JavaScript runtime errors—such as accessing properties on undefined values or passing incorrect argument types—are detected during compilation rather than manifesting as failures in production. This consistency in type safety across both layers of the application reduces the overall defect rate and improves system reliability.

API Contract Enforcement Through Interfaces

The WebUI communicates with the Go backend through RESTful API endpoints that exchange JSON payloads. TypeScript's interface system enables precise definition of these data contracts, ensuring that frontend code correctly handles the structures returned by the backend.

For example, if the Go backend defines a database connection response containing a numeric identifier, TypeScript interfaces enforce this contract throughout the frontend codebase. Any attempt to treat this identifier as a string or access non-existent properties

results in a compile-time error rather than a subtle runtime bug. This compile-time validation is particularly valuable as the API evolves, since TypeScript immediately flags any frontend code that becomes incompatible with backend changes.

Enhanced Developer Tooling

TypeScript's static analysis enables sophisticated editor features including intelligent autocompletion, inline documentation, and real-time error highlighting. These capabilities accelerate development velocity by reducing the cognitive load on developers and eliminating the need for frequent context switches to reference documentation or debug type-related issues.

13.2.3 Node.js: Ecosystem Access and Build Infrastructure

While the production backend runs entirely on Go, the frontend development environment leverages Node.js to access the extensive JavaScript ecosystem and modern build tooling.

NPM Registry and Library Ecosystem

The Node Package Manager (NPM) registry provides access to thousands of production-ready libraries that would be impractical to develop in-house. For the zGate dashboard, this ecosystem enables rapid integration of specialized capabilities:

- **Data Visualization:** Libraries such as Recharts and Chart.js provide sophisticated charting capabilities for rendering traffic volume graphs, connection histograms, and latency distributions—visualizations essential for monitoring proxy health and identifying anomalies.
- **State Management:** Complex dashboard state—including authentication status, cached configuration data, and real-time metrics—is managed through established patterns using libraries like Zustand or Redux, providing predictable state transitions and debugging capabilities.
- **UI Component Libraries:** Pre-built component libraries such as Radix UI provide accessible, well-tested interface primitives that accelerate development while ensuring consistency and accessibility compliance.

Modern Build Tooling

Node.js powers the build pipeline through tools like Vite, which provides near-instantaneous hot module replacement during development. When a developer saves a file, changes appear in the browser within milliseconds without losing application state. This rapid feed-

back loop dramatically improves development efficiency compared to traditional build-refresh cycles.

13.2.4 Architectural Separation of Concerns

The decision to implement the WebUI as a separate application from the Go proxy reflects a deliberate architectural pattern that provides operational and reliability benefits.

Decoupled Failure Domains

By separating the presentation layer (React/Node.js) from the core proxy logic (Go), the system establishes independent failure domains. If the WebUI experiences an error—whether due to a browser compatibility issue, a JavaScript exception, or a frontend deployment problem—the Go proxy continues operating uninterrupted. Database connections remain active, policies continue to be enforced, and audit logging persists. Administrators may temporarily lose dashboard visibility, but the security-critical proxy functionality remains unaffected.

Independent Scaling and Resource Allocation

The decoupled architecture enables independent resource allocation for each tier. The Go proxy may require substantial CPU and memory resources to handle high connection volumes, while the WebUI imposes minimal server-side load since rendering occurs in the administrator's browser. This separation prevents dashboard activity from competing with proxy operations for system resources, ensuring that administrative tasks do not degrade proxy performance under load.

Independent Development and Deployment Cycles

Frontend and backend teams can develop, test, and deploy their respective components independently, provided API contracts are maintained. UI improvements, new dashboard features, or visual redesigns can be shipped without redeploying or restarting the proxy, minimizing operational risk and enabling more frequent iterations on the administrative experience.

13.3 Why SQLite for Internal Storage

The evolution of the zGate Gateway's configuration management architecture represents a critical design decision that directly impacts system reliability, security, and operational efficiency. Initially, the system utilized YAML files for storing configuration data, including database connection strings, user permissions, and policy rules. However, this

approach proved insufficient for a production-grade security gateway, leading to the adoption of SQLite as the internal storage mechanism.

13.3.1 Zero-Downtime Updates

The most significant limitation of file-based configuration was the requirement for application restarts to apply changes. In the YAML-based implementation, configuration data was loaded into memory only during application startup. Any modification to proxy rules, database connection strings, or user permissions required editing the file and restarting the entire gateway—an operation that necessarily resulted in dropped connections and service interruption.

SQLite fundamentally resolves this issue by enabling real-time configuration queries. When an administrator updates a setting through the WebUI, the change is committed to the database immediately and atomically. Subsequent requests automatically retrieve the updated configuration without requiring any application restart or service disruption. This capability is essential for maintaining high availability in production environments where configuration changes are routine operational tasks.

13.3.2 Efficiency & Memory Management

The YAML approach required loading the entire configuration dataset into memory at startup, creating a cached representation of all configuration data. This architecture introduced several problems, including memory inefficiency and the risk of state drift—a condition where in-memory data diverges from the on-disk representation if files are modified externally or by concurrent processes.

SQLite's relational model eliminates these issues through structured, on-demand data access. Rather than maintaining complex nested maps and arrays in memory, the gateway queries specific configuration elements exactly when needed. This approach significantly reduces the application's memory footprint, particularly as the configuration dataset grows to encompass hundreds of database connections, thousands of user accounts, and complex policy rules.

Furthermore, SQLite's query optimizer ensures that data retrieval operations are efficient even as the dataset scales, something that would require substantial custom indexing logic if implemented with in-memory data structures.

13.3.3 API Integration & WebUI Compatibility

Modern administrative interfaces require comprehensive Create, Read, Update, and Delete (CRUD) operations on configuration data. Implementing these operations safely with

YAML files—particularly handling concurrent modifications, maintaining data integrity, and providing transactional semantics—is complex and error-prone.

SQLite provides a standardized SQL interface that dramatically simplifies API development. The Go backend can leverage SQL's powerful query capabilities to implement sophisticated operations with minimal code. For example, filtering connection strings by environment, paginating audit logs, or searching for users by role becomes trivial with SQL queries:

```
SELECT * FROM connections
WHERE environment='production'
ORDER BY name LIMIT 10
```

This SQL-based approach integrates seamlessly with the React/Node.js WebUI, which can issue standard REST API calls that translate directly to SQL queries. The result is a clean, maintainable codebase with clear separation between the presentation layer (React), business logic (Node.js API), and data persistence (SQLite).

13.3.4 Enhanced Security Through Data-at-Rest Encryption

Security considerations provided the final compelling argument for SQLite adoption. YAML files are inherently plain text, meaning that any attacker who gains read access to the server's filesystem can immediately view all sensitive configuration data, including database credentials, encryption keys, and access tokens.

To address this vulnerability, the gateway implements a data-at-rest encryption strategy using AES (Advanced Encryption Standard). Before persisting sensitive data to the SQLite database, the Go application encrypts it using AES-256 in GCM mode. Database connection strings, API keys, and other sensitive fields are stored only in their encrypted form.

The practical impact of this approach is substantial: even if an attacker obtains a copy of the SQLite database file through a filesystem breach or backup compromise, the sensitive columns contain only cryptographically secure ciphertext. Only the running Go application, which holds the master encryption key in memory (and never persists it to disk), can decrypt and utilize the actual configuration data. This defense-in-depth strategy aligns with Zero Trust principles by assuming that filesystem access controls may be breached and providing an additional layer of protection.

13.4 Why mTLS (and why TCP is temporary)

13.5 Design Decision Summary

14. Prototype – Semester 1

14.1 Implemented Features

14.2 Screenshots (CLI & Dashboard)

14.3 What Works vs What Doesn't

14.4 Technical Decisions Made

14.5 Implementation Challenges

15. Development Methodology

15.1 Agile Scrum Framework

To manage the complexity of the project and ensure continuous development, the team adopted the Agile Scrum methodology. We structured the development lifecycle into one-week sprints, enabling rapid iteration and frequent feedback cycles. This short sprint duration allowed us to demonstrate tangible progress weekly and incorporate supervisor feedback more frequently, ensuring alignment with project objectives throughout the development process.

15.2 Meeting Structure

Our workflow is organized around three distinct meeting types: the Weekly Kick-off, Daily Stand-ups, and the Sprint Review with stakeholders. This structured rhythm of recurring meetings ensures that team milestones and progress are consistently monitored and synchronized. Collectively, these meetings serve to define, review, and align all weekly tasks in accordance with agile best practices.

15.2.1 Weekly Kick-off Meeting

This meeting is held at the start of every sprint to align the team for the upcoming week. It encompasses three key components:

- **Retrospective:** We briefly analyze the previous sprint, discussing what went well and identifying bottlenecks (e.g., merge conflicts or unclear requirements). This reflection helps the team avoid repeating past mistakes and continuously improve our process.
- **Backlog Refinement:** We review upcoming tasks to ensure they are clearly defined and that all technical requirements are understood before assignment. This step reduces ambiguity and sets clear expectations.
- **Sprint Planning:** We select specific tasks from the refined backlog to be completed in the current week. Tasks are assigned to team members based on priority,

complexity, and estimated effort.

15.2.2 Daily Stand-up Meeting

This brief synchronization meeting is held daily to maintain continuous team alignment and identify blockers early.

- **Duration:** Limited to approximately 15 minutes total, with each member speaking for no more than 2 minutes. This time constraint ensures the meeting remains focused and efficient.
- **Format:** Each team member addresses two specific points: what they accomplished yesterday and what they plan to work on today. Any blockers or dependencies are also raised.
- **Objective:** This practice ensures that no team member works in isolation or is blocked by dependencies without the rest of the team being aware. It promotes transparency and enables rapid problem-solving.

15.2.3 Sprint Review (Weekly Supervisor Meeting)

At the conclusion of each sprint, a formal review is conducted with our project supervisor and mentors to validate progress and gather feedback.

- **Demonstration:** The team presents the functional features completed during the sprint, showcasing working software rather than just documentation or plans.
- **Validation:** Our supervisors provide immediate feedback on the implementation. This feedback is either approved for integration or converted into new change requests to be prioritized in the next sprint's backlog.

15.3 Collaboration Tools

To ensure accessibility and efficient collaboration across all phases of development, we employ an integrated tool stack that supports both synchronous and asynchronous communication:

Central Knowledge Base (Notion): Serves as the single source of truth for the project wiki. It is used for sprint planning, task and goal tracking, documenting new code features, and archiving meeting notes and recordings. All team members have real-time access to project documentation.

Version Control & Technical Documentation (GitHub): The primary repository for source code, version control, and code reviews. GitHub also hosts technical documentation, including the Proxy Installation Guide and API references.

Synchronous Communication: Discord, Microsoft Teams, and Google Meet are used for scheduled meetings and real-time voice/video collaboration. These platforms facilitate immediate discussion and decision-making.

Asynchronous Communication: WhatsApp is used for quick, urgent team notifications and simple coordination when immediate responses are needed outside of scheduled meetings.

Auxiliary Tools: We leverage AI-powered tools for generating meeting summaries and conclusions, creating immediate and searchable records of discussions. Excalidraw is used for creating collaborative diagrams, flowcharts, and architectural drawings during design discussions.

15.4 Documentation & Observability

We prioritize high observability by documenting all progress, regardless of scale. This comprehensive documentation approach ensures transparency and facilitates knowledge transfer within the team.

While day-to-day execution is tracked in Notion, high-level progress reporting is documented in a formal **Supervisor Meeting Log** maintained as a Word document. This log serves as an official record and tracks:

- **Attendance & Date:** A record of meeting participants and timestamps.
- **Retrospective:** Feedback on completed tasks, including what was delivered and any deviations from the plan.
- **Forward Planning:** Clearly defined goals and expected outcomes for the subsequent meeting, establishing accountability and measurable targets.

16. Task Tracking

16.1 Team Task Tracking (Actual Examples)

16.2 Supervisor Tracking Logs

16.3 Blockers, Risks & Resolution Notes

17. Milestones

The development of zGate is structured around six critical milestones that progressively build the system from foundational research to a production-ready solution. The first three milestones, which have been successfully completed during Term 1, established the architectural core and delivered a functional Proof of Concept (POC). The remaining three milestones, scheduled for Term 2, focus on layering security integrations, governance features, and comprehensive validation.

17.1 Milestone Roadmap

17.1.1 Milestone 1: Problem Definition & Market Validation

Status: Completed **Duration:** September 5 – October 15

This foundational phase established the theoretical framework for zGate by validating the necessity for a granular, identity-based database proxy within the Zero Trust security paradigm.

Objectives

- Define the scope of the "Zero Trust Database Access" problem domain.
- Benchmark existing solutions and identify market gaps.
- Establish the initial system architecture.

Key Activities

- **Gap Analysis:** Investigated the limitations of traditional VPNs and existing database proxies such as Hoop.dev and PgBouncer. The analysis revealed critical weaknesses in access granularity and user experience that zGate aims to address.
- **Architecture Design:** Defined the "Man-in-the-Middle" architecture required to intercept binary wire protocols for MySQL, PostgreSQL, and MongoDB.
- **Requirements Specification:** Finalized the requirement for protocol-agnostic handling and identity provider integration.

- **Feasibility Study:** Researched the technical viability of intercepting binary wire protocols without introducing significant latency overhead.

Deliverables

- Validated Software Requirements Specification (SRS).
- Initial architectural diagrams and system design documents.

17.1.2 Milestone 2: Technical Skill Acquisition & Research

Status: Completed **Duration:** October 16 – November 15

Following the initial research phase, this milestone was dedicated to equipping the development team with the specialized technical skills and theoretical knowledge required to build a high-performance database proxy.

Objectives

- Acquire proficiency in the Go programming language.
- Understand low-level database communication protocols.
- Explore modern observability concepts and standards.

Key Focus Areas

- **Database Protocol Engineering:** Conducted in-depth study of the binary wire protocols for MySQL, PostgreSQL, and MongoDB. This involved reverse-engineering how databases handle handshakes, authentication sequences, and query transmission at the packet level.
- **Go Language Mastery:** Focused on mastering the Go programming language, with particular emphasis on its concurrency primitives—Goroutines and Channels—which are essential for managing multiple simultaneous network connections efficiently.
- **Observability Research:** Investigated the principles of system observability, including OpenTelemetry standards and eBPF concepts for potential kernel-level tracing in future iterations.

Deliverables

- Team proficiency in the project's technology stack.
- Internal documentation on protocol structures.

- Initial Go practice implementations and code samples.

17.1.3 Milestone 3: Functional Proof of Concept & Core Features Delivery

Status: Completed **Duration:** November 16 – December 15

Originally scoped to focus solely on connection handling, this milestone was strategically expanded to deliver a complete Proof of Concept. The POC encompasses the initial web interface, basic security mechanisms, and internal token management capabilities.

Objectives

- Validate the full technology stack from the web frontend to database connectivity.
- Demonstrate that the proxy can handle traffic, mask sensitive data, and manage internal sessions.

Key Activities

- **Proxy Core Development:**
  - Implemented TCP listeners and connection pooling mechanisms for high-concurrency traffic.
  - Developed protocol parsers for MySQL, PostgreSQL, and MongoDB packets.
  - Built the core data forwarding logic with support for bidirectional communication.
  - Handled CLI errors and established graceful shutdown procedures.
- **Security Logic Implementation:**
  - Implemented Data Masking functionality using regex-based redaction for PII.
  - Built Token Refresh and Revocation mechanisms for session management.
  - Developed OAuth2-style authentication flows (login, refresh, logout).
  - Implemented AES encryption for secure storage of credentials.
- **User Interface Development:**
  - Developed the Initial Admin Dashboard Web UI to visualize system status.
  - Built components for managing database accounts and access control roles.
  - Integrated real-time session and query monitoring capabilities.

Deliverables

- A working end-to-end Proof of Concept demonstrating core functionality.
- Admin dashboard with system visibility and management capabilities.
- Functional data masking and token management subsystems.

17.1.4 Milestone 4: External IAM & Identity Integration

Status: Planned **Duration:** January – February (Term 2)

This milestone bridges the internal token system established in Milestone 3 with external Identity Providers (IdP), enabling enterprise-grade Single Sign-On (SSO) capabilities.

Objectives

- Integrate external Identity Providers for SSO authentication.
- Replace internal authentication with federated identity management.
- Map external identity claims to internal Role-Based Access Control (RBAC) policies.

Key Activities

- **Third-Party Integration:** Connect the Admin Dashboard and Proxy to external providers (Google, Okta, Azure AD) to enable SSO workflows.
- **Authentication Flow:** Implement the OAuth2/OIDC handshake where the interface redirects to the IdP and receives identity tokens.
- **RBAC Mapping:** Map identity claims from third-party providers to user roles within the zGate policy engine.

Expected Deliverables

- SSO login capability via "Sign in with Google/Okta" or other configured IdPs.
- Removal of hardcoded or internal-only authentication from the POC.
- Comprehensive documentation for IdP configuration.

17.1.5 Milestone 5: Observability & Production Readiness

Status: Planned **Duration:** March – April (Term 2)

This milestone transforms the initial dashboard into a comprehensive monitoring platform suitable for production deployment.

Objectives

- Implement comprehensive metrics visualization and monitoring.
- Establish audit logging for compliance requirements.
- Ensure system robustness under various failure conditions.

Key Activities

- **Metric Visualization:** Populate the dashboard with real-time graphs utilizing connection pooling metrics and system health indicators.
- **Audit Logging:** Structure session data and access events into a searchable audit trail for compliance and forensic purposes.
- **Error Handling Polish:** Implement graceful handling of all possible error conditions across the system, ensuring stability under load.

Expected Deliverables

- Production-ready Admin Dashboard with live metrics visualization.
- Searchable audit logs with filtering and export capabilities.
- Comprehensive error handling and graceful degradation.

17.1.6 Milestone 6: Final Validation, Benchmarking & Delivery

Status: Planned **Duration:** May – June (Term 2)

The final milestone serves as a comprehensive quality assurance phase to ensure the system meets production standards and is fully documented.

Objectives

- Stress-test the system under realistic load conditions.
- Validate security robustness through penetration testing.
- Finalize all documentation and prepare for project handoff.

Key Activities

- **Performance Benchmarking:** Execute load testing to measure latency overhead introduced by the proxy. Optimize Go code based on profiling results.
- **Security Auditing:** Conduct internal penetration testing, including SQL injection and bypass attack attempts, to validate the robustness of the Policy Engine.
- **Final Documentation:** Complete the Administrator's Guide, Developer Handbook, and final academic report.

Expected Deliverables

- Final Source Code Release (v1.0).
- Comprehensive Final Project Report.
- Robust Demo Environment ready for final presentation.
- Complete documentation suite (installation guides, API documentation, user manuals).

17.2 Timeline Overview

Table 17.1 presents a consolidated view of all milestones with their respective timelines and completion status.

Table 17.1: Milestone Timeline Summary

MS	Milestone Name	Timeline	Status
1	Problem Definition & Market Validation	Sept 5 – Oct 15	✓ Completed
2	Technical Skill Acquisition & Research	Oct 16 – Nov 15	✓ Completed
3	Functional POC & Core Features	Nov 16 – Dec 15	✓ Completed
4	External IAM & Identity Integration	Jan – Feb	Planned
5	Observability & Production Readiness	Mar – Apr	Planned
6	Final Validation & Delivery	May – June	Planned

17.3 Term 1 Timeline Chart

Figure 17.1 illustrates the project timeline for Term 1, depicting the progression from the Planning & Analysis phase through the Design phase and into the initial Development phase.

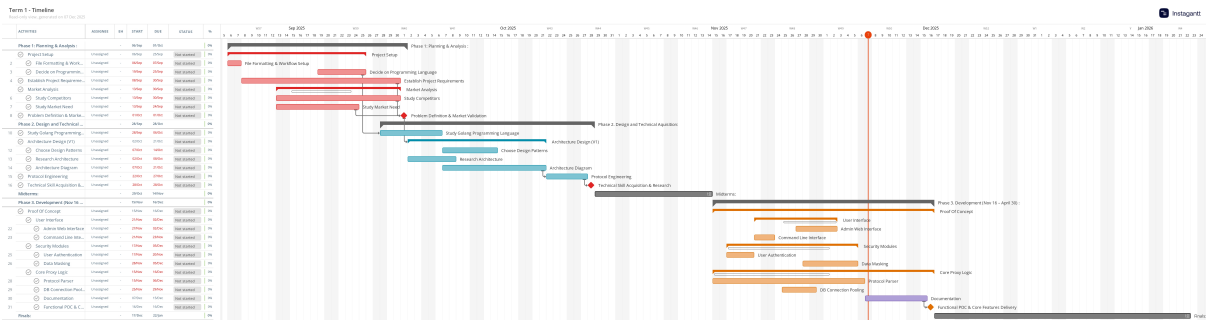


Figure 17.1: Term 1 Project Timeline (September – December)

18. Threat Model & Security Considerations

18.1 Threat Model

18.2 Risks & Attack Vectors

18.3 Mitigation Techniques

18.4 Why Zero Trust is Needed

19. Roadmap for Term 2

Term 2 represents the critical transition from a functional Proof of Concept to a production-ready database security solution. Building upon the foundational work completed in Term 1, the development team will focus on three primary areas: integrating enterprise-grade identity management, implementing comprehensive observability features, and conducting rigorous validation to ensure system reliability and security.

19.1 Remaining Features

The following features are scheduled for implementation during Term 2, organized by milestone and priority.

19.1.1 External Identity Provider Integration (Milestone 4)

The current POC utilizes an internal token-based authentication system. Term 2 will extend this to support federated identity management through external providers.

- **OAuth2/OIDC Implementation:** Integrate industry-standard authentication protocols to enable Single Sign-On (SSO) capabilities.
- **Identity Provider Support:** Configure connectors for major identity providers including:
  - Google Workspace
  - Microsoft Azure Active Directory
  - Okta
  - Generic SAML 2.0 providers
- **Claims-to-Role Mapping:** Develop a flexible mapping engine that translates IdP claims (groups, roles, attributes) to zGate's internal RBAC policies.
- **Just-in-Time Provisioning:** Implement automatic user provisioning based on IdP attributes upon first login.

19.1.2 Enhanced Observability Features (Milestone 5)

Transform the initial dashboard into a comprehensive monitoring and compliance platform.

- **Real-Time Metrics Dashboard:**
  - Connection pool utilization graphs
  - Query throughput and latency percentiles
  - Active session monitoring
  - Database health indicators
- **Audit Logging System:**
  - Structured logging of all database access events
  - Query content capture with optional redaction
  - User activity timeline visualization
  - Export capabilities for compliance reporting (CSV, JSON, SIEM integration)
- **Alerting Framework:**
  - Configurable threshold-based alerts
  - Anomaly detection for unusual access patterns
  - Integration with notification channels (email, Slack, webhooks)

19.1.3 Advanced Policy Engine Enhancements

Extend the current policy engine with more granular control mechanisms.

- **Time-Based Access Controls:** Implement policies that restrict database access to specific time windows.
- **Query Complexity Limits:** Add configurable limits on query complexity to prevent resource exhaustion.
- **Row-Level Security Policies:** Develop mechanisms to enforce row-level filtering based on user identity.
- **Dynamic Masking Rules:** Extend data masking to support context-aware redaction based on user roles.

19.2 Architecture Improvements

Based on lessons learned during POC development, the following architectural enhancements are planned.

19.2.1 Scalability Enhancements

- **Horizontal Scaling Support:** Implement stateless proxy instances that can be load-balanced for high availability.
- **Connection Multiplexing:** Optimize connection pooling to support higher concurrent user loads with fewer backend connections.
- **Caching Layer:** Introduce caching for policy decisions and session metadata to reduce latency.

19.2.2 Security Hardening

- **TLS Certificate Management:** Implement automated certificate rotation and enhanced certificate validation.
- **Secret Management Integration:** Support for external secret stores (HashiCorp Vault, AWS Secrets Manager).
- **Audit Trail Integrity:** Implement cryptographic signing of audit logs to ensure tamper-evidence.

19.2.3 Deployment Improvements

- **Container Orchestration:** Official Docker images and Kubernetes Helm charts for production deployment.
- **Configuration Management:** Migration to environment-based configuration with validation on startup.
- **Health Check Endpoints:** Enhanced liveness and readiness probes for orchestrator integration.

19.3 Performance Goals

Table 19.1 outlines the target performance metrics for the production release.

Table 19.1: Term 2 Performance Targets

Metric	Target	Measurement Method
Latency Overhead (p50)	< 2ms	Benchmarking with synthetic load
Latency Overhead (p99)	< 10ms	Benchmarking under stress
Concurrent Connections	≥ 1000	Load testing with connection ramp
Connection Setup Time	< 50ms	End-to-end connection timing
Memory Footprint	< 500MB (idle)	Resource monitoring under load
CPU Utilization	$< 30\%$ (normal load)	Profiling during benchmark

19.3.1 Optimization Strategies

To achieve these performance targets, the following optimization strategies will be employed:

1. **Profiling-Driven Optimization:** Use Go's built-in profiler (pprof) to identify and eliminate bottlenecks.
2. **Memory Pool Management:** Implement buffer pooling to reduce garbage collection overhead.
3. **Protocol Parser Optimization:** Optimize hot paths in protocol parsing using zero-copy techniques where applicable.
4. **Goroutine Pool Management:** Implement worker pools to prevent goroutine explosion under high load.

19.4 Testing & Validation Plan

A comprehensive testing strategy will ensure system reliability and security before final delivery.

19.4.1 Automated Testing Framework

- **Unit Tests:** Target minimum 80% code coverage for core proxy logic and security modules.
- **Integration Tests:** End-to-end tests covering authentication flows, query processing, and data masking across all supported databases.
- **Regression Test Suite:** Automated test suite executed on every code commit via CI/CD pipeline.

19.4.2 Performance Validation

- **Load Testing:** Use tools such as `pgbench` (PostgreSQL), `mysqlslap` (MySQL), and custom scripts to simulate production workloads.
- **Stress Testing:** Evaluate system behavior under extreme conditions (connection floods, memory pressure).
- **Latency Profiling:** Measure and document latency distribution across different query types and database backends.

19.4.3 Security Validation

- **Penetration Testing:** Internal security audit focusing on:
  - SQL injection bypass attempts
  - Authentication bypass vectors
  - Token forging and replay attacks
  - Policy engine circumvention
- **Threat Modeling:** Document potential attack vectors and corresponding mitigations.
- **Dependency Audit:** Scan all dependencies for known vulnerabilities using automated tools.

19.4.4 User Acceptance Testing

- **Supervisor Demonstrations:** Weekly demonstrations of new features to project supervisors.
- **Documentation Review:** Validation that all user-facing documentation accurately reflects system behavior.
- **Demo Environment:** Prepare a stable demonstration environment for final presentation.

19.4.5 Validation Timeline

Table 19.2 presents the planned testing phases and their timelines.

Table 19.2: Testing & Validation Timeline

Testing Phase	**Timeline**	**Focus Area**
Unit & Integration Testing	Ongoing (all sprints)	Continuous quality assurance
Performance Benchmarking	May 1–15	Latency and throughput validation
Security Audit	May 15–31	Penetration testing and hardening
User Acceptance Testing	June 1–10	Feature validation and feedback
Final Documentation Review	June 10–15	Documentation completeness
Demo Environment Preparation	June 15–20	Final presentation preparation

20. Team Contribution

20.1 Overview of Contribution Approach

20.2 Individual Contributions

21. Expected Outcomes

22. Conclusion

22.1 Restated Purpose

22.2 Summary of Achievements

22.3 Importance & Contribution

22.4 Transition to Next Semester

References

Bibliography

A. Glossary

B. Dashboard Mockups