

# Secret Management System: Design Document

## Overview

A distributed secret management system inspired by HashiCorp Vault that provides secure storage, dynamic secret generation, and fine-grained access control for applications. The key architectural challenge is building a zero-trust security model that encrypts secrets at rest, enforces path-based policies, and maintains high availability while requiring human operators to unseal the system.

This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

## Context and Problem Statement

**Milestone(s):** This section provides the foundational context that applies to all milestones, establishing why we need a centralized secret management system and the security challenges it addresses.

## The Bank Vault Analogy

Think of a modern bank's security system as the perfect mental model for understanding secret management. A bank doesn't just throw cash into a filing cabinet and call it secure. Instead, it employs multiple layers of protection: a massive steel vault with time locks, individual safety deposit boxes within that vault, armed guards who verify your identity before granting access, detailed logs of who accessed what and when, and multiple keys or combinations required to open the main vault (no single person can access everything alone).

Your application's secrets—database passwords, API keys, encryption keys, certificates—are just as valuable as the cash in that bank vault. A compromised database password can expose millions of customer records. A leaked API key can rack up thousands of dollars in cloud costs or grant access to sensitive services. An exposed encryption key can render all your "secure" data readable by attackers.

Just as banks evolved from simple lockboxes to sophisticated multi-layered security systems, application secret management has evolved from simple configuration files to centralized secret management systems. The bank vault analogy maps perfectly to our secret management system:

- **The Steel Vault** → Our encryption engine with envelope encryption
- **Individual Safety Deposit Boxes** → Path-based secret storage with fine-grained access control
- **Bank Guards and ID Verification** → Authentication and authorization engines
- **Access Logs and Cameras** → Comprehensive audit logging
- **Multiple Keys Required** → Shamir's secret sharing for unsealing the system

- **Branch Locations** → High availability with multiple nodes

The critical insight here is that security isn't just about encryption—it's about creating a **zero-trust system** where every request must be authenticated, authorized, logged, and encrypted, just like every bank transaction.

## The Secret Sprawl Problem

In most organizations, secrets proliferate like weeds across the infrastructure landscape, creating what security professionals call "secret sprawl." This scattered approach creates numerous attack vectors and operational nightmares.

Consider a typical web application stack: the frontend needs API keys for third-party services, the backend needs database credentials and encryption keys, the deployment pipeline needs cloud provider credentials, monitoring systems need service account tokens, and backup scripts need storage access keys. Without centralized management, these secrets end up everywhere:

**Configuration Files** become secret repositories by accident. Developers embed database URLs with passwords directly into `config.yaml` files, then check them into version control. Even after rotating the password, it remains in git history forever. The `application.properties` file becomes a treasure trove of sensitive data that gets copied to every deployment environment.

**Environment Variables** seem safer initially, but they leak in process listings, crash dumps, and container orchestration logs. When a developer runs `env | grep DATABASE_PASSWORD` on a production server for debugging, that password appears in shell history. Container platforms often log environment variables during deployment, creating permanent records of sensitive data.

**Hardcoded Secrets** represent the worst-case scenario—passwords embedded directly in source code. These spread through copy-paste programming, appear in multiple repositories, and become nearly impossible to rotate without coordinating changes across dozens of codebases simultaneously.

The fundamental problems with secret sprawl include:

Problem	Impact	Concrete Example
No Centralized Rotation	A single compromised secret requires manual updates across dozens of systems	Database password leak requires updating 15 microservices, 3 batch jobs, and 5 monitoring scripts individually
Unclear Access Patterns	No visibility into which applications access which secrets	Security team can't determine blast radius when API key is found in public GitHub repo
Long-Lived Secrets	Static passwords remain valid for months or years	Production database password hasn't changed in 2 years; compromise goes undetected for months
Inconsistent Security	Each team implements their own secret handling with varying security levels	Team A encrypts secrets with AES-256; Team B stores them in plain text
Audit Gaps	No comprehensive logging of secret access	Compliance audit fails because company can't prove who accessed customer encryption keys
Emergency Response Delays	Incident response requires hunting secrets across multiple systems	After data breach, team spends 6 hours finding all systems that need new credentials

**The Operational Nightmare** manifests during incidents. Imagine your cloud provider notifies you that your S3 access keys were found in a public code repository. In a sprawled environment, you must:

1. Identify every application, script, and service using those keys
2. Generate new credentials in the cloud provider console
3. Update configuration files across multiple repositories
4. Deploy updated configurations to staging and production environments
5. Verify that all systems are working with new credentials
6. Monitor logs to ensure no systems are still using old credentials

This process can take hours or days, during which your systems remain vulnerable. With centralized secret management, this same scenario becomes: update the secret in one location, and all authorized applications automatically receive the new credentials within minutes.

## Current Approaches Comparison

Organizations today employ various strategies for secret management, each with distinct trade-offs that influence security posture, operational complexity, and developer experience.

## Decision: Secret Management Strategy Selection

- **Context:** Applications need secure access to databases, APIs, and other services requiring authentication, but must balance security, operational overhead, and development velocity
- **Options Considered:** Environment variables, configuration files, cloud secret services, dedicated secret management systems
- **Decision:** Implement a dedicated secret management system with centralized storage, encryption, and access control
- **Rationale:** Provides strongest security guarantees while enabling automated secret rotation and comprehensive audit trails
- **Consequences:** Requires additional infrastructure but eliminates secret sprawl and enables sophisticated security policies

Approach	Security Level	Operational Overhead	Developer Experience	Use Cases
Environment Variables	Low-Medium	Low	Excellent	Development environments, simple deployments
Configuration Files	Low	Medium	Good	Traditional applications, legacy systems
Cloud Secret Services	High	Medium	Good	Cloud-native applications, single-provider environments
Dedicated Secret Management	Very High	High	Medium	Enterprise environments, multi-cloud, compliance requirements

**Environment Variables** represent the simplest approach, leveraging the operating system's built-in capability to pass configuration to processes. Developers appreciate the simplicity—

`os.Getenv("DATABASE_PASSWORD")` works identically across languages and platforms. However, environment variables appear in process lists (`ps aux` shows them to any user), get logged by container orchestration systems, and persist in shell history. They work well for development environments where security threats are minimal, but create significant risks in production.

The environmental variable approach breaks down with secret rotation. Changing a database password requires updating environment variables across every deployment, then restarting all affected services. During the transition window, some services use old credentials while others use new ones, creating authentication failures and service disruptions.

**Configuration Files** offer more structure and can support encryption at rest. A `secrets.yaml` file can contain all application secrets in one location, potentially encrypted with a master key. This approach enables version control of secret configurations (with encrypted values) and supports complex secret hierarchies with nested structures.

However, configuration files introduce key distribution problems—how do applications securely obtain the master key to decrypt the configuration file? The key often ends up in environment variables or hardcoded, recreating the original problem. Additionally, updating configuration files requires application restarts, preventing dynamic secret rotation.

**Cloud Secret Services** like AWS Secrets Manager, Azure Key Vault, or Google Secret Manager provide professionally managed secret storage with strong encryption, automated rotation capabilities, and comprehensive audit logging. These services integrate seamlessly with cloud-native applications and handle the operational complexity of secret management.

The primary limitations of cloud secret services include vendor lock-in (applications become tightly coupled to specific cloud providers), limited customization options, and challenges in multi-cloud environments. Organizations using multiple cloud providers must integrate with different secret APIs, each with unique authentication models and feature sets.

**Dedicated Secret Management Systems** like HashiCorp Vault provide the highest level of security and flexibility. These systems implement sophisticated encryption schemes, support multiple authentication methods, enable dynamic secret generation, and offer granular access control policies. They operate independently of cloud providers, supporting multi-cloud and hybrid environments.

The trade-off involves operational complexity—dedicated secret management systems require specialized knowledge to deploy, configure, and maintain. They introduce additional infrastructure dependencies and require careful attention to high availability and disaster recovery.

The fundamental insight driving our architectural decisions is that **security and convenience exist in constant tension**. The most secure approach (air-gapped systems with manual key management) proves unusable for modern application development. The most convenient approach (hardcoded secrets) creates catastrophic security vulnerabilities. Our secret management system aims to find the optimal balance point.

**Dynamic vs Static Secret Models** represent a critical architectural distinction. Traditional approaches treat secrets as static configuration—passwords that change infrequently and must be distributed to applications. Dynamic secret systems generate unique, short-lived credentials on-demand for each application or session.

Consider database access patterns: static approaches use shared database credentials across all application instances. If those credentials leak, every application instance is compromised, and rotation requires coordinating updates across the entire fleet. Dynamic approaches generate unique database users for each application instance, with automatic expiration after configurable time periods. Credential leaks affect only single instances, and rotation happens automatically without coordination.

<b>Secret Model</b>	<b>Credential Sharing</b>	<b>Rotation Complexity</b>	<b>Blast Radius</b>	<b>Audit Granularity</b>
<b>Static</b>	Multiple applications share same credentials	High - requires coordination	High - affects all users	Low - can't distinguish application instances
<b>Dynamic</b>	Each application gets unique credentials	Low - automatic with TTL	Low - affects single application	High - individual application tracking

The mental model for dynamic secrets resembles a hotel key card system. Instead of giving every guest the same master key, hotels generate unique key cards for each guest's stay, programmed to work only for specific rooms during specific dates. When a guest checks out, their key card automatically becomes invalid. If a key card is lost or stolen, it affects only that guest's access, not the entire hotel's security.

**Implementation Complexity Considerations** influence the architectural approach significantly. Simple secret management implementations can start with encrypted configuration files and basic environment variable injection. However, this approach doesn't scale to enterprise requirements for audit logging, fine-grained access control, or dynamic secret generation.

Our architectural decision prioritizes **security depth over implementation simplicity**. While this increases initial development complexity, it provides a foundation for sophisticated security policies, comprehensive audit trails, and automated secret lifecycle management that prove essential for production environments and compliance requirements.

The subsequent sections detail how our secret management system addresses each challenge identified in this problem statement, building from basic encrypted storage through sophisticated high-availability clustering with automated failover and recovery capabilities.

## Implementation Guidance

### A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
HTTP Server	<code>net/http</code> with JSON	<code>gRPC</code> with Protocol Buffers
Storage Backend	Local filesystem with <code>os.File</code>	<code>etcd</code> or <code>consul</code> for distributed storage
Encryption	<code>crypto/aes</code> with <code>crypto/rand</code>	Hardware Security Modules (HSMs)
Authentication	Simple bearer tokens	<code>x/crypto/bcrypt</code> + JWT with <a href="https://github.com/golang-jwt/jwt">github.com/golang-jwt/jwt</a>
Logging	Standard library <code>log</code>	Structured logging with <a href="https://github.com/sirupsen/logrus">github.com/sirupsen/logrus</a>
Configuration	JSON files with <code>encoding/json</code>	YAML with <code>gopkg.in/yaml.v3</code>

## B. Recommended File/Module Structure

Organize your secret management system with clear separation of concerns from the beginning:

```
secret-manager/
├── cmd/
│   └── secret-server/
│       └── main.go           ← Entry point and server startup
├── internal/
│   ├── config/
│   │   ├── config.go         ← Configuration loading and validation
│   │   └── config_test.go
│   ├── storage/
│   │   ├── storage.go        ← Storage interface definition
│   │   ├── file_storage.go   ← Filesystem storage implementation
│   │   └── storage_test.go
│   ├── crypto/
│   │   ├── envelope.go       ← Envelope encryption implementation
│   │   ├── keys.go           ← Key generation and management
│   │   └── crypto_test.go
│   ├── auth/
│   │   ├── authenticator.go  ← Authentication interface
│   │   ├── token_auth.go     ← Token-based authentication
│   │   └── auth_test.go
│   ├── policy/
│   │   ├── evaluator.go      ← Policy evaluation engine
│   │   ├── policy.go          ← Policy data structures
│   │   └── policy_test.go
│   └── server/
│       ├── server.go          ← HTTP server and routing
│       ├── handlers.go        ← HTTP request handlers
│       └── server_test.go
└── pkg/
    └── client/
        ├── client.go          ← Client library for applications
        └── client_test.go
├── configs/
│   ├── server.yaml          ← Server configuration template
│   └── policies.yaml        ← Example policy configurations
└── scripts/
    ├── setup.sh              ← Development environment setup
    └── generate-certs.sh     ← TLS certificate generation
```

## C. Infrastructure Starter Code

Here's a complete configuration management system to handle server settings and policy loading:

GO

```
// internal/config/config.go

package config

import (
    "encoding/json"

    "fmt"

    "os"

    "time"
)

// ServerConfig holds all configuration for the secret management server

type ServerConfig struct {

    Server struct {
        Address     string      `json:"address"`
        Port        int         `json:"port"`
        TLSCertFile string      `json:"tls_cert_file"`
        TLSKeyFile  string      `json:"tls_key_file"`
        ReadTimeout time.Duration `json:"read_timeout"`
    } `json:"server"`

    Storage struct {
        Type string `json:"type"` // "file", "etcd", "consul"
        Path string `json:"path"` // For file storage
        URL  string `json:"url"` // For distributed storage
    } `json:"storage"`

    Encryption struct {
        MasterKeyFile string `json:"master_key_file"`
    }
}
```

```
        Algorithm      string `json:"algorithm"` // "aes-256-gcm"

    } `json:"encryption"`

Auth struct {

    Methods []string `json:"methods"` // ["token", "mtls"]

    TokenTTL time.Duration `json:"token_ttl"`

} `json:"auth"`

}

// LoadConfig reads configuration from the specified file path

func LoadConfig(configPath string) (*ServerConfig, error) {

    file, err := os.Open(configPath)

    if err != nil {

        return nil, fmt.Errorf("failed to open config file: %w", err)

    }

    defer file.Close()

    var config ServerConfig

    decoder := json.NewDecoder(file)

    if err := decoder.Decode(&config); err != nil {

        return nil, fmt.Errorf("failed to parse config file: %w", err)

    }

    if err := config.validate(); err != nil {

        return nil, fmt.Errorf("invalid configuration: %w", err)

    }

}
```

```
    return &config, nil
}

// validate ensures all required configuration fields are present and valid
func (c *ServerConfig) validate() error {
    if c.Server.Address == "" {
        c.Server.Address = "localhost"
    }

    if c.Server.Port == 0 {
        c.Server.Port = 8443
    }

    if c.Server.ReadTimeout == 0 {
        c.Server.ReadTimeout = 30 * time.Second
    }

    if c.Storage.Type == "" {
        c.Storage.Type = "file"
    }

    if c.Storage.Type == "file" && c.Storage.Path == "" {
        c.Storage.Path = "./data"
    }

    if c.Encryption.Algorithm == "" {
        c.Encryption.Algorithm = "aes-256-gcm"
    }

    if len(c.Auth.Methods) == 0 {
```

```
c.Auth.Methods = []string{"token"}  
}  
  
if c.Auth.TokenTTL == 0 {  
  
    c.Auth.TokenTTL = 24 * time.Hour  
}  
  
return nil  
}
```

Complete HTTP server foundation with security headers and logging:

GO

```
// internal/server/server.go

package server

import (
    "context"
    "fmt"
    "net/http"
    "time"

    "secret-manager/internal/config"
)

// SecretServer handles HTTP requests for the secret management system

type SecretServer struct {
    config *config.ServerConfig
    server *http.Server
    // TODO: Add encryption engine, auth engine, storage backend
}

// NewSecretServer creates a new secret management server

func NewSecretServer(cfg *config.ServerConfig) *SecretServer {
    mux := http.NewServeMux()

    server := &http.Server{
        Addr:         fmt.Sprintf("%s:%d", cfg.Server.Address, cfg.Server.Port),
        Handler:     securityMiddleware(loggingMiddleware(mux)),
        ReadTimeout: cfg.Server.ReadTimeout,
        WriteTimeout: cfg.Server.ReadTimeout,
```

```
    IdleTimeout:  cfg.Server.ReadTimeout * 2,
}

s := &SecretServer{

    config: cfg,
    server: server,
}

s.setupRoutes(mux)

return s
}

// setupRoutes configures all HTTP endpoints

func (s *SecretServer) setupRoutes(mux *http.ServeMux) {

    // Health check endpoint

    mux.HandleFunc("/v1/health", s.handleHealth)

    // Secret management endpoints (to be implemented in later milestones)

    mux.HandleFunc("/v1/secret/", s.handleSecret)

    mux.HandleFunc("/v1/auth/", s.handleAuth)

    mux.HandleFunc("/v1/policy/", s.handlePolicy)
}

// Start begins serving HTTP requests

func (s *SecretServer) Start() error {

    fmt.Printf("Secret server starting on %s\n", s.server.Addr)

    if s.config.Server.TLCertFile != "" && s.config.Server.TLSKeyFile != "" {
```

```
    return s.server.ListenAndServeTLS(s.config.Server.TLCertFile,
s.config.Server.TLSKeyFile)

}

return s.server.ListenAndServe()

}

// Shutdown gracefully stops the server

func (s *SecretServer) Shutdown(ctx context.Context) error {

    return s.server.Shutdown(ctx)

}

// Security middleware adds essential security headers

func securityMiddleware(next http.Handler) http.Handler {

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        w.Header().Set("X-Content-Type-Options", "nosniff")

        w.Header().Set("X-Frame-Options", "DENY")

        w.Header().Set("X-XSS-Protection", "1; mode=block")

        w.Header().Set("Strict-Transport-Security", "max-age=31536000; includeSubDomains")

        next.ServeHTTP(w, r)

    })

}

// Logging middleware logs all requests for audit purposes

func loggingMiddleware(next http.Handler) http.Handler {

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        start := time.Now()


```

```
next.ServeHTTP(w, r)

fmt.Printf("[%s] %s %s %s %v\n",
    time.Now().Format(time.RFC3339),
    r.Method,
    r.URL.Path,
    r.RemoteAddr,
    time.Since(start))

})
}
```

## D. Core Logic Skeleton Code

The main server initialization logic that you'll expand throughout the milestones:

```
// cmd/secret-server/main.go                                     GO

package main

import (
    "context"
    "flag"
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"

    "secret-manager/internal/config"
    "secret-manager/internal/server"
)

func main() {
    configPath := flag.String("config", "configs/server.json", "Path to configuration file")
    flag.Parse()

    // TODO 1: Load and validate configuration from file
    cfg, err := config.LoadConfig(*configPath)
    if err != nil {
        fmt.Printf("Failed to load configuration: %v\n", err)
        os.Exit(1)
    }

    // TODO 2: Initialize storage backend based on configuration
```

```
// This will be implemented in Milestone 1

// TODO 3: Initialize encryption engine with master key

// This will be implemented in Milestone 1

// TODO 4: Initialize authentication and policy engines

// This will be implemented in Milestone 2

// TODO 5: Initialize dynamic secret engines

// This will be implemented in Milestone 3

// TODO 6: Check if vault is sealed and require unsealing

// This will be implemented in Milestone 4

secretServer := server.NewSecretServer(cfg)

// Start server in background goroutine

go func() {
    if err := secretServer.Start(); err != nil {
        fmt.Printf("Server failed: %v\n", err)
        os.Exit(1)
    }
}()

// Wait for shutdown signal

sigChan := make(chan os.Signal, 1)

signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)
```

```

<-sigChan

    fmt.Println("Shutting down server...")

    ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)

    defer cancel()

    if err := secretServer.Shutdown(ctx); err != nil {

        fmt.Printf("Forced shutdown: %v\n", err)

        os.Exit(1)

    }

    fmt.Println("Server stopped gracefully")
}

```

## E. Language-Specific Hints

- **Configuration Management:** Use `encoding/json` for simple setups or `gopkg.in/yaml.v3` for more complex configurations with comments and multi-line strings
- **HTTP Security:** Always use TLS in production; the `crypto/tls` package provides excellent defaults with `tls.Config{MinVersion: tls.VersionTLS12}`
- **Error Handling:** Wrap errors with context using `fmt.Errorf("operation failed: %w", err)` to maintain error chains
- **Graceful Shutdown:** Use `context.WithTimeout()` for server shutdown to ensure connections close properly
- **File Operations:** Use `os.OpenFile()` with specific permissions (0600 for secret files) and always `defer file.Close()`
- **Memory Security:** Use `crypto/rand` for all random number generation; never use `math/rand` for security-sensitive operations

## F. Milestone Checkpoint

After implementing the foundational components described in this section, you should achieve these verification points:

### **Basic Server Startup:**

```
go run cmd/secret-server/main.go -config configs/server.json
```

BASH

Expected output:

```
Secret server starting on localhost:8443  
[2024-01-01T12:00:00Z] GET /v1/health 127.0.0.1:54321 2.1ms
```

### Health Check Verification:

```
curl -k https://localhost:8443/v1/health
```

BASH

Expected response:

```
{"status": "ok", "version": "1.0.0", "timestamp": "2024-01-01T12:00:00Z"}
```

JSON

**Configuration Validation:** Create a test configuration file with invalid settings to verify error handling works correctly. The server should refuse to start with clear error messages.

### Security Headers Verification:

```
curl -I -k https://localhost:8443/v1/health
```

BASH

Verify response includes security headers:

```
X-Content-Type-Options: nosniff  
X-Frame-Options: DENY  
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

### Signs of Problems and Diagnostics:

- **Server won't start:** Check configuration file syntax with `json.Valid()` or a JSON validator
- **TLS errors:** Verify certificate files exist and have correct permissions (readable by server process)
- **Port binding failures:** Ensure no other process is using the configured port with `netstat -tlnp | grep :8443`
- **Configuration not loading:** Add debug logging to see exactly which configuration values are being loaded

## Goals and Non-Goals

**Milestone(s):** This section establishes the foundation for all milestones by defining system scope and success criteria that guide implementation decisions.

Before diving into the technical architecture, we must clearly establish what this secret management system will and will not do. Think of this as the **system charter** — a contract between the development team and stakeholders that prevents scope creep while ensuring we build something genuinely useful. Just as a bank vault has specific purposes (storing valuables, controlling access) and explicit limitations (not a safety deposit box for oversized items), our secret management system must have clear boundaries.

The goals we define here directly influence every architectural decision throughout the project. They determine which security models we implement, what performance characteristics we optimize for, and which features we deliberately exclude to maintain focus and simplicity.

## Functional Goals

These are the **core capabilities** our secret management system must deliver to be considered successful. Each functional goal maps to specific milestones and acceptance criteria, ensuring our implementation stays focused on essential features.

### Secret Storage and Retrieval

The system must provide **secure, versioned storage** for sensitive data with strong encryption guarantees. Think of this as building a digital safe deposit box where each secret has its own compartment, complete history, and tamper-evident seals.

Capability	Description	Acceptance Criteria
Encrypted Storage	All secrets encrypted at rest using AES-256-GCM	No plaintext secrets in storage backend
Secret Versioning	Maintain complete history of secret value changes	Retrieve any previous version by ID
Key Rotation	Replace encryption keys without service interruption	All secrets re-encrypted with new keys
Atomic Operations	Secret updates succeed completely or fail completely	No partial writes during failures

The **envelope encryption** model forms the security foundation here. The master key encrypts data encryption keys, which in turn encrypt individual secrets. This creates multiple security layers — even if storage is compromised, secrets remain protected by the master key hierarchy.

## Decision: Envelope Encryption Architecture

- **Context:** Secrets need encryption at rest, but managing individual keys per secret creates complexity
- **Options Considered:**
  1. Single master key encrypts all secrets directly
  2. Envelope encryption with master key protecting data keys
  3. Per-secret encryption keys stored separately
- **Decision:** Envelope encryption with hierarchical key structure
- **Rationale:** Provides key rotation without re-encrypting all data, isolates key management from data encryption, enables fine-grained access control
- **Consequences:** Adds complexity but enables secure key rotation and better security isolation

## Access Control and Authentication

The system must implement **zero-trust authentication** and **path-based authorization** to ensure only authorized clients can access specific secrets. Consider this like a bank's security model — every person must identify themselves, and their identity determines which safe deposit boxes they can access.

Authentication Method	Use Case	Security Properties
Token-based	Service-to-service communication	Revocable, time-limited, auditable
Mutual TLS	High-security environments	Certificate-based identity, network encryption
AppRole	CI/CD and automated systems	Role-based with secret delivery separation

The **policy evaluation engine** implements path-based access control using glob patterns. For example, a policy might grant the `web-service` role access to `secrets/production/web/*` but deny access to `secrets/production/database/*`. This fine-grained control ensures secrets follow the principle of least privilege.

## Decision: Path-Based Access Control

- **Context:** Need flexible authorization that scales with organizational structure
- **Options Considered:**
  1. Role-based access control with fixed permission sets
  2. Path-based ACLs with pattern matching
  3. Attribute-based access control with complex rules
- **Decision:** Path-based ACLs with glob pattern support
- **Rationale:** Maps naturally to secret organization, easy to understand and audit, performant evaluation
- **Consequences:** Enables intuitive permission models but requires careful pattern design to avoid overly broad access

## Dynamic Secret Generation

The system must generate **short-lived credentials** on demand for external systems like databases and cloud providers. Think of this as a **credential vending machine** — insert a valid request token, receive fresh credentials that automatically expire.

Backend Type	Generated Credential	Lifecycle Management
Database	Username/password with limited privileges	Auto-revocation on lease expiry
Cloud Provider	API keys with scoped permissions	Cleanup via provider APIs
SSH	Certificate-based access	Certificate expiration handling

The **lease management system** tracks every generated credential with time-to-live (TTL) limits. Background processes automatically revoke expired credentials, preventing credential accumulation in external systems. This solves the credential rotation problem by making credentials inherently temporary.

## High Availability and Disaster Recovery

The system must operate reliably in distributed environments with **automatic failover** and **data durability** guarantees. Picture this as building multiple bank branches that stay synchronized — if one location fails, others continue serving customers with complete access to their accounts.

Availability Feature	Implementation	Recovery Time
Leader Election	Raft consensus algorithm	< 30 seconds failover
Data Replication	Write-ahead logging with quorum	Zero data loss with majority
Backup and Restore	Encrypted backup snapshots	Point-in-time recovery
Auto-unseal	Cloud KMS integration	Automatic startup after restart

## Non-Functional Goals

These **quality attributes** define how well the system performs its functional capabilities. Non-functional goals often drive architectural decisions more than functional requirements — they determine whether we choose simplicity or performance, consistency or availability.

## Security Properties

The system must maintain **defense in depth** with multiple security layers protecting against various attack vectors. Security isn't a feature we add later — it's a foundational property that influences every design decision.

Security Property	Target	Measurement
Encryption at Rest	AES-256-GCM for all stored data	No plaintext secrets in storage
Encryption in Transit	TLS 1.3 for all client communication	Certificate validation required
Authentication	Multi-factor verification	Token + certificate validation
Audit Logging	Complete access trail	Every secret operation logged
Memory Protection	Clear sensitive data after use	Zero sensitive data in memory dumps

The critical security principle here is **assume breach** — design as if attackers will eventually access the storage backend, network traffic, or even application memory. Every protection layer should remain effective even if other layers are compromised.

## Performance and Scalability

The system must handle production workloads with **predictable latency** and **horizontal scaling** capabilities. Performance goals must be realistic — we're not building a high-frequency trading system, but we need consistent response times for application startup and credential rotation.

Performance Metric	Target	Rationale
Secret Retrieval Latency	< 100ms P95	Acceptable for application startup
Dynamic Secret Generation	< 500ms P95	Database connection establishment time
Throughput	1000 requests/second/node	Supports moderate application load
Storage Efficiency	< 10KB overhead per secret	Reasonable metadata storage cost

## Operational Simplicity

The system must be **operationally manageable** with clear deployment procedures, monitoring capabilities, and troubleshooting guides. Complex systems that work perfectly in development often fail in production due to operational complexity.

Operational Aspect	Requirement	Benefit
Single Binary Deployment	No external runtime dependencies	Simplified installation and updates
Configuration Management	File-based with validation	Version control and audit trails
Health Check Endpoints	HTTP endpoints for load balancers	Automated health monitoring
Structured Logging	JSON format with correlation IDs	Centralized log aggregation
Metrics Export	Prometheus-compatible metrics	Performance monitoring and alerting

## Explicit Non-Goals

These are **intentional limitations** we accept to maintain focus and avoid over-engineering. Explicitly stating what we won't build is as important as defining what we will build — it prevents feature creep and keeps the implementation manageable.

## Enterprise Features (Out of Scope)

We deliberately exclude enterprise-grade features that add significant complexity without providing core value for the learning objectives.

Excluded Feature	Rationale	Alternative
Multi-tenancy	Adds namespace isolation complexity	Deploy separate instances per tenant
LDAP/Active Directory	Complex integration with legacy systems	Focus on token and certificate auth
Hardware Security Modules	Requires specialized hardware	Use cloud KMS for production
GUI/Web Interface	Frontend development outside scope	CLI and REST API provide full functionality
Plugin Architecture	Dynamic loading adds security and complexity	Built-in backends only

### Decision: No Plugin Architecture

- **Context:** Extensibility through plugins versus built-in secret backends
- **Options Considered:**
  1. Dynamic plugin loading with shared libraries
  2. Compile-time plugin registration
  3. Fixed set of built-in backends
- **Decision:** Built-in backends only (database, cloud providers)
- **Rationale:** Avoids security risks of dynamic code loading, reduces operational complexity, sufficient for learning objectives
- **Consequences:** Less extensible but more secure and simpler to operate

### Advanced Cryptographic Features

We focus on proven, standard cryptographic primitives rather than implementing cutting-edge or specialized crypto features.

Excluded Crypto Feature	Rationale	Standard Alternative
Zero-Knowledge Proofs	Complex math, limited practical benefit	Standard authentication tokens
Homomorphic Encryption	Academic complexity, performance issues	Decrypt for processing
Custom Cipher Implementations	High risk of implementation bugs	Standard library AES-GCM
Post-Quantum Cryptography	Standards still evolving	Current NIST recommendations

### Performance Optimization (Deferred)

We prioritize correctness and security over extreme performance optimization. A working, secure system is more valuable than a fast, broken one.

Optimization Not Implemented	Rationale	Acceptable Trade-off
Connection Pooling	Adds complexity to backend implementations	Slight performance overhead acceptable
Caching Layer	Cache invalidation and consistency challenges	Acceptable latency increase
Horizontal Sharding	Distributed system complexity	Single cluster handles target load
Async Processing	Complicates error handling and consistency	Synchronous operations simpler

## Cloud-Specific Features

We build a cloud-agnostic system rather than optimizing for specific cloud provider features.

Cloud Feature	Exclusion Reason	Generic Alternative
AWS IAM Integration	Vendor lock-in, complex permission mapping	Generic token authentication
GCP Service Accounts	Platform-specific authentication flow	Mutual TLS certificates
Azure Key Vault	Managed service, reduces learning value	Self-managed encryption
Kubernetes Secrets	Platform coupling, different security model	HTTP API works with any orchestrator

The philosophy here is **build once, deploy anywhere**. While cloud-native features provide operational benefits, they couple the system to specific platforms and reduce the educational value of implementing core secret management concepts.

## Success Criteria and Validation

To ensure we meet our goals, we define **measurable success criteria** for each functional goal and **validation procedures** to verify achievement.

Goal Category	Success Metric	Validation Method
Security	All secrets encrypted with AES-256-GCM	Audit storage backend for plaintext
Access Control	Policy violations return 403 Forbidden	Attempt unauthorized secret access
Dynamic Secrets	Database credentials expire within TTL	Monitor external system user tables
High Availability	< 30 second failover during leader failure	Kill leader node, measure recovery time
Performance	Secret retrieval under 100ms P95	Load testing with realistic workloads

## Implementation Guidance

This guidance helps translate our high-level goals into concrete development tasks and validation steps.

## Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	<code>net/http</code> with middleware	<code>gin-gonic/gin</code> framework
Configuration	<code>encoding/json</code> with structs	<code>spf13/viper</code> with validation
Database Backend	<code>database/sql</code> with SQLite	PostgreSQL with connection pooling
Testing Framework	Standard <code>testing</code> package	<code>stretchr/testify</code> with assertions
Logging	Standard <code>log/slog</code> package	<code>uber-go/zap</code> structured logging

## Goal Validation Checklist

Use this checklist after each milestone to verify you're meeting the established goals:

### Milestone 1 Validation (Encrypted Storage):

```
# Verify encryption at rest                                BASH
go test ./internal/encryption -v

# Check no plaintext in storage
strings vault.db | grep -v "expected_metadata" | head -20

# Should show only encrypted binary data, no readable secrets
```

### Milestone 2 Validation (Access Control):

```
# Test unauthorized access returns 403                                BASH
curl -k https://localhost:8443/v1/secrets/test

# Should return: {"error": "missing or invalid token"}


# Test policy enforcement
curl -k -H "X-Vault-Token: invalid" https://localhost:8443/v1/secrets/test

# Should return: {"error": "permission denied"}
```

### Milestone 3 Validation (Dynamic Secrets):

```
# Generate database credentials                                BASH

curl -k -H "X-Vault-Token: $ROOT_TOKEN" \
      https://localhost:8443/v1/database/creds/my-role

# Should return: {"username": "v-root-my-role-XYZ", "password": "...", "lease_id": "..."}\n\n# Verify TTL enforcement (wait for expiration)

# Check that generated user is removed from database
```

#### Milestone 4 Validation (High Availability):

```
# Test unsealing with key shares                                BASH

./vault operator unseal $SHARE_1

./vault operator unseal $SHARE_2

./vault operator unseal $SHARE_3

# Should transition from sealed to active state

# Test failover (requires multi-node setup)

# Kill leader, verify standby promotion within 30 seconds
```

### Common Implementation Pitfalls

**⚠ Pitfall: Scope Creep During Development** It's tempting to add "just one more feature" when you see how easy it would be to implement. Resist this urge — every additional feature adds complexity, testing overhead, and potential security vulnerabilities. Stick to the defined functional goals and defer enhancements until the core system is solid.

#### ⚠ Pitfall: Premature Performance Optimization

Don't implement caching, connection pooling, or async processing until you've measured actual performance problems. These optimizations add complexity that makes debugging harder and often introduce subtle bugs. Build a correct, simple system first, then optimize specific bottlenecks with measurement data.

**⚠ Pitfall: Security Theater vs. Real Security** Avoid implementing security features that look impressive but don't address real threats. For example, custom encryption algorithms or complex authentication schemes often introduce vulnerabilities. Focus on proven security practices: standard crypto libraries, defense in depth, and comprehensive audit logging.

**⚠ Pitfall: Ignoring Operational Requirements** A system that works perfectly on your laptop but requires complex deployment procedures will fail in production. Design for operations from the beginning: single binary

deployment, clear configuration, health checks, and structured logging. These aren't "nice to have" features — they're essential for any production system.

## File Structure for Goal-Driven Development

Organize your code to reflect the functional goals, making it easy to work on one capability at a time:

```
vault-system/
├── cmd/vault/main.go           ← Single binary entry point
├── internal/
│   ├── server/
│   │   ├── server.go
│   │   └── handlers.go
│   ├── encryption/
│   │   ├── envelope.go
│   │   └── rotation.go
│   ├── auth/
│   │   ├── tokens.go
│   │   └── policies.go
│   ├── dynamic/
│   │   ├── database.go
│   │   └── leases.go
│   ├── unsealing/
│   │   ├── shamir.go
│   │   └── consensus.go
│   └── storage/
│       └── backend.go
└── configs/
    └── vault.json
└── docs/
    └── testing.md

```

This structure makes it clear which code relates to which functional goal, enabling focused development and easier validation of individual capabilities.

## High-Level Architecture

**Milestone(s):** This section provides the architectural foundation for all four milestones, establishing the component structure and security boundaries that will guide implementation across encrypted storage (Milestone 1), access control (Milestone 2), dynamic secrets (Milestone 3), and high availability (Milestone 4).

The high-level architecture of our secret management system follows a **defense in depth** security model, where multiple layers of protection work together to create a **zero-trust system**. Think of it like a modern bank: the vault has multiple independent security systems (cameras, motion sensors, time locks, armed guards), and each system can detect different types of threats. If one system fails, the others continue protecting the assets.

Our secret management system employs the same principle with four distinct subsystems, each responsible for a specific aspect of security. The encryption engine ensures that even if someone gains access to the storage, they cannot read the secrets. The authentication and authorization engine ensures that only verified identities with proper permissions can access secrets. The dynamic secret engine minimizes exposure by generating short-lived credentials. The unsealing mechanism ensures that the system cannot operate without human authorization, preventing automated attacks.

## Component Overview

The secret management system is built around four main subsystems that work together to provide comprehensive secret protection. Each subsystem has clear responsibilities and operates with minimal trust assumptions about the others.

### The Four Core Subsystems:

Subsystem	Primary Responsibility	Security Purpose	Data Owned
API Server	HTTP request handling and routing	Entry point validation and protocol termination	Request/response formatting, TLS certificates
Encryption Engine	Secret encryption, decryption, and key management	Confidentiality and integrity of stored data	Master keys, data encryption keys, encrypted secrets
Authentication & Authorization Engine	Identity verification and policy enforcement	Access control and audit trail	Policies, tokens, identity mappings, audit logs
Storage Backend	Persistent data storage and retrieval	Durability and consistency of encrypted data	Raw encrypted data, metadata, configuration

Think of these subsystems like the departments in a secure facility. The **API Server** is like the front desk—it receives all requests, verifies that visitors are following proper protocols, and routes them to the appropriate department. It doesn't make security decisions but ensures that all communication follows established procedures.

The **Encryption Engine** is like the vault mechanism itself—it knows how to lock and unlock the safes, manages the combination codes, and ensures that anything stored is properly secured. It never decides who can access what, but it guarantees that unauthorized access is impossible even if someone breaks into the storage room.

The **Authentication & Authorization Engine** is like the security department—it maintains the list of authorized personnel, checks credentials, and decides whether each person should have access to specific areas. It creates audit trails but doesn't handle the actual storage or encryption of assets.

The **Storage Backend** is like the physical infrastructure—the walls, floors, and filing systems that hold everything. It provides reliable storage but doesn't understand what it's storing or who should access it.

**Design Insight:** This separation follows the principle of **single responsibility** at the subsystem level. Each component can be secured, tested, and potentially replaced independently. For example, we could swap from a local file storage backend to a distributed database without changing the encryption or authorization logic.

### Detailed Component Responsibilities:

The **API Server** subsystem handles all external communication and protocol concerns. It terminates TLS connections, parses HTTP requests, validates request formats, and serializes responses. This component implements rate limiting, request logging, and connection management. It routes authenticated requests to appropriate internal subsystems but never accesses secret data directly. The API server also handles the system's sealed/unsealed state, rejecting most operations when the system is sealed.

The **Encryption Engine** subsystem manages all cryptographic operations using **envelope encryption**. It maintains the master key hierarchy, generates and rotates data encryption keys, and performs AES-256-GCM encryption/decryption operations. This component handles secret versioning, allowing multiple versions of the same secret path to coexist during key rotation. It also manages secure memory operations, ensuring that plaintext secrets and encryption keys are properly zeroed after use.

The **Authentication & Authorization Engine** subsystem implements a complete identity and access management system. It supports multiple authentication methods (tokens, mutual TLS, AppRole), maintains policy definitions with path-based access control, and evaluates authorization requests. This component generates and validates access tokens, maintains session state, and produces comprehensive audit logs. It implements the policy evaluation engine that matches requests against defined rules.

The **Storage Backend** subsystem provides persistent, reliable storage for all system data. It stores encrypted secrets, policy definitions, token metadata, and system configuration. This component handles atomic operations, consistency guarantees, and backup/restore functionality. It implements key-value storage with support for transactions and range queries needed by higher-level components.

## Decision: Component Communication Model

- **Context:** The four subsystems need to communicate reliably while maintaining security boundaries and enabling independent testing.
- **Options Considered:**
  1. Direct function calls within a monolithic process
  2. Message passing through internal channels
  3. Internal API calls over localhost HTTP
- **Decision:** Direct function calls with well-defined interfaces
- **Rationale:** Function calls provide the lowest latency and highest reliability for internal communication. They enable atomic operations across subsystems and simplify error handling. Interface-based design still allows for independent testing through mocking.
- **Consequences:** All subsystems must run in the same process, but we gain performance and transactional consistency. Future scaling would require refactoring to a distributed model.

Communication Option	Latency	Reliability	Testability	Complexity	Chosen?
Direct function calls	Lowest	Highest	Good (mockable interfaces)	Low	✓
Message passing	Medium	Good	Excellent	Medium	✗
Internal HTTP	Highest	Medium	Excellent	High	✗

## Security Boundaries

The secret management system implements multiple security boundaries to create **defense in depth**. Think of these boundaries like the security zones in a government facility—each zone has different clearance requirements, and moving between zones requires additional verification. Even if an attacker compromises one zone, they still cannot access resources in higher-security zones.

### Trust Zones and Privilege Separation:

Trust Zone	Components	Data Access	Privilege Level	Threat Model
External Zone	Client applications, operators	None (encrypted requests only)	No privileges	Untrusted network, compromised clients
API Boundary	API Server, TLS termination	Request metadata, routing info	Limited (protocol handling)	Network attacks, protocol exploits
Application Zone	Auth Engine, API routing	Policies, tokens, encrypted secrets	Medium (business logic)	Logic bugs, injection attacks
Cryptographic Zone	Encryption Engine	Plaintext secrets, encryption keys	Highest (key material)	Memory dumps, side-channel attacks
Storage Zone	Storage Backend	Raw encrypted data only	Low (persistence only)	Storage compromise, backup theft

The **External Zone** represents completely untrusted territory. Client applications, even those with valid credentials, operate in this zone. All communication crosses the network and is assumed to be observable by attackers. Clients receive only encrypted responses and must prove their identity for every request.

The **API Boundary** is the first line of defense within the system. The API Server component operates here, handling TLS termination and basic request validation. This component can see request metadata and routing information but never accesses plaintext secrets. It implements rate limiting and basic attack detection, protecting the inner zones from malicious or malformed requests.

The **Application Zone** contains the business logic components: the Authentication & Authorization Engine and API request routing. Components in this zone can access policy information, token metadata, and encrypted secret data, but they cannot decrypt secrets without coordinating with the Cryptographic Zone. This separation ensures that authorization bugs cannot directly expose plaintext secret data.

The **Cryptographic Zone** represents the highest privilege level within the system. Only the Encryption Engine operates here, and it has access to master keys and can produce plaintext secrets. This component uses secure memory handling, implements constant-time operations where appropriate, and minimizes the lifetime of plaintext data in memory.

The **Storage Zone** operates with intentionally limited privileges. The Storage Backend can persist and retrieve data but cannot interpret the encrypted contents. This ensures that storage compromise or backup theft cannot directly expose secret values—the attacker would also need access to the encryption keys from the Cryptographic Zone.

### Inter-Zone Communication Security:

Communication between zones follows strict protocols designed to minimize privilege escalation risks. The API Server validates all external requests before forwarding them to internal components. The Authentication Engine verifies identity and authorization before allowing any secret access. The Encryption Engine only decrypts secrets after receiving valid authorization tokens from the Authentication Engine.

**Critical Security Principle: Assume breach** at every boundary. Each zone assumes that external zones may be compromised and implements defenses accordingly. The Encryption Engine assumes that API requests might be forged, so it requires cryptographically signed authorization tokens. The Storage Backend assumes that application logic might be compromised, so it never stores plaintext data.

### Memory Protection and Data Handling:

Within each zone, components implement appropriate data protection measures. The Cryptographic Zone uses secure memory allocation and explicit zeroing of sensitive data. The Application Zone implements constant-time comparison for token validation to prevent timing attacks. The Storage Zone ensures that temporary buffers used for I/O operations are cleared after use.

Security Boundary	Protection Mechanism	Attack Prevention	Recovery Method
Network boundary	Mutual TLS, certificate validation	Man-in-the-middle, eavesdropping	Certificate rotation, connection retry
Process boundary	Memory isolation, secure allocation	Memory dumps, cross-process access	Process restart, memory encryption
Component boundary	Interface contracts, input validation	Injection, privilege escalation	Component isolation, audit logging
Storage boundary	Encryption at rest, access controls	Storage compromise, backup theft	Key rotation, encrypted backups

## Deployment Topology

The secret management system supports multiple deployment topologies to balance security, availability, and operational requirements. The architecture accommodates everything from single-node development environments to highly available production clusters with geographic distribution.

### Single Node Deployment:

For development and small-scale production environments, all four subsystems run within a single process on one machine. Think of this like a small bank branch where one building contains the vault, teller windows, security office, and records storage. Everything is co-located, which simplifies operations but creates a single point of failure.

Component	Process Location	Data Storage	Network Access
API Server	Main process (port 8443)	None (stateless)	External clients
Encryption Engine	Main process (in-memory)	Local file system	Internal only
Auth & Policy Engine	Main process (in-memory)	Local file system	Internal only
Storage Backend	Main process (file I/O)	Local disk (encrypted)	Internal only

The single-node deployment stores the master key on local disk, protected by file system permissions and encryption. Secret data, policies, and audit logs are stored as encrypted files in a configurable directory structure. This topology provides the simplest operational model but requires careful backup procedures since all system state exists on one machine.

#### High Availability Cluster Deployment:

For production environments requiring high availability, the system can operate as a cluster with multiple nodes. This resembles a banking network where multiple branches can serve customers, but they coordinate through a central system to maintain consistency. One node acts as the active leader while others remain on standby, ready to take over if the leader fails.

Node Role	Active Components	Data Replication	Client Access
Leader	All four subsystems	Writes to storage, replicates to followers	Serves all requests
Follower	Storage Backend, limited API Server	Receives replicated data	Read-only operations (optional)
Standby	All subsystems (inactive)	Maintains current state	No client access

The high availability deployment uses a **consensus protocol** to coordinate between nodes and ensure that only one leader is active at a time. This prevents **split-brain scenarios** where multiple nodes might accept conflicting write operations. The leader performs all cryptographic operations and policy decisions, while follower nodes maintain synchronized copies of the encrypted data.

## Decision: Leader-Follower vs Active-Active Architecture

- **Context:** High availability requires coordination between multiple nodes, but secret management has strict consistency requirements.
- **Options Considered:**
  1. Active-active with conflict resolution
  2. Leader-follower with automatic failover
  3. Manual failover with cold standby
- **Decision:** Leader-follower with automatic failover
- **Rationale:** Secret management requires strict consistency—conflicting policy changes or overlapping dynamic secret generation could create security vulnerabilities. Leader-follower ensures linearizable operations while still providing automatic recovery.
- **Consequences:** Write operations are limited to leader capacity, but we maintain consistency guarantees and enable automated failover.

## Network Architecture and Client Routing:

Clients connect to the cluster through a **load balancer** or **service discovery mechanism** that routes requests to the current leader. The load balancer performs health checks to detect leader changes and automatically redirects traffic during failover events. Follower nodes can optionally serve read-only requests for policies and audit logs, reducing load on the leader.

Traffic Type	Routing Strategy	Consistency Level	Failover Behavior
Secret read/write	Leader only	Strict consistency	Redirect to new leader
Policy evaluation	Leader preferred, follower acceptable	Eventually consistent	Best-effort routing
Audit log queries	Any node	Eventually consistent	Round-robin distribution
Health checks	All nodes	Local state	Per-node response

The network topology implements **defense in depth** at the infrastructure level. All inter-node communication uses mutual TLS with node-specific certificates. Client connections require valid certificates and are rate-limited per identity. Network segmentation isolates the secret management cluster from other systems, with firewall rules allowing only necessary ports.

## Data Distribution and Replication:

The cluster maintains **strong consistency** for critical data while allowing **eventual consistency** for less sensitive information. Secret data, policies, and encryption keys are replicated synchronously to ensure that failover doesn't lose recent changes. Audit logs and configuration data can be replicated asynchronously to improve write performance.

Data Type	Replication Mode	Consistency Requirement	Storage Location
Secrets (encrypted)	Synchronous	Strong consistency	Leader + all followers
Encryption keys	Synchronous	Strong consistency	Leader + all followers
Policies	Synchronous	Strong consistency	Leader + all followers
Tokens/sessions	Synchronous	Strong consistency	Leader + all followers
Audit logs	Asynchronous	Eventually consistent	All nodes + external system
Configuration	Asynchronous	Eventually consistent	All nodes

### Disaster Recovery and Backup Strategy:

The deployment topology supports multiple levels of backup and recovery. Regular **encrypted backups** capture the complete system state, including all secrets, policies, and configuration. These backups are encrypted with separate keys and stored in geographically distributed locations. For critical environments, **cross-region replication** maintains live copies of the data in multiple data centers.

The **unsealing process** is designed to work with the distributed topology. Shamir's secret sharing key shares are distributed among multiple operators across different locations. During disaster recovery, operators can unseal a restored cluster without requiring access to the original master key material, since the key is reconstructed from the distributed shares.

**Operational Insight:** The deployment topology directly impacts the unsealing workflow. In a single-node deployment, all key shares might be held by operators in one location. In a distributed deployment, key shares should be distributed geographically to ensure that natural disasters or regional outages don't prevent unsealing operations.

### Scaling Characteristics and Resource Requirements:

Different deployment topologies have distinct scaling characteristics and resource requirements. The single-node deployment scales vertically by adding CPU and memory to handle more concurrent requests. The cluster deployment scales primarily through read replicas, since write operations are constrained by the leader's capacity and the consensus protocol overhead.

<b>Deployment Type</b>	<b>CPU Utilization</b>	<b>Memory Requirements</b>	<b>Storage I/O</b>	<b>Network Bandwidth</b>
Single node	Moderate (all operations)	High (in-memory caching)	High (all writes local)	Low (no replication)
HA leader	High (all write operations)	High (in-memory state)	High (writes + replication)	High (replication traffic)
HA follower	Low (replication only)	Medium (replicated state)	Medium (replicated writes)	Medium (replication inbound)

## Implementation Guidance

The high-level architecture maps to a modular Go codebase that separates concerns while enabling efficient communication between subsystems. Each subsystem is implemented as a separate package with well-defined interfaces, making the system testable and maintainable.

### Technology Recommendations:

<b>Component</b>	<b>Simple Option</b>	<b>Advanced Option</b>
HTTP Server	net/http with gorilla/mux routing	Custom HTTP/2 server with connection pooling
Storage Backend	Local JSON files with file locking	Distributed key-value store (etcd, Consul)
Cryptography	Go crypto/aes and crypto/rand	Hardware security module (HSM) integration
Clustering	Single node with file persistence	Raft consensus with distributed storage
TLS/Authentication	Go crypto/tls with static certificates	Dynamic certificate management with rotation
Configuration	YAML files with validation	Dynamic configuration with hot reloading

### Recommended File Structure:

The codebase follows Go's standard project layout with clear separation between public APIs, internal implementation, and supporting tools:

```

secret-vault/
├── cmd/
│   ├── server/
│   │   └── main.go           ← Server entry point, configuration loading
│   └── cli/
│       └── main.go          ← Client CLI tool for operators
├── internal/
│   ├── server/
│   │   ├── server.go         ← SecretServer implementation
│   │   └── config.go         ← ServerConfig and LoadConfig
│   ├── api/
│   │   ├── handlers.go       ← HTTP request handlers
│   │   ├── middleware.go     ← Authentication, logging middleware
│   │   └── routes.go         ← Route definitions and registration
│   ├── encryption/
│   │   ├── engine.go          ← Envelope encryption implementation
│   │   ├── keys.go            ← Key generation and rotation
│   │   └── versioning.go      ← Secret version management
│   ├── auth/
│   │   ├── engine.go          ← Authentication and authorization
│   │   ├── policies.go        ← Policy definition and evaluation
│   │   ├── tokens.go          ← Token generation and validation
│   │   └── audit.go           ← Audit logging
│   ├── storage/
│   │   ├── backend.go         ← Storage interface definition
│   │   ├── file.go            ← File-based storage implementation
│   │   └── memory.go          ← In-memory storage for testing
│   ├── dynamic/
│   │   ├── engine.go          ← Dynamic secret generation
│   │   ├── database.go        ← Database credential backend
│   │   └── leases.go          ← Lease management and TTL tracking
│   └── unseal/
│       ├── shamir.go          ← Shamir's secret sharing
│       ├── seal.go            ← Seal/unseal operations
│       └── ha.go               ← High availability and consensus
└── pkg/
    └── client/
        └── client.go          ← Go client library for applications
configs/
├── dev.yaml          ← Development configuration
└── prod.yaml          ← Production configuration template
scripts/
├── generate-certs.sh  ← TLS certificate generation
└── init-cluster.sh    ← Cluster initialization

```

## Core Configuration Structure:

The system configuration uses a hierarchical structure that maps to the component architecture:

```
// ServerConfig holds all configuration for the secret management server
```

GO

```
type ServerConfig struct {

    Server     ServerSettings      `yaml:"server"`

    Storage   StorageSettings     `yaml:"storage"`

    Encryption EncryptionSettings `yaml:"encryption"`

    Auth      AuthSettings       `yaml:"auth"`

    Cluster   ClusterSettings    `yaml:"cluster"`

}
```

```
type ServerSettings struct {

    Port          int             `yaml:"port"`

    TLSCertPath   string          `yaml:"tls_cert_path"`

    TLSKeyPath    string          `yaml:"tls_key_path"`

    ReadTimeout   time.Duration  `yaml:"read_timeout"`

    WriteTimeout  time.Duration  `yaml:"write_timeout"`

}
```

```
type StorageSettings struct {

    Backend   string           `yaml:"backend"` // "file", "memory", "etcd"

    Config    map[string]string `yaml:"config"` // Backend-specific settings

    BackupDir string          `yaml:"backup_dir"`

}
```

```
type EncryptionSettings struct {

    KeySize      int      `yaml:"key_size"` // 256 for AES-256

    MasterKeyPath string  `yaml:"master_key_path"`

    AutoRotate   bool    `yaml:"auto_rotate"`

}
```

```

type AuthSettings struct {

    Methods      []string      `yaml:"methods"`      // ["token", "mtls", "approle"]

    TokenTTL     time.Duration `yaml:"token_ttl"`

    PolicyDir    string        `yaml:"policy_dir"`

    AuditLogPath string        `yaml:"audit_log_path"`

}

type ClusterSettings struct {

    Enabled      bool          `yaml:"enabled"`

    NodeID       string        `yaml:"node_id"`

    Peers        []string      `yaml:"peers"`

    DataDir      string        `yaml:"data_dir"`

    BindAddress  string        `yaml:"bind_address"`

}

```

### Main Server Implementation Skeleton:

GO

```
// SecretServer represents the main secret management server with all subsystems

type SecretServer struct {

    config      ServerConfig

    httpServer  *http.Server


    // Core subsystems - each implemented as a separate component

    storage     storage.Backend

    encryption  *encryption.Engine

    auth        *auth.Engine

    dynamic     *dynamic.Engine

    unseal      *unseal.Manager


    // Server state

    isSealed    bool

    sealMutex   sync.RWMutex

}

// LoadConfig reads and validates configuration from the specified file path

func LoadConfig(configPath string) (ServerConfig, error) {

    // TODO 1: Read YAML file from configPath

    // TODO 2: Parse YAML into ServerConfig struct

    // TODO 3: Validate all required fields are present

    // TODO 4: Apply default values for optional fields (DEFAULT_PORT, DEFAULT_TIMEOUT)

    // TODO 5: Validate that certificate files exist if TLS is enabled

    // TODO 6: Return validated configuration or descriptive error

}

// NewSecretServer creates a new server instance with the provided configuration
```

```
func NewSecretServer(cfg ServerConfig) (*SecretServer, error) {

    // TODO 1: Initialize storage backend based on cfg.Storage.Backend

    // TODO 2: Create encryption engine with master key from cfg.Encryption

    // TODO 3: Initialize auth engine with policy directory from cfg.Auth

    // TODO 4: Create dynamic secret engine for credential generation

    // TODO 5: Initialize unseal manager for Shamir's secret sharing

    // TODO 6: Create HTTP server with TLS configuration

    // TODO 7: Register HTTP routes connecting to subsystem handlers

    // TODO 8: Return configured SecretServer instance

    //

    // Hint: Server starts in sealed state - unseal operation required before serving
    secrets

}

// Start begins serving HTTP requests on the configured port

func (s *SecretServer) Start() error {

    // TODO 1: Verify server is properly configured (all subsystems initialized)

    // TODO 2: Start background processes (lease renewal, audit log rotation)

    // TODO 3: Begin listening on configured port with TLS

    // TODO 4: Log startup message with server version and configuration summary

    //

    // Note: Server will reject secret operations until unsealed, but health checks work

}

// Shutdown gracefully stops the server and cleans up resources

func (s *SecretServer) Shutdown(ctx context.Context) error {

    // TODO 1: Stop accepting new HTTP connections

    // TODO 2: Wait for existing requests to complete (up to context deadline)
```

```
// TODO 3: Stop background processes (lease management, audit logging)

// TODO 4: Seal the server (clear encryption keys from memory)

// TODO 5: Close storage backend and flush any pending writes

// TODO 6: Return any errors encountered during shutdown

}
```

### **Subsystem Interface Definitions:**

Each subsystem exposes a clean interface that abstracts its internal complexity:

GO

```
// Storage backend interface - abstracts persistence mechanism

type Backend interface {

    Get(ctx context.Context, key string) ([]byte, error)

    Put(ctx context.Context, key string, value []byte) error

    Delete(ctx context.Context, key string) error

    List(ctx context.Context, prefix string) ([]string, error)

    Transaction(ctx context.Context, ops []Operation) error

}

// Encryption engine interface - handles all cryptographic operations

type Engine interface {

    Encrypt(ctx context.Context, plaintext []byte, path string) (*EncryptedData, error)

    Decrypt(ctx context.Context, encrypted *EncryptedData) ([]byte, error)

    RotateKeys(ctx context.Context) error

    GetKeyVersion(ctx context.Context) (int, error)

}

// Authentication engine interface - identity and access control

type Engine interface {

    Authenticate(ctx context.Context, credentials interface{}) (*Identity, error)

    Authorize(ctx context.Context, identity *Identity, action string, path string) error

    CreateToken(ctx context.Context, identity *Identity, ttl time.Duration) (string, error)

    ValidateToken(ctx context.Context, token string) (*Identity, error)

}
```

## Constants and Defaults:

```

const (
    DEFAULT_PORT      = 8443                      // HTTPS port for secret management
    DEFAULT_TIMEOUT   = 30 * time.Second           // HTTP request timeout

    // Encryption constants
    AES_KEY_SIZE     = 32 // 256 bits for AES-256
    GCM_NONCE_SIZE   = 12 // 96 bits for AES-GCM

    // Authentication constants
    TOKEN_HEADER     = "X-Vault-Token"
    DEFAULT_TOKEN_TTL = 1 * time.Hour

    // Storage constants
    SECRET_PREFIX    = "secret/"
    POLICY_PREFIX    = "policy/"
    TOKEN_PREFIX     = "token/"

)

```

### Milestone Checkpoints:

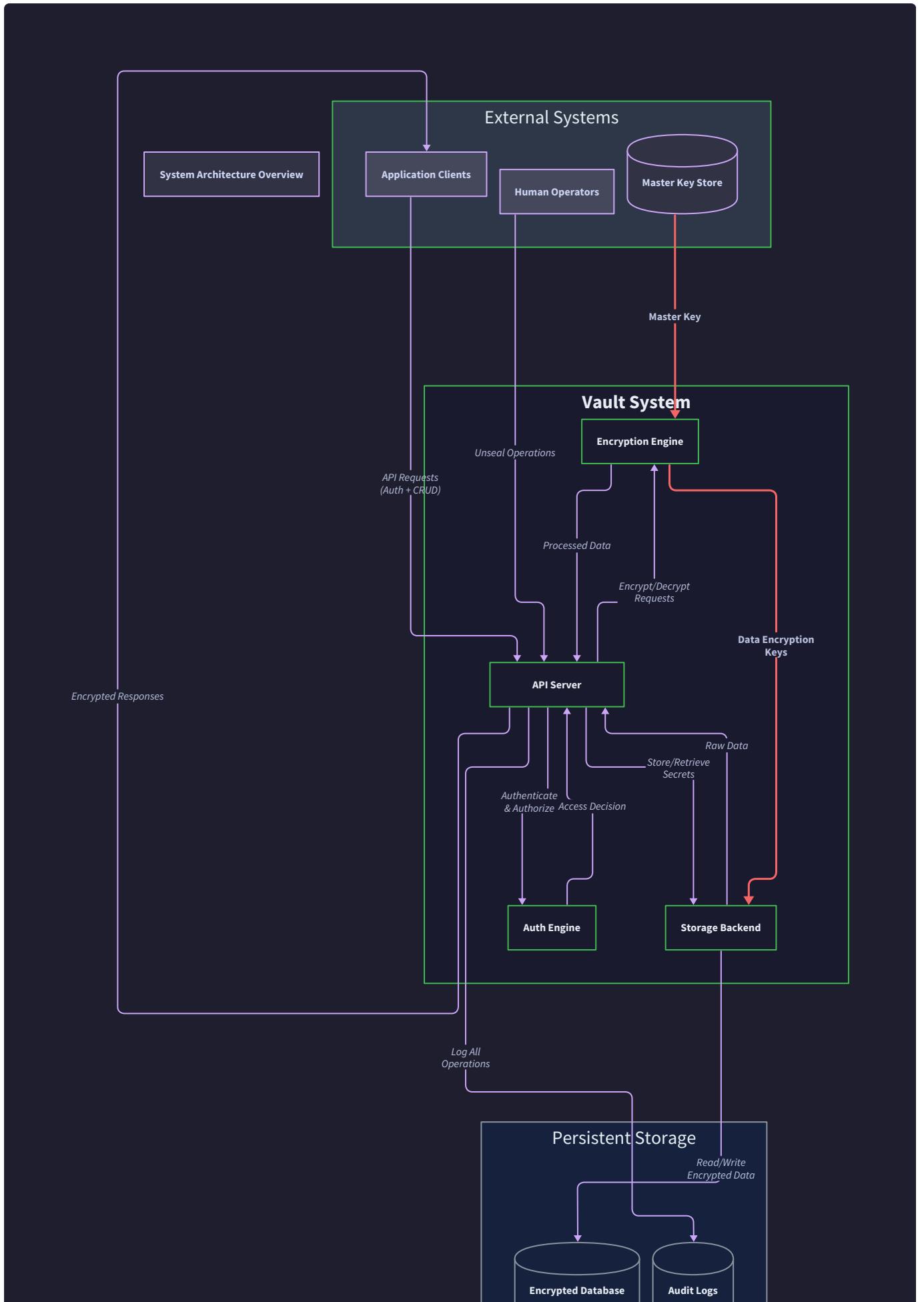
After implementing the basic server structure, verify these behaviors:

- 1. Configuration Loading:** Run `go run cmd/server/main.go -config configs/dev.yaml` - server should start and log configuration summary
- 2. TLS Endpoint:** Test with `curl -k https://localhost:8443/v1/sys/health` - should return sealed status
- 3. Component Integration:** Check logs for successful initialization of all four subsystems
- 4. Graceful Shutdown:** Send SIGTERM and verify clean shutdown with resource cleanup

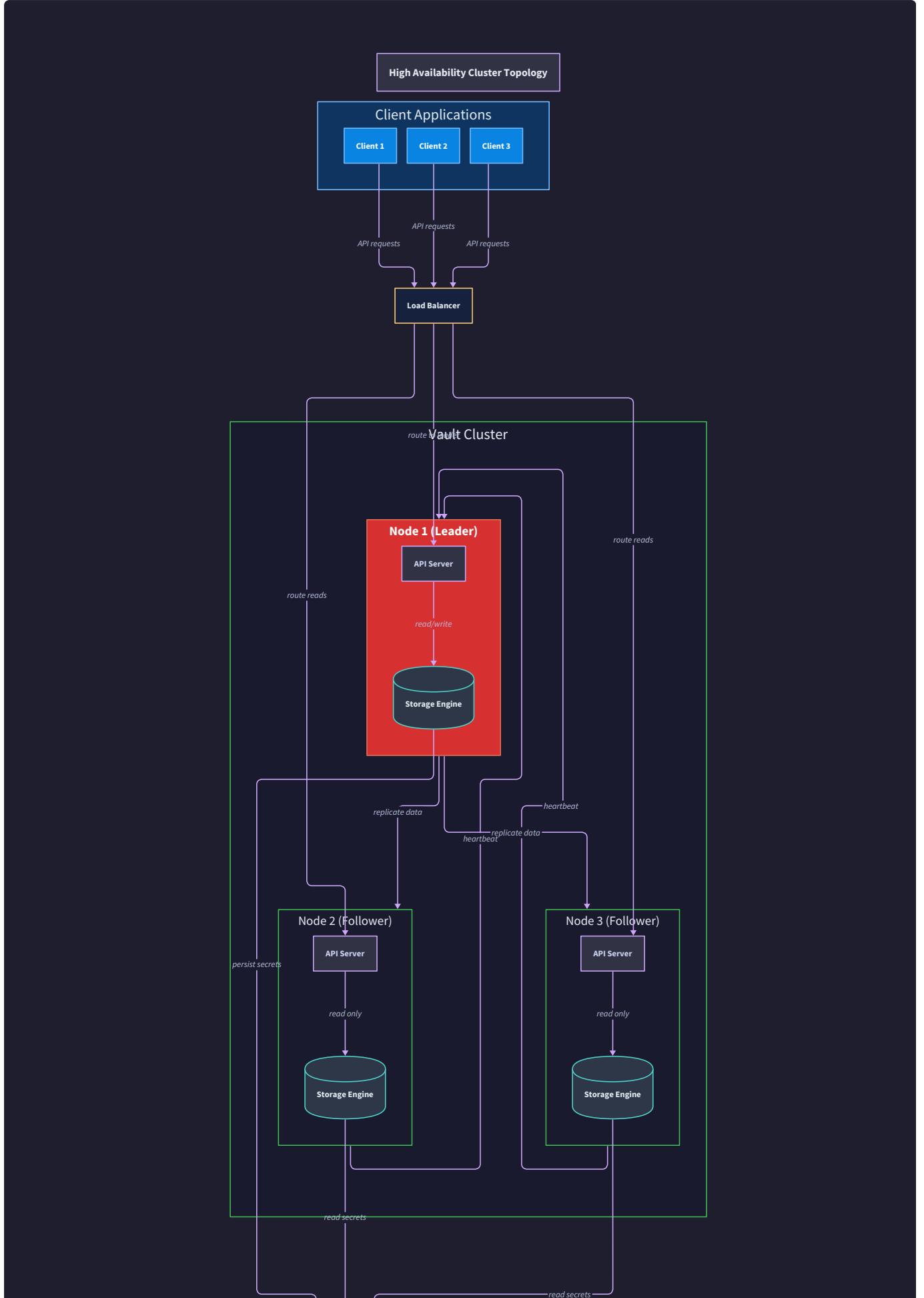
### Language-Specific Implementation Hints:

- Use `sync.RWMutex` for the sealed state - multiple readers can check seal status concurrently, but unsealing requires exclusive write access
- Implement `context.Context` support throughout for request timeouts and cancellation

- Use `crypto/rand.Reader` for all random number generation - never use `math/rand` for cryptographic operations
- Store sensitive data in `[]byte` slices and explicitly zero them with `for i := range data { data[i] = 0 }` after use
- Use `encoding/yaml` for configuration parsing with struct tags for validation
- Implement proper error wrapping with `fmt.Errorf("operation failed: %w", err)` for error context







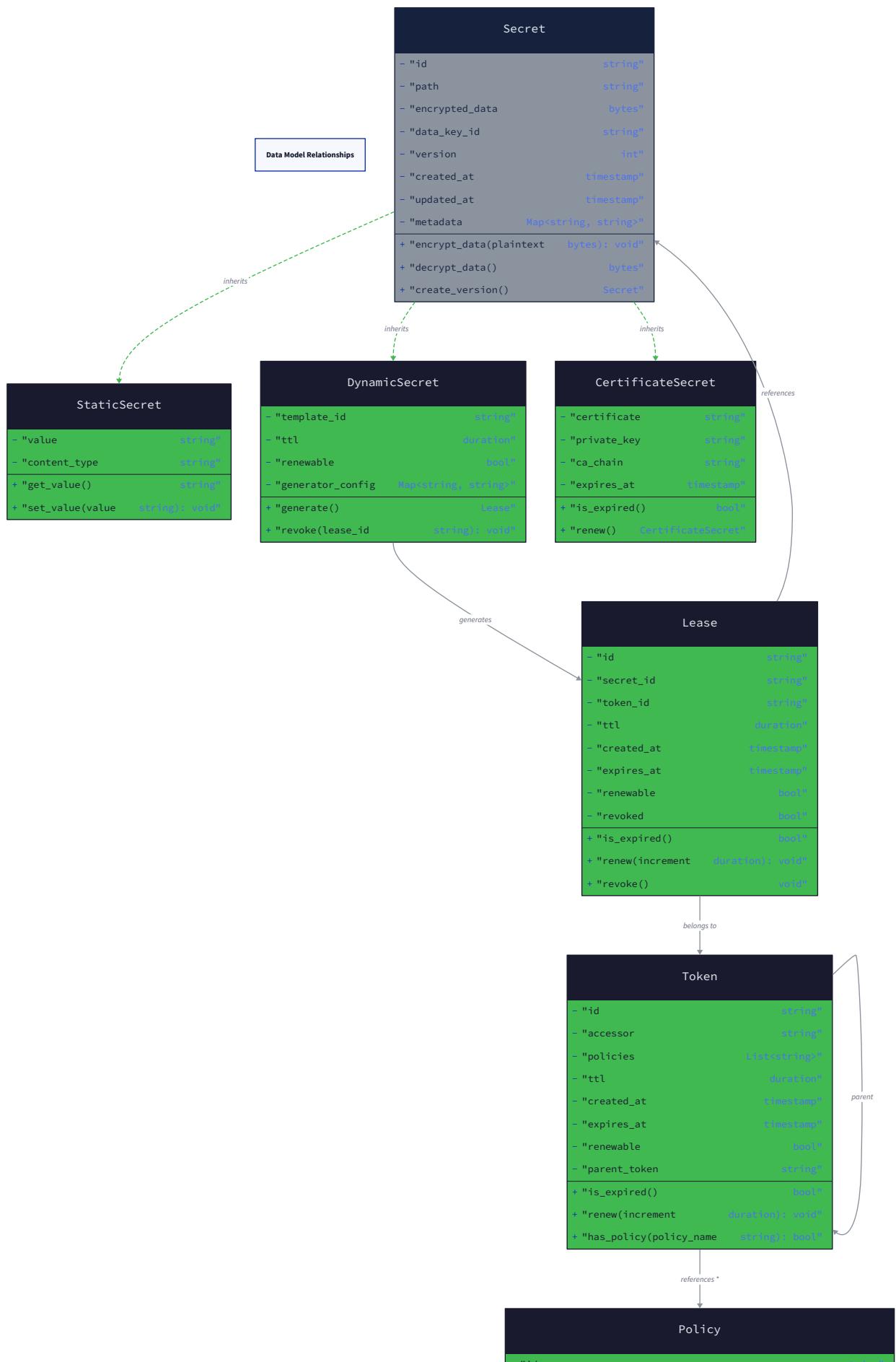


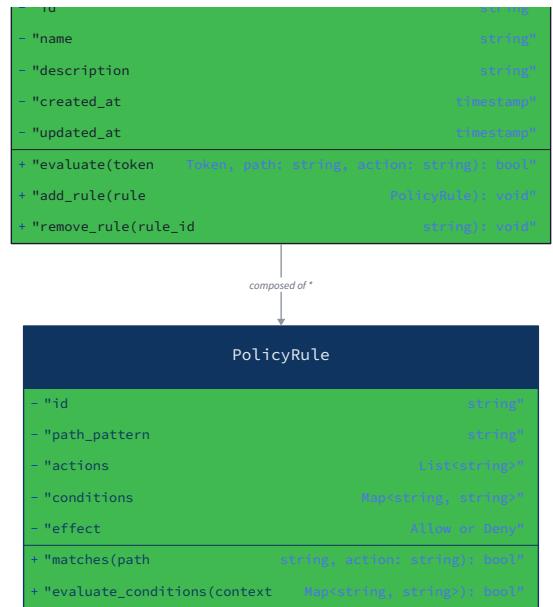
## Data Model

**Milestone(s):** This section establishes the core data structures that are implemented across all milestones, with specific focus on encrypted storage (Milestone 1), access control structures (Milestone 2), and dynamic secret management (Milestone 3).

The data model forms the foundational layer of our secret management system, defining how secrets, policies, tokens, and leases are structured, stored, and related to each other. Think of the data model as the **blueprint for a multi-vault bank** - it specifies not just how individual safety deposit boxes (secrets) are constructed, but also how access cards (tokens), security policies, and time-limited access passes (leases) work together to create a comprehensive security system.

Understanding this data model is crucial because every component of our system - from the encryption engine to the authentication system to the dynamic secret generators - operates on these core data structures. The model must balance security requirements (encrypted storage, access control) with operational needs (versioning, auditing, high availability) while maintaining the flexibility to support different types of secrets and backends.





## Secret Storage Model

The secret storage model defines how secrets are versioned, encrypted, and persisted in our system. This model must address the fundamental challenge of **envelope encryption** while supporting secret versioning for operational safety and key rotation scenarios.

### Mental Model: The Nested Safe System

Think of our secret storage like a **high-security bank with nested safes**. The bank has a master vault (master key) that protects individual safety deposit boxes (data encryption keys), and each safety deposit box contains multiple versions of a customer's valuables (secret versions). When a customer wants to access their valuables, the bank uses its master key to open the safety deposit box, retrieves the individual container key (data encryption key), and then uses that key to unlock the specific version of the valuables they need.

This nested approach means that even if someone gains access to the encrypted safety deposit box contents, they still can't read the valuables without the master vault key. Similarly, if the master key needs to be changed (key rotation), the bank can re-encrypt all the safety deposit box keys without touching each individual valuable.

### Secret Entity Structure

The `Secret` entity represents a logical secret path with all its versions and metadata. Each secret acts as a container for multiple encrypted versions, supporting both operational rollbacks and zero-downtime key rotation.

Field Name	Type	Description
Path	string	Unique secret identifier (e.g., "database/prod/password")
Versions	[]SecretVersion	Ordered list of secret versions, newest first
CurrentVersion	int	Version number of the active secret value
MaxVersions	int	Maximum versions to retain (older versions auto-pruned)
CreatedAt	time.Time	Timestamp when secret was first created
UpdatedAt	time.Time	Timestamp of most recent version addition
CreatedBy	string	Identity that created this secret path
DeletedAt	*time.Time	Soft deletion timestamp (nil if not deleted)
Metadata	map[string]string	User-defined key-value pairs for secret categorization
CASRequired	bool	Whether check-and-set is required for updates

The secret path follows a hierarchical naming convention similar to file systems, enabling policy engines to apply path-based access controls using wildcard patterns. The versioning system ensures that secret updates never overwrite existing values immediately, providing safety during deployment rollbacks and key rotation operations.

## Secret Version Structure

Each `SecretVersion` represents a specific encrypted value of a secret at a point in time, along with all metadata necessary for decryption and audit trails.

Field Name	Type	Description
Version	int	Monotonically increasing version number
EncryptedValue	[]byte	Secret value encrypted with data encryption key
KeyVersion	int	Version of data encryption key used for this secret
Algorithm	string	Encryption algorithm (typically "AES-256-GCM")
Nonce	[]byte	Cryptographic nonce used for authenticated encryption
AuthData	[]byte	Additional authenticated data (secret path + version)
CreatedAt	time.Time	Timestamp when this version was created
CreatedBy	string	Identity that created this version
TTL	time.Duration	Time-to-live for this version (zero means no expiration)
Checksum	[]byte	SHA-256 hash of plaintext for integrity verification

The `AuthData` field contains the secret path and version number, providing cryptographic binding between the encrypted value and its identity. This prevents attackers from copying encrypted values between different secret paths or versions. The checksum enables integrity verification after decryption without storing plaintext.

## Data Encryption Key Structure

The `DataEncryptionKey` (DEK) represents the keys used to encrypt individual secrets, which are themselves encrypted by the master key in our envelope encryption scheme.

Field Name	Type	Description
Version	int	Unique version number for this key
EncryptedKey	[]byte	The DEK encrypted with current master key
Algorithm	string	DEK algorithm (typically "AES-256")
MasterKeyVersion	int	Version of master key used to encrypt this DEK
CreatedAt	time.Time	Timestamp when this key version was generated
Status	string	Key lifecycle status (active, rotating, deprecated)
UsageCount	int64	Number of secrets encrypted with this key (for rotation triggers)

The key versioning system enables gradual key rotation without service disruption. During rotation, new secrets use the latest key version while existing secrets continue using their original keys until they are naturally updated or explicitly re-encrypted.

## Decision: Envelope Encryption with Versioned Data Keys

- **Context:** Secrets must be encrypted at rest, and we need to support key rotation without re-encrypting every secret simultaneously
- **Options Considered:**
  1. Single master key encrypts all secrets directly
  2. Envelope encryption with versioned data encryption keys
  3. Per-secret random keys encrypted with master key
- **Decision:** Envelope encryption with versioned data encryption keys
- **Rationale:** Enables gradual key rotation, limits master key exposure, and provides operational flexibility for key management. Per-secret keys would create too much key management overhead, while single master key rotation would require atomic re-encryption of all secrets.
- **Consequences:** Adds complexity with key versioning but enables zero-downtime key rotation and better security isolation.

Option	Pros	Cons	Chosen?
Direct master key encryption	Simple implementation, fewer keys to manage	Key rotation requires re-encrypting all secrets atomically	No
Envelope encryption with versioned DEKs	Gradual rotation, operational flexibility, limited master key exposure	More complex key management, additional storage overhead	Yes
Per-secret random keys	Maximum isolation, simple rotation per secret	Excessive key storage, complex key lifecycle management	No

## Access Control Model

The access control model defines how identities are authenticated, policies are structured, and authorization decisions are made. This model implements a **path-based access control** system that combines the flexibility of wildcard matching with the security of explicit policy definitions.

### Mental Model: The Corporate Badge System

Think of our access control like a **modern corporate badge system**. Employees carry access cards (tokens) that identify who they are and which security groups they belong to. The building has policy rules (policies) that specify which badge types can access which areas - for example, "Engineering badges can access floors 3-5" or "Facilities badges can access all floors but only during business hours." When someone tries to enter a secured area (access a secret), the card reader (authorization engine) checks their badge against the area's policy rules to make an access decision.

Just as corporate policies use patterns ("all Engineering areas" rather than listing every room), our system uses path patterns to efficiently specify access rules across hierarchical secret namespaces.

## Policy Structure

The `Policy` entity defines access permissions for secret paths, supporting wildcard patterns and fine-grained capability controls.

Field Name	Type	Description
<code>Name</code>	<code>string</code>	Unique policy identifier (e.g., "database-admin-policy")
<code>Rules</code>	<code>[]PolicyRule</code>	List of path-based access rules
<code>Description</code>	<code>string</code>	Human-readable policy purpose description
<code>CreatedAt</code>	<code>time.Time</code>	Policy creation timestamp
<code>UpdatedAt</code>	<code>time.Time</code>	Policy last modification timestamp
<code>CreatedBy</code>	<code>string</code>	Identity that created this policy
<code>Version</code>	<code>int</code>	Policy version for audit and rollback purposes

## Policy Rule Structure

Each `PolicyRule` specifies capabilities (permissions) for a specific path pattern, enabling fine-grained access control across the secret hierarchy.

Field Name	Type	Description
<code>Path</code>	<code>string</code>	Secret path pattern with wildcard support (e.g., "database/prod/*")
<code>Capabilities</code>	<code>[]string</code>	Allowed operations: "read", "write", "delete", "list"
<code>RequiredParameters</code>	<code>map[string] []string</code>	Required request parameters and allowed values
<code>AllowedParameters</code>	<code>map[string] []string</code>	Optional parameters and their allowed values
<code>DeniedParameters</code>	<code>map[string] []string</code>	Explicitly forbidden parameters
<code>MaxTTL</code>	<code>time.Duration</code>	Maximum TTL for secrets created/read under this path
<code>MinWrappingTTL</code>	<code>time.Duration</code>	Minimum TTL for response wrapping (if supported)

The path patterns support glob-style wildcards where `*` matches any characters within a path segment and `**` matches across multiple path segments. The capability system provides standard CRUD operations plus special operations like "list" for path enumeration.

## Token Structure

The `Token` entity represents authentication credentials issued to identities, carrying policy attachments and usage constraints.

Field Name	Type	Description
<code>ID</code>	<code>string</code>	Unique token identifier (cryptographically random)
<code>Accessor</code>	<code>string</code>	Non-sensitive token reference for audit logs
<code>Policies</code>	<code>[]string</code>	List of policy names attached to this token
<code>TokenType</code>	<code>string</code>	Token type: "service" (long-lived) or "batch" (lightweight)
<code>CreatedAt</code>	<code>time.Time</code>	Token issuance timestamp
<code>ExpiresAt</code>	<code>*time.Time</code>	Token expiration (nil for non-expiring service tokens)
<code>LastUsedAt</code>	<code>*time.Time</code>	Timestamp of most recent token usage
<code>UsageCount</code>	<code>int64</code>	Number of times token has been used
<code>MaxUses</code>	<code>int</code>	Maximum number of uses (0 = unlimited)
<code>Renewable</code>	<code>bool</code>	Whether token TTL can be extended
<code>ParentToken</code>	<code>string</code>	Token that created this token (for hierarchical revocation)
<code>DisplayName</code>	<code>string</code>	Human-readable token description
<code>Metadata</code>	<code>map[string]string</code>	Additional token metadata for audit purposes
<code>BoundCIDRs</code>	<code>[]string</code>	IP address ranges from which token can be used

The token hierarchy supports parent-child relationships, enabling revocation of all child tokens when a parent token is revoked. The accessor field provides a way to reference tokens in audit logs without exposing the actual token value.

## Identity Structure

The `Identity` entity represents authenticated principals (users, services, applications) that can be assigned policies and issued tokens.

Field Name	Type	Description
ID	string	Unique identity identifier
Name	string	Human-readable identity name
Type	string	Identity type: "user", "service", "role"
AuthMethod	string	Authentication method used: "token", "tls-cert", "approle"
Policies	[]string	Direct policy attachments for this identity
Groups	[]string	Group memberships (groups have their own policies)
CreatedAt	time.Time	Identity creation timestamp
LastAuthAt	*time.Time	Most recent successful authentication
AuthCount	int64	Total number of successful authentications
Metadata	map[string]string	Identity-specific metadata (department, team, etc.)
Disabled	bool	Whether identity is currently disabled

### Decision: Path-Based Access Control with Wildcard Patterns

- **Context:** Need flexible access control that scales across hierarchical secret namespaces without requiring individual rules for every secret path
- **Options Considered:**
  1. Per-secret explicit permissions (each secret lists authorized identities)
  2. Role-based access control with predefined roles
  3. Path-based policies with wildcard pattern matching
- **Decision:** Path-based policies with wildcard pattern matching
- **Rationale:** Provides operational scalability (one policy rule covers many secrets), aligns with hierarchical secret organization, and offers flexibility for dynamic secret paths. RBAC alone would be too rigid for diverse secret access patterns.
- **Consequences:** Requires careful policy design to avoid over-broad permissions, and wildcard matching adds complexity to authorization evaluation.

Option	Pros	Cons	Chosen?
Per-secret explicit permissions	Maximum security granularity	Doesn't scale, management overhead	No
Role-based access control	Simple to understand, common pattern	Too rigid for diverse access needs	No
Path-based wildcard policies	Scales well, flexible, intuitive	Risk of over-broad patterns	<b>Yes</b>

## Dynamic Secret Model

The dynamic secret model defines how time-limited credentials are generated, tracked, and revoked across different backend systems. This model addresses the challenge of **just-in-time credential provisioning** while maintaining strong lifecycle management and cleanup guarantees.

### Mental Model: The Credential Vending Machine

Think of dynamic secrets like a **sophisticated vending machine for credentials**. When you need database access, you insert your token (authentication), select the type of access you need (role specification), and the machine generates a fresh set of credentials just for you with an expiration time clearly printed on them (lease). The machine keeps a record of what it dispensed and when it expires (lease tracking), and it has a cleanup crew (revocation engine) that goes around collecting expired credentials and deactivating them in the backend systems.

Unlike static secrets that are like owning a key to a building, dynamic secrets are like getting a temporary visitor badge that automatically expires and gets deactivated - you get exactly the access you need for exactly the time you need it, and the system ensures cleanup happens automatically.

### Lease Structure

The `Lease` entity represents the lifecycle of a dynamically generated secret, tracking its validity period and enabling automatic revocation.

Field Name	Type	Description
ID	string	Unique lease identifier (cryptographically random)
SecretType	string	Type of secret: "database", "aws-iam", "ssh-key", etc.
BackendPath	string	Secret backend that generated this lease (e.g., "database/prod")
RoleName	string	Backend role used for credential generation
CreatedAt	time.Time	Lease creation timestamp
ExpiresAt	time.Time	Lease expiration timestamp (never nil for dynamic secrets)
RenewedAt	*time.Time	Timestamp of most recent renewal
RenewalCount	int	Number of times lease has been renewed
MaxTTL	time.Duration	Maximum total lifetime regardless of renewals
Renewable	bool	Whether this lease can have its TTL extended
TokenID	string	Token that requested this lease (for audit and revocation)
RevocationData	map[string]interface{}	Backend-specific data needed for credential cleanup
SecretData	map[string]interface{}	The actual generated credentials (encrypted at rest)
Status	string	Lease status: "active", "renewed", "expired", "revoked"

The `RevocationData` field contains backend-specific information needed to clean up credentials - for database backends, this might include the username and connection parameters; for cloud providers, it might include the user ARN and access key ID.

## Secret Backend Configuration

The `SecretBackend` entity defines how dynamic secret engines connect to and manage credentials in external systems.

Field Name	Type	Description
Path	string	Backend mount path (e.g., "database/prod", "aws/dev")
Type	string	Backend type: "database", "aws", "gcp", "ssh", "pki"
Description	string	Human-readable backend description
Config	map[string]interface{}	Backend-specific configuration (connection strings, etc.)
DefaultTTL	time.Duration	Default lease duration for secrets from this backend
MaxTTL	time.Duration	Maximum allowed lease duration
Roles	map[string]BackendRole	Named roles defining different credential types
CreatedAt	time.Time	Backend creation timestamp
UpdatedAt	time.Time	Backend last modification timestamp

The backend configuration is encrypted at rest since it contains sensitive connection information like database connection strings, cloud API keys, and service account credentials.

## Backend Role Configuration

The `BackendRole` entity defines templates for generating specific types of credentials within a secret backend.

Field Name	Type	Description
Name	string	Role name within the backend (e.g., "readonly", "admin")
CreationStatements	[]string	Commands to create credentials (SQL, API calls, etc.)
RevocationStatements	[]string	Commands to revoke/cleanup credentials
RenewStatements	[]string	Commands to extend credential lifetime (if supported)
DefaultTTL	time.Duration	Default lease duration for this role
MaxTTL	time.Duration	Maximum lease duration for this role
RenewIncrement	time.Duration	TTL increment when renewing leases
AllowedDomains	[]string	Allowed domains/suffixes for generated usernames
RoleOptions	map[string]interface{}	Backend-specific role configuration

For database backends, the `CreationStatements` might contain SQL commands like `CREATE USER '{{username}}'@'%' IDENTIFIED BY '{{password}}'` where the template variables are replaced with generated values. The revocation statements would contain corresponding cleanup commands.

## Lease Tracking and Revocation

The lease management system maintains several tracking structures to enable efficient renewal and revocation operations.

The `LeaseIndex` provides efficient lookups for lease operations:

Field Name	Type	Description
<code>ByExpiration</code>	<code>map[time.Time][]string</code>	Lease IDs grouped by expiration time buckets
<code>ByToken</code>	<code>map[string][]string</code>	Lease IDs associated with each token (for token revocation)
<code>ByBackend</code>	<code>map[string][]string</code>	Lease IDs associated with each backend (for backend cleanup)
<code>ActiveCount</code>	<code>int64</code>	Total number of active leases (for monitoring)

The `RevocationQueue` manages asynchronous cleanup of expired or revoked credentials:

Field Name	Type	Description
<code>LeaseID</code>	<code>string</code>	Lease identifier to revoke
<code>BackendPath</code>	<code>string</code>	Backend responsible for revocation
<code>RevocationData</code>	<code>map[string]interface{}</code>	Data needed to clean up credentials
<code>QueuedAt</code>	<code>time.Time</code>	When revocation was queued
<code>AttemptCount</code>	<code>int</code>	Number of revocation attempts made
<code>NextAttempt</code>	<code>time.Time</code>	When to retry revocation (for failed attempts)
<code>Priority</code>	<code>int</code>	Revocation priority (expired vs. explicitly revoked)

## Decision: Lease-Based Dynamic Secret Management

- **Context:** Dynamic secrets need automatic expiration and cleanup without manual intervention, while supporting renewal for long-running operations
- **Options Considered:**
  1. TTL-based secrets with background cleanup scanning
  2. Lease-based tracking with explicit revocation queues
  3. Event-driven cleanup with external system notifications
- **Decision:** Lease-based tracking with explicit revocation queues
- **Rationale:** Provides predictable cleanup guarantees, supports complex renewal patterns, and enables audit trails for credential lifecycle. Background scanning alone cannot handle revocation failures reliably.
- **Consequences:** Adds complexity with lease tracking but ensures credential cleanup even when backend systems are temporarily unavailable.

Option	Pros	Cons	Chosen?
TTL-based with background scanning	Simple implementation, low storage overhead	No cleanup guarantees, difficult failure handling	No
Lease-based with revocation queues	Reliable cleanup, audit trail, failure recovery	Higher storage overhead, more complex	Yes
Event-driven external notifications	Minimal storage, real-time cleanup	Depends on external systems, no retry logic	No

## Common Pitfalls

⚠ **Pitfall: Storing Revocation Data in Secret Data** Many implementations mistakenly store backend cleanup information alongside the generated credentials. This creates problems because the secret data is typically returned to clients, potentially exposing internal backend details like connection strings or service account names. The revocation data should be stored separately and never exposed in API responses.

⚠ **Pitfall: Not Encrypting Backend Configuration** Backend configurations contain highly sensitive information like database passwords and cloud API keys. Storing this configuration in plaintext makes it a high-value target for attackers. All backend configuration should be encrypted using the same envelope encryption system as regular secrets.

⚠ **Pitfall: Ignoring Lease Revocation Failures** When credential revocation fails (backend is down, network issues, etc.), many systems simply log the error and move on. This leaves orphaned credentials active in backend systems indefinitely. Implement retry logic with exponential backoff and alerting for repeatedly failed revocations.

**⚠ Pitfall: Unbounded Lease Renewal** Without proper max TTL enforcement, clients can renew leases indefinitely, defeating the purpose of dynamic secrets. Always enforce a maximum total lifetime regardless of renewal count, and consider implementing renewal limits or decay factors.

## Implementation Guidance

The data model implementation requires careful attention to encryption, serialization, and database schema design. The following guidance provides practical approaches for implementing these structures in Go.

## Technology Recommendations

Component	Simple Option	Advanced Option
Storage Backend	SQLite with GORM	PostgreSQL with custom queries
Serialization	JSON with struct tags	Protocol Buffers with schema evolution
Encryption Library	crypto/aes + crypto/cipher	<a href="https://github.com/hashicorp/go-kms-wrapping">github.com/hashicorp/go-kms-wrapping</a>
Key Derivation	crypto/pbkdf2	<a href="https://golang.org/x/crypto/argon2">golang.org/x/crypto/argon2</a>
Time Handling	time.Time with UTC	Custom time with nanosecond precision

## Recommended File Structure

```
internal/
  storage/
    models/
      secret.go      ← Secret and SecretVersion structs
      policy.go      ← Policy, PolicyRule, Token, Identity structs
      lease.go       ← Lease, SecretBackend, BackendRole structs
      encryption.go  ← DataEncryptionKey and crypto helpers
    backends/
      sqlite/
        migrations/   ← Database schema migration files
        sqlite.go     ← SQLite storage implementation
      postgres/
        postgres.go   ← PostgreSQL storage implementation (future)
      storage.go     ← Storage interface definition
    crypto/
      envelope.go   ← Envelope encryption implementation
      keys.go        ← Key generation and rotation
```

## Core Data Structure Definitions

```
// Package models provides the core data structures for secret management.          GO

package models

import (
    "time"
    "encoding/json"
)

// Secret represents a versioned secret with metadata and access control.

type Secret struct {

    Path        string      `json:"path" gorm:"primaryKey"`
    Versions    []SecretVersion `json:"versions" gorm:"foreignKey:SecretPath"`
    CurrentVersion int       `json:"current_version"`
    MaxVersions  int        `json:"max_versions" gorm:"default:10"`
    CreatedAt   time.Time   `json:"created_at"`
    UpdatedAt   time.Time   `json:"updated_at"`
    CreatedBy   string      `json:"created_by"`
    DeletedAt   *time.Time  `json:"deleted_at,omitempty"`
    Metadata    map[string]string `json:"metadata" gorm:"serializer:json"`
    CASRequired bool       `json:"cas_required"`

}

// SecretVersion represents a specific encrypted version of a secret value.

type SecretVersion struct {

    SecretPath  string      `json:"secret_path" gorm:"primaryKey"`
    Version     int         `json:"version" gorm:"primaryKey"`
    EncryptedValue []byte    `json:"-"` // Never serialize in JSON responses
}
```

```

    KeyVersion      int      `json:"key_version"`

    Algorithm       string   `json:"algorithm"`

    Nonce          []byte   `json:"-"` // Never serialize in JSON responses

    AuthData        []byte   `json:"-"` // Never serialize in JSON responses

    CreatedAt      time.Time `json:"created_at"`

    CreatedBy      string   `json:"created_by"`

    TTL            time.Duration `json:"ttl"`

    Checksum       []byte   `json:"-"` // Never serialize in JSON responses

}

// Policy represents access control rules for secret paths.

type Policy struct {

    Name      string   `json:"name" gorm:"primaryKey"`

    Rules     []PolicyRule `json:"rules" gorm:"foreignKey:PolicyName"`

    Description string   `json:"description"`

    CreatedAt  time.Time `json:"created_at"`

    UpdatedAt  time.Time `json:"updated_at"`

    CreatedBy  string   `json:"created_by"`

    Version    int      `json:"version"`

}

// PolicyRule defines capabilities for a specific path pattern.

type PolicyRule struct {

    ID          uint     `json:"-" gorm:"primaryKey"`

    PolicyName  string   `json:"-"`
    Path        string   `json:"path"`

    Capabilities []string `json:"capabilities" gorm:"serializer:json"`
}

```

```

    RequiredParameters map[string][]string `json:"required_parameters"
gorm:"serializer:json"`

    AllowedParameters map[string][]string `json:"allowed_parameters"
gorm:"serializer:json"`

    DeniedParameters map[string][]string `json:"denied_parameters"
gorm:"serializer:json"`

        MaxTTL           time.Duration     `json:"max_ttl"`
        MinWrappingTTL   time.Duration     `json:"min_wrapping_ttl"`

}

// Token represents an authentication credential with policy attachments.

type Token struct {

    ID      string      `json:"id" gorm:"primaryKey"`
    Accessor string      `json:"accessor" gorm:"uniqueIndex"`
    Policies []string    `json:"policies" gorm:"serializer:json"`
    TokenType string      `json:"token_type"`
    CreatedAt time.Time   `json:"created_at"`
    ExpiresAt *time.Time  `json:"expires_at,omitempty"`
    LastUsedAt *time.Time  `json:"last_used_at,omitempty"`
    UsageCount int64       `json:"usage_count"`
    MaxUses    int         `json:"max_uses"`
    Renewable  bool        `json:"renewable"`
    ParentToken string      `json:"parent_token"`
    DisplayName string      `json:"display_name"`
    Metadata   map[string]string `json:"metadata" gorm:"serializer:json"`
    BoundCIDRs []string    `json:"bound_cidrs" gorm:"serializer:json"`

}

```

## Storage Interface Implementation

```
// Package storage defines the interface for persisting secret management data.          GO
package storage

import (
    "context"
    "time"
    "your-project/internal/storage/models"
)

// Backend defines the interface for secret storage operations.

type Backend interface {
    // Secret operations

    CreateSecret(ctx context.Context, secret *models.Secret) error
    GetSecret(ctx context.Context, path string) (*models.Secret, error)
    GetSecretVersion(ctx context.Context, path string, version int) (*models.SecretVersion, error)
    UpdateSecret(ctx context.Context, path string, version *models.SecretVersion) error
    DeleteSecret(ctx context.Context, path string) error
    ListSecrets(ctx context.Context, pathPrefix string) ([]string, error)

    // Policy operations

    CreatePolicy(ctx context.Context, policy *models.Policy) error
    GetPolicy(ctx context.Context, name string) (*models.Policy, error)
    UpdatePolicy(ctx context.Context, policy *models.Policy) error
    DeletePolicy(ctx context.Context, name string) error
    ListPolicies(ctx context.Context) ([]string, error)
}
```

```
// Token operations

CreateToken(ctx context.Context, token *models.Token) error

GetToken(ctx context.Context, id string) (*models.Token, error)

UpdateToken(ctx context.Context, token *models.Token) error

DeleteToken(ctx context.Context, id string) error

DeleteTokensByParent(ctx context.Context, parentID string) error


// Lease operations

CreateLease(ctx context.Context, lease *models.Lease) error

GetLease(ctx context.Context, id string) (*models.Lease, error)

UpdateLease(ctx context.Context, lease *models.Lease) error

DeleteLease(ctx context.Context, id string) error

GetExpiredLeases(ctx context.Context, before time.Time) ([]*models.Lease, error)

GetLeasesByToken(ctx context.Context, tokenID string) ([]*models.Lease, error)


// Maintenance operations

Close() error

}

// TODO: Implement SQLite backend with proper transaction handling

// TODO: Add connection pooling and prepared statement caching

// TODO: Implement database migration system for schema updates

// TODO: Add metrics collection for storage operations

// TODO: Implement backup and restore functionality
```

## Envelope Encryption Helper

```
// Package crypto provides envelope encryption for secret values.          GO

package crypto

import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
    "crypto/sha256"
    "errors"
    "fmt"
)

const (
    AES_KEY_SIZE      = 32 // 256-bit keys for AES-256
    GCM_NONCE_SIZE   = 12 // 96-bit nonces for GCM
)

// DataEncryptionKey represents a key used to encrypt secret data.

type DataEncryptionKey struct {

    Version        int      `json:"version"`
    EncryptedKey   []byte   `json:"encrypted_key"`
    Algorithm      string   `json:"algorithm"`
    MasterKeyVersion int     `json:"master_key_version"`
    CreatedAt      time.Time `json:"created_at"`
    Status         string   `json:"status"` // active, rotating, deprecated
    UsageCount     int64    `json:"usage_count"`

}
```

```
// EnvelopeEncryption handles the encryption and decryption of secrets using envelope
// encryption.

type EnvelopeEncryption struct {

    masterKey []byte // Current master key (AES-256)

    keyVersion int    // Current master key version

}

// TODO: Initialize envelope encryption with master key loaded from secure storage

// TODO: Implement master key rotation with gradual DEK re-encryption

// TODO: Add key derivation from password using Argon2 (for development/testing)

// TODO: Integrate with cloud KMS for master key protection (auto-unseal)

// EncryptSecret encrypts a plaintext secret using envelope encryption.

// Returns encrypted data, nonce, and additional authenticated data.

func (e *EnvelopeEncryption) EncryptSecret(plaintext []byte, path string, version int,
dekVersion int) ([]byte, []byte, []byte, error) {

    // TODO 1: Generate or retrieve data encryption key for dekVersion

    // TODO 2: Generate random nonce for AES-GCM (12 bytes)

    // TODO 3: Create additional authenticated data from path + version

    // TODO 4: Create AES-GCM cipher with DEK

    // TODO 5: Encrypt plaintext with nonce and additional authenticated data

    // TODO 6: Return ciphertext, nonce, and auth data

    return nil, nil, nil, errors.New("not implemented")
}

// DecryptSecret decrypts an encrypted secret using envelope encryption.

func (e *EnvelopeEncryption) DecryptSecret(ciphertext, nonce, authData []byte, dekVersion
int) ([]byte, error) {

    // TODO 1: Retrieve and decrypt data encryption key for dekVersion

    // TODO 2: Create AES-GCM cipher with decrypted DEK
```

```
// TODO 3: Decrypt ciphertext using nonce and auth data for verification

// TODO 4: Verify checksum if provided (optional integrity check)

// TODO 5: Securely zero DEK from memory after use

return nil, errors.New("not implemented")

}

// GenerateDataEncryptionKey creates a new data encryption key encrypted with the master
key.

func (e *EnvelopeEncryption) GenerateDataEncryptionKey(version int) (*DataEncryptionKey,
error) {

    // TODO 1: Generate random 256-bit key for AES-256 using crypto/rand

    // TODO 2: Encrypt the DEK with current master key using AES-GCM

    // TODO 3: Create DataEncryptionKey struct with metadata

    // TODO 4: Securely zero the plaintext DEK from memory

    return nil, errors.New("not implemented")

}
```

## Database Migration Example

```
-- Migration 001: Initial secret management schema  
-- File: internal/storage/backends/sqlite/migrations/001_initial.sql  
  
-- Secrets table with versioning support  
  
CREATE TABLE secrets (  
  
    path TEXT PRIMARY KEY,  
  
    current_version INTEGER NOT NULL DEFAULT 1,  
  
    max_versions INTEGER NOT NULL DEFAULT 10,  
  
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  
    created_by TEXT NOT NULL,  
  
    deleted_at DATETIME,  
  
    metadata TEXT, -- JSON encoded map[string]string  
  
    cas_required BOOLEAN NOT NULL DEFAULT FALSE  
  
);  
  
-- Secret versions table storing encrypted values  
  
CREATE TABLE secret_versions (  
  
    secret_path TEXT NOT NULL,  
  
    version INTEGER NOT NULL,  
  
    encrypted_value BLOB NOT NULL,  
  
    key_version INTEGER NOT NULL,  
  
    algorithm TEXT NOT NULL DEFAULT 'AES-256-GCM',  
  
    nonce BLOB NOT NULL,  
  
    auth_data BLOB NOT NULL,  
  
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  
    created_by TEXT NOT NULL,
```

```
ttl INTEGER DEFAULT 0, -- TTL in nanoseconds, 0 = no expiration

checksum BLOB NOT NULL,

PRIMARY KEY (secret_path, version),

FOREIGN KEY (secret_path) REFERENCES secrets(path) ON DELETE CASCADE

);

-- Policies table for access control rules

CREATE TABLE policies (

    name TEXT PRIMARY KEY,

    description TEXT,

    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,

    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,

    created_by TEXT NOT NULL,

    version INTEGER NOT NULL DEFAULT 1

);

-- Policy rules table for path-based permissions

CREATE TABLE policy_rules (

    id INTEGER PRIMARY KEY AUTOINCREMENT,

    policy_name TEXT NOT NULL,

    path TEXT NOT NULL,

    capabilities TEXT NOT NULL, -- JSON encoded []string

    required_parameters TEXT, -- JSON encoded map[string][]string

    allowed_parameters TEXT, -- JSON encoded map[string][]string

    denied_parameters TEXT, -- JSON encoded map[string][]string

    max_ttl INTEGER DEFAULT 0, -- TTL in nanoseconds

    min_wrapping_ttl INTEGER DEFAULT 0,

    FOREIGN KEY (policy_name) REFERENCES policies(name) ON DELETE CASCADE
```

```
);

-- Indexes for efficient secret and policy lookups

CREATE INDEX idx_secrets_path_prefix ON secrets(path);

CREATE INDEX idx_secret_versions_created_at ON secret_versions(created_at);

CREATE INDEX idx_policy_rules_path ON policy_rules(path);

CREATE INDEX idx_policy_rules_policy ON policy_rules(policy_name);
```

## Milestone Checkpoints

### After implementing Secret Storage Model:

1. Run `go test ./internal/storage/...` - all storage interface tests should pass
2. Create a test secret: `POST /v1/secret/test` with JSON body `{"value": "test-secret"}`
3. Retrieve the secret: `GET /v1/secret/test` should return the encrypted value decrypted
4. Verify versioning: Update the same secret and confirm version increments
5. Check encryption: Examine database directly - secret values should be encrypted blob data

### After implementing Access Control Model:

1. Create a policy: `POST /v1/sys/policy/test-policy` with JSON policy rules
2. Create a token: `POST /v1/auth/token/create` with the policy attached
3. Test access: Use the token to access secrets matching the policy paths
4. Test denial: Verify requests outside policy paths return 403 Forbidden
5. Check audit logs: Confirm all access attempts are logged with token accessor

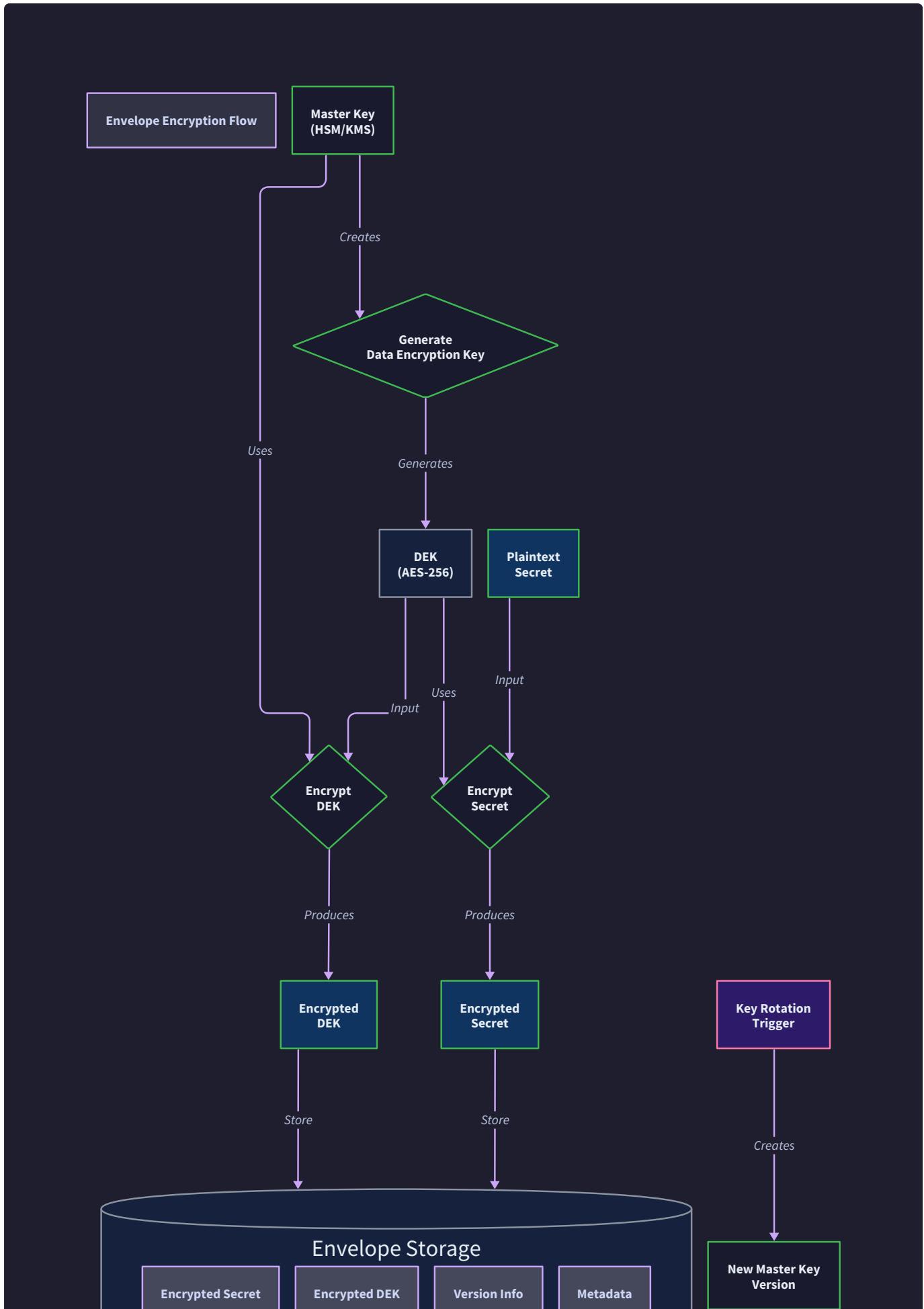
### After implementing Dynamic Secret Model:

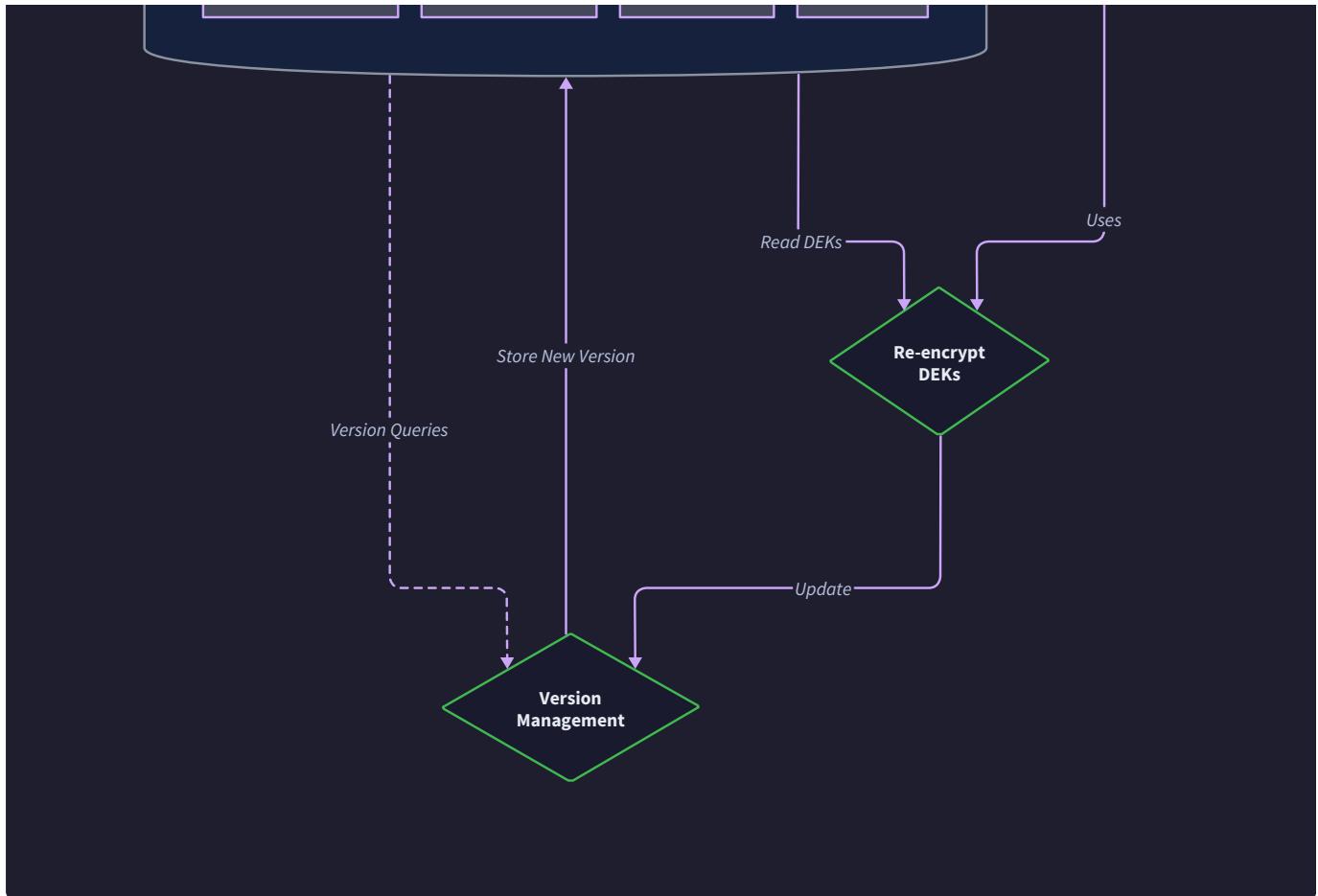
1. Configure a database backend: `POST /v1/database/config/test-db` with connection details
2. Create a role: `POST /v1/database/roles/readonly` with SQL creation statements
3. Request credentials: `GET /v1/database/creds/readonly` should return username/password
4. Verify TTL: Wait for lease expiration and confirm credentials are revoked
5. Test renewal: Renew lease before expiration and verify extended lifetime

## Encryption Engine Design

**Milestone(s):** This section implements Milestone 1 (Encrypted Secret Storage), focusing on envelope encryption with master keys and data encryption keys, secret versioning, and key rotation mechanisms.

The encryption engine forms the security heart of our secret management system, implementing a **nested safe system** that protects secrets through multiple layers of encryption. Like a bank that stores valuables in individual safety deposit boxes within a master vault, our system uses envelope encryption to create defense in depth against various attack vectors.





The encryption engine must solve several challenging problems simultaneously: protecting secrets at rest while maintaining performance, enabling key rotation without system downtime, supporting secret versioning for backward compatibility, and handling the cryptographic complexity of envelope encryption. Each of these requirements introduces technical constraints that shape our architectural decisions.

## Envelope Encryption Mental Model

**Envelope encryption** operates like a **nested safe system** where you have a master safe containing many smaller safes, each with their own keys. In a physical bank, the master vault key never leaves the secure control room, while individual deposit box keys are generated on demand and can be rotated regularly without touching the master vault.

In our digital implementation, the **master key** acts like the bank's master vault key - it never directly encrypts user secrets but instead protects a collection of **data encryption keys (DEKs)**. Each DEK is like an individual safety deposit box key that actually encrypts the secrets. When a client stores a secret, we generate a fresh DEK, use it to encrypt the secret, then encrypt the DEK itself with the master key before storing both pieces.

This separation provides several critical security properties. First, **key rotation isolation** - we can rotate DEKs regularly without touching the master key, and we can rotate the master key by re-encrypting all DEKs without re-encrypting every secret. Second, **blast radius containment** - if a single DEK is compromised, it only affects secrets encrypted with that specific key. Third, **performance optimization** - the expensive master key operations happen infrequently, while the bulk encryption work uses faster symmetric DEKs.

The envelope structure creates multiple failure points that an attacker must compromise. Even if they gain access to the encrypted secrets and encrypted DEKs in storage, they still need the master key to decrypt the DEKs. Even if they compromise the master key, they must still perform the multi-step decryption process for each secret individually.

**Critical Security Insight:** The master key should never directly touch user data. This principle ensures that master key compromise requires additional steps to access secrets, buying time for detection and response.

## Key Hierarchy and Rotation

Our encryption system implements a **three-tier key hierarchy** that balances security, performance, and operational flexibility. At the top sits the **master key** - a 256-bit AES key that serves as the root of trust for the entire system. The master key encrypts **data encryption keys (DEKs)**, which are also 256-bit AES keys generated for each encryption operation or time period. Finally, **secret-specific nonces** ensure that identical secrets produce different ciphertext even when encrypted with the same DEK.

Key Type	Size	Lifetime	Purpose	Rotation Frequency
Master Key	256-bit AES	System lifetime	Encrypts DEKs	Yearly or on compromise
Data Encryption Key	256-bit AES	Configurable (default 30 days)	Encrypts secrets	Monthly or on compromise
Nonce	96-bit random	Per-operation	Ensures ciphertext uniqueness	Every encryption

The **master key lifecycle** follows strict operational procedures. During system initialization, we generate the master key using a cryptographically secure random number generator and immediately split it using Shamir's secret sharing (covered in the unsealing section). The master key remains in memory only while the system is unsealed, and we clear it from memory during system shutdown or sealing operations.

**Data encryption key management** implements a more dynamic lifecycle. We maintain a **DEK rotation schedule** where new DEKs are generated automatically based on time intervals or usage counts. Each DEK has a version number and lifecycle status: `active` for new encryptions, `retired` for decryption-only, and `revoked` for compromised keys. The system maintains multiple DEK versions simultaneously to support decryption of older secrets while using only the latest active DEK for new encryptions.

## Decision: Time-Based DEK Rotation

- **Context:** We need to determine when to generate new data encryption keys - time-based, usage-based, or administrative triggers
- **Options Considered:**
  1. Fixed time intervals (daily/weekly/monthly)
  2. Usage count thresholds (encrypt N secrets then rotate)
  3. Hybrid approach with maximum time and usage limits
- **Decision:** Hybrid approach with 30-day maximum lifetime and 10,000 encryption maximum
- **Rationale:** Time limits prevent indefinite key usage, usage limits prevent single key from encrypting too much data, hybrid provides flexibility for both high and low volume environments
- **Consequences:** Requires tracking both time and usage metrics, provides defense against both long-term cryptanalysis and high-volume attacks

Rotation Trigger	Threshold	Action Taken	Backward Compatibility
Time Limit	30 days since creation	Generate new active DEK, mark current as retired	Retired DEKs remain available for decryption
Usage Limit	10,000 encryptions	Generate new active DEK, mark current as retired	All previous versions remain accessible
Compromise Detection	Administrative command	Immediately revoke DEK, generate replacement	Revoked DEK blocked, secrets must be re-encrypted
Master Key Rotation	Administrative command	Re-encrypt all DEKs with new master key	Transparent to secret access, versions preserved

**Key rotation algorithms** implement careful state management to prevent data loss during transitions. The rotation process follows these steps:

1. **Pre-rotation validation** - verify the current active DEK exists and system is in healthy state
2. **New DEK generation** - create new 256-bit key using secure random generator and assign next version number
3. **Master key encryption** - encrypt the new DEK with current master key and store encrypted version
4. **Atomic state transition** - update the active DEK pointer and mark previous DEK as retired in a single transaction
5. **Background cleanup** - optionally re-encrypt old secrets with new DEK based on policy configuration
6. **Audit logging** - record rotation event with timestamps, versions, and triggering conditions

The rotation process must handle **concurrent operations gracefully**. During rotation, in-flight encrypt operations complete with the DEK they started with, while new encrypt operations wait for rotation completion

before proceeding with the new DEK. Decrypt operations are unaffected since they specify the exact DEK version needed.

## Encrypt/Decrypt Operations

The **envelope encryption algorithm** implements a carefully orchestrated sequence that ensures both security and performance. Each encryption operation must produce ciphertext that can be decrypted later, track all necessary metadata for the decryption process, and maintain compatibility with secret versioning and key rotation.

### Secret Encryption Process:

1. **Input validation and preprocessing** - verify the secret path is valid, check secret length limits, and normalize the plaintext data for encryption
2. **DEK selection** - identify the current active data encryption key version and retrieve the decrypted DEK from secure memory cache
3. **Nonce generation** - create a 96-bit cryptographically random nonce using the system's secure random number generator
4. **Associated data preparation** - construct authenticated additional data including secret path, version number, timestamp, and DEK version
5. **AES-GCM encryption** - encrypt the plaintext secret using AES-256-GCM with the selected DEK, nonce, and associated data
6. **Metadata packaging** - bundle the ciphertext, nonce, associated data, DEK version, and algorithm identifier for storage
7. **Storage preparation** - serialize the complete encryption package and compute integrity checksums for corruption detection

The encryption operation produces a `SecretVersion` structure that contains all information needed for future decryption. The ciphertext itself is opaque, but the metadata provides versioning, integrity checking, and key identification without revealing the secret content.

Encryption Output Field	Source	Purpose	Security Properties
EncryptedValue	AES-GCM ciphertext	Encrypted secret data	Confidentiality via AES-256
Nonce	Secure random generator	GCM initialization vector	Prevents ciphertext reuse
AuthData	Path + version + timestamp	Authenticated context	Prevents ciphertext substitution
KeyVersion	Current active DEK	Identifies decryption key	Supports key rotation
Algorithm	Fixed "AES-256-GCM"	Crypto algorithm identifier	Enables future algorithm migration
Checksum	SHA-256 of complete package	Corruption detection	Integrity verification

#### Secret Decryption Process:

- Metadata extraction** - parse the stored `SecretVersion` to extract DEK version, nonce, associated data, and algorithm identifier
- DEK retrieval** - locate the specified DEK version, decrypt it with the master key if not cached, and verify the DEK is not revoked
- Algorithm verification** - confirm the encryption algorithm is supported and matches expected security parameters
- Associated data reconstruction** - rebuild the authenticated additional data using the same process as encryption
- AES-GCM decryption** - decrypt the ciphertext using AES-256-GCM with the retrieved DEK, stored nonce, and reconstructed associated data
- Integrity verification** - validate the authentication tag and verify the overall package checksum to detect corruption
- Plaintext return** - return the decrypted secret data and clear sensitive values from memory

The decryption process must handle **version compatibility** across key rotations and system upgrades. Each `SecretVersion` is self-contained with all metadata needed for decryption, ensuring that secrets encrypted months ago can still be decrypted even after multiple key rotations.

**Critical Implementation Detail:** The associated data must be reconstructed identically during decryption. Any variation in path normalization, timestamp formatting, or version numbering will cause authentication failures that prevent decryption.

**Performance optimization** strategies focus on the expensive cryptographic operations. The system maintains an **in-memory DEK cache** that keeps recently used decrypted DEKs available without requiring master key operations. The cache implements LRU eviction and automatic clearing when DEKs are rotated or revoked. Additionally, **batch operations** can reuse the same DEK for multiple secrets, reducing the per-secret encryption overhead.

**Error handling** during encryption and decryption operations requires careful consideration of information disclosure. Decryption failures should not reveal whether the failure was due to wrong DEK version, corrupted ciphertext, or authentication failures, as this information could aid attackers. Instead, all decryption failures return a generic "decryption failed" error while logging detailed failure reasons for administrative debugging.

Operation	Success Indicators	Failure Modes	Error Response
Encrypt	Valid ciphertext + metadata	DEK unavailable, RNG failure, storage error	Specific error for debugging
Decrypt	Valid plaintext	Wrong DEK, corrupt data, auth failure	Generic "decryption failed"
DEK Retrieve	DEK in memory or successfully decrypted	Master key unavailable, corrupted DEK	Internal error, retry logic
Nonce Generation	96 bits of entropy	RNG failure, insufficient entropy	System error, operation blocked

## Secret Versioning

**Secret versioning** implements a sophisticated system for maintaining historical secret values while supporting key rotation and data migration. Like a document version control system, each secret maintains multiple timestamped versions that can be retrieved independently while sharing common metadata and access policies.

The versioning system must solve several complex challenges. **Version immutability** ensures that once a secret version is created, its encrypted content cannot be changed - only new versions can be added. **Key rotation compatibility** allows old versions encrypted with rotated keys to remain accessible. **Storage efficiency** prevents unlimited version accumulation from consuming excessive storage. **Access control consistency** ensures that policy changes apply uniformly across all versions of a secret.

Each secret is represented by a `Secret` structure that acts as a container for multiple `SecretVersion` entries. The secret container maintains metadata that applies to all versions: creation timestamp, access policies, maximum version limits, and soft deletion status. Individual versions contain the encrypted data, encryption metadata, and version-specific attributes like TTL and creator identity.

Secret Container Field	Purpose	Version Scope	Mutability
Path	Unique secret identifier	All versions	Immutable after creation
CurrentVersion	Latest version number	Container-level	Incremented on updates
MaxVersions	Version retention limit	Container-level	Configurable policy
CreatedAt	Initial secret creation	Container-level	Immutable
UpdatedAt	Last version addition	Container-level	Updated on new versions
DeletedAt	Soft deletion timestamp	All versions	Set once, prevents new versions
Metadata	User-defined key-value pairs	All versions	Mutable via policy

The **version lifecycle** follows a strict progression from creation through active use to eventual cleanup. New versions receive sequential numbers starting from 1, ensuring a total ordering of all changes to the secret. The system maintains a **current version pointer** that identifies the latest version returned for unqualified read operations, while allowing explicit access to any historical version by number.

**Version creation process** implements atomic operations that prevent partial updates and ensure consistency:

1. **Concurrency control** - acquire write lock on the secret path to prevent concurrent version creation
2. **Validation checks** - verify the secret exists, is not soft-deleted, and has not reached the maximum version limit
3. **Version number assignment** - increment the current version counter and assign the new number atomically
4. **Encryption operation** - encrypt the new secret value using current active DEK and create `SecretVersion` structure
5. **Storage transaction** - store the new version and update container metadata in a single atomic operation
6. **Index updates** - update secondary indexes for version lookup and ensure audit log entries are created
7. **Cleanup trigger** - if versions exceed the maximum limit, schedule background cleanup of oldest versions

## Decision: Sequential Version Numbering vs UUID-Based Versions

- **Context:** Need to uniquely identify secret versions while supporting efficient lookup and ordering
- **Options Considered:**
  1. Sequential integers (1, 2, 3, ...)
  2. UUID-based version identifiers
  3. Timestamp-based version numbers
- **Decision:** Sequential integers with atomic increment
- **Rationale:** Provides natural ordering, enables efficient "latest N versions" queries, simplifies client usage patterns, and avoids clock synchronization issues in distributed deployments
- **Consequences:** Requires atomic counter management, reveals version creation frequency, but provides predictable and efficient access patterns

**Key rotation impact on versioning** requires careful coordination between the encryption engine and version management. When DEKs are rotated, existing secret versions remain encrypted with their original keys, creating a dependency between version lifetime and key lifetime. The system must ensure that DEK versions remain available for decryption as long as any secret version depends on them.

The system implements **lazy re-encryption** policies that allow administrators to migrate old secret versions to new keys without impacting system availability. Background processes can iterate through old versions, decrypt them with their original keys, and re-encrypt with current keys while preserving version numbers and metadata.

Re-encryption Policy	Trigger Conditions	Process	Version Impact
Immediate	DEK compromise detected	Re-encrypt all affected versions synchronously	Version numbers preserved, new encryption metadata
Background Lazy	DEK reaches retirement age	Queue versions for background re-encryption	Transparent to clients, gradual migration
On-Access	Version accessed after DEK retirement	Re-encrypt during read operation if policy allows	Single version updated on demand
Manual	Administrative command	Re-encrypt specified secret or version range	Controlled timing, batch operations

**Version cleanup and retention** policies prevent unlimited storage growth while preserving important historical data. The system supports multiple retention strategies that can be combined based on organizational requirements.

Default retention keeps the most recent 10 versions of each secret, automatically cleaning up older versions when new ones are created. **Time-based retention** deletes versions older than a specified age, while **usage-**

**based retention** preserves frequently accessed versions regardless of age. **Compliance retention** can mark specific versions as immutable for regulatory requirements.

The cleanup process implements **safe deletion** that ensures versions are not removed while they might be needed for decryption or audit purposes. Before deleting a version, the system verifies that no active leases reference it, no pending operations depend on it, and the retention policy explicitly allows its removal.

**⚠ Pitfall: Version Cleanup During Key Rotation** Deleting secret versions before their associated DEKs can create permanent data loss. If a secret version is deleted but its DEK is retained for other versions, there's no issue. However, if the DEK is deleted while secret versions still reference it, those versions become permanently undecryptable. The system must maintain reference counts from secret versions to DEK versions and prevent DEK deletion while references exist.

**Version access patterns** support both specific version retrieval and version-relative access. Clients can request version 5 of a secret explicitly, ask for the current version (default behavior), or use relative references like "previous version" or "version from 24 hours ago". The system translates these relative references into specific version numbers before processing the request.

## Common Pitfalls

**⚠ Pitfall: Master Key in Swap Space** Storing the master key in regular memory variables allows the operating system to page that memory to disk swap files, where it persists after the process exits. An attacker with disk access could extract the master key from swap space. Use memory locking syscalls (`mlock` on Unix systems) to prevent sensitive memory pages from being swapped to disk, and explicitly zero memory before freeing it.

**⚠ Pitfall: DEK Version Race Conditions** When rotating DEKs, concurrent encryption operations might start with one DEK version but complete with metadata pointing to a different version, creating decryption failures. Always capture the DEK version at the start of the encryption operation and use that same version throughout the entire process. Implement proper locking to ensure version transitions are atomic from the perspective of concurrent operations.

**⚠ Pitfall: Nonce Reuse Attacks** Reusing the same nonce with the same DEK completely breaks AES-GCM security, allowing attackers to recover plaintext or forge authenticated data. This can happen if the random number generator fails, if system clocks reset during nonce generation, or if DEK rotation doesn't properly coordinate with nonce generation. Always use a cryptographically secure random number generator for nonce creation, and consider including timestamp or counter components in nonce generation for additional protection.

**⚠ Pitfall: Associated Data Inconsistency** AES-GCM associated data must be identical between encryption and decryption, but subtle differences in path normalization, timestamp formatting, or metadata serialization can cause authentication failures. For example, storing paths as `/secret/app/db` during encryption but reconstructing as `secret/app/db` during decryption will fail authentication. Establish strict canonicalization rules for all associated data components and implement comprehensive test cases covering various input formats.

**⚠ Pitfall: Version Number Overflow** Using fixed-size integers for version numbers can cause overflow after many updates, potentially wrapping around to previously used version numbers and causing conflicts. While 64-bit integers provide enormous capacity (9 quintillion versions), consider the implications of long-lived secrets in high-update environments. Implement monitoring for version number consumption and establish procedures for secret migration before overflow occurs.

**⚠ Pitfall: Incomplete Cleanup on Errors** If encryption operations fail partway through (e.g., after generating the DEK but before storing the encrypted secret), sensitive data may remain in memory or temporary storage. Similarly, failed decryption operations might leave partially decrypted data accessible. Implement comprehensive cleanup using `defer` statements or similar mechanisms to ensure sensitive data is zeroed from memory regardless of operation outcome.

## Implementation Guidance

This implementation guidance provides the foundational encryption engine that handles envelope encryption, key management, and secret versioning. The core learning challenge is implementing the envelope encryption algorithm correctly while managing the complexity of key rotation and version compatibility.

### Technology Recommendations:

Component	Simple Option	Advanced Option
Encryption	Go crypto/aes + crypto/cipher (GCM)	Hardware security modules (HSM)
Key Storage	Encrypted files with proper permissions	Cloud KMS integration (AWS KMS, GCP KMS)
Random Generation	<code>crypto/rand.Reader</code>	Hardware entropy sources
Memory Security	Manual zeroing with explicit loops	Memory protection libraries
Key Derivation	<code>crypto/pbkdf2</code> for password-based keys	Argon2 with tuned parameters

### Recommended File Structure:

```
internal/encryption/
  engine.go          ← main encryption engine
  envelope.go        ← envelope encryption implementation
  keys.go            ← DEK management and rotation
  versioning.go      ← secret versioning logic
  engine_test.go     ← comprehensive test suite
  testdata/
    test_secrets.json ← test vectors for crypto operations
internal/storage/
  backend.go          ← storage interface
  memory.go           ← in-memory backend for development
cmd/vault/
  main.go             ← server entry point
```

### Infrastructure Starter Code (Complete and Ready to Use):

GO

```
// internal/encryption/crypto_utils.go

package encryption

import (
    "crypto/rand"
    "crypto/sha256"
    "crypto/subtle"
    "fmt"
    "io"
)

// SecureRandom provides cryptographically secure random bytes

type SecureRandom struct {
    source io.Reader
}

func NewSecureRandom() *SecureRandom {
    return &SecureRandom{source: rand.Reader}
}

func (sr *SecureRandom) GenerateKey(size int) ([]byte, error) {
    key := make([]byte, size)
    if _, err := io.ReadFull(sr.source, key); err != nil {
        return nil, fmt.Errorf("failed to generate random key: %w", err)
    }
    return key, nil
}

func (sr *SecureRandom) GenerateNonce(size int) ([]byte, error) {
```

```
nonce := make([]byte, size)

if _, err := io.ReadFull(sr.source, nonce); err != nil {

    return nil, fmt.Errorf("failed to generate random nonce: %w", err)
}

return nonce, nil
}

// SecureMemory provides utilities for secure memory handling

type SecureMemory struct{}


func (sm *SecureMemory) Zero(data []byte) {

    for i := range data {

        data[i] = 0
    }
}

func (sm *SecureMemory) Compare(a, b []byte) bool {

    return subtle.ConstantTimeCompare(a, b) == 1
}

func (sm *SecureMemory) ComputeChecksum(data []byte) []byte {

    hash := sha256.Sum256(data)

    return hash[:]
}

// KeyDerivation provides password-based key derivation

type KeyDerivation struct {

    iterations int

    saltSize   int
}
```

```
}

func NewKeyDerivation() *KeyDerivation {
    return &KeyDerivation{
        iterations: 480000, // OWASP recommended minimum
        saltSize:   32,     // 256 bits
    }
}

func (kd *KeyDerivation) DeriveKey(password []byte, salt []byte, keySize int) []byte {
    // Implementation would use golang.org/x/crypto/pbkdf2
    // This is a placeholder - implement PBKDF2 derivation
    return make([]byte, keySize) // TODO: Implement PBKDF2
}
```

#### Core Logic Skeleton (Students Implement These):

GO

```
// internal/encryption/engine.go

package encryption

import (
    "context"
    "crypto/aes"
    "crypto/cipher"
    "encoding/json"
    "fmt"
    "sync"
    "time"
)

// Engine manages envelope encryption for the secret management system

type Engine struct {

    masterKey      []byte
    keyVersion     int
    dekCache       map[int]*DataEncryptionKey
    currentDEK     *DataEncryptionKey
    mutex          sync.RWMutex
    random         *SecureRandom
    memory         *SecureMemory
    rotationPolicy RotationPolicy
}

type RotationPolicy struct {

    MaxAge        time.Duration
    MaxUsageCount int64
}
```

```
    AutoRotate    bool  
}  
  
// NewEngine creates a new encryption engine with the specified master key  
  
func NewEngine(masterKey []byte, policy RotationPolicy) (*Engine, error) {  
  
    if len(masterKey) != AES_KEY_SIZE {  
  
        return nil, fmt.Errorf("master key must be %d bytes", AES_KEY_SIZE)  
    }  
  
    // TODO 1: Initialize the engine with copied master key (don't store reference)  
  
    // TODO 2: Create secure random and memory utilities  
  
    // TODO 3: Initialize DEK cache and generate initial DEK  
  
    // TODO 4: Set up rotation policy and start background rotation if enabled  
  
    // Hint: Use copy() to safely copy the master key into engine memory  
  
    return nil, fmt.Errorf("not implemented")  
}  
  
// EncryptSecret encrypts a secret using envelope encryption  
  
func (e *Engine) EncryptSecret(plaintext []byte, secretPath string, version int)  
(*SecretVersion, error) {  
  
    // TODO 1: Validate input parameters (non-nil plaintext, valid path)  
  
    // TODO 2: Acquire read lock and get current active DEK  
  
    // TODO 3: Generate cryptographically secure nonce using e.random  
  
    // TODO 4: Construct associated data (path + version + timestamp + DEK version)  
  
    // TODO 5: Create AES-GCM cipher using DEK and encrypt with associated data  
  
    // TODO 6: Increment DEK usage count and check if rotation is needed  
  
    // TODO 7: Create SecretVersion with all encryption metadata and checksum
```

```
// Hint: Use json.Marshal for consistent associated data serialization


return nil, fmt.Errorf("not implemented")

}

// DecryptSecret decrypts a secret using envelope encryption

func (e *Engine) DecryptSecret(secretVersion *SecretVersion) ([]byte, error) {

    // TODO 1: Validate secret version has required fields (EncryptedValue, KeyVersion,
Nonce)

    // TODO 2: Retrieve the DEK for the specified version (check cache first)

    // TODO 3: Verify the encryption algorithm matches what we support

    // TODO 4: Reconstruct associated data using same process as encryption

    // TODO 5: Create AES-GCM cipher and decrypt with associated data verification

    // TODO 6: Verify overall package checksum to detect corruption

    // TODO 7: Return plaintext and zero sensitive intermediate values

    // Hint: Authentication failures should return generic errors, not specific causes


return nil, fmt.Errorf("not implemented")

}

// GenerateDataEncryptionKey creates a new DEK encrypted with the master key

func (e *Engine) GenerateDataEncryptionKey(version int) (*DataEncryptionKey, error) {

    // TODO 1: Generate new 256-bit random key using secure random generator

    // TODO 2: Create AES cipher using master key for DEK encryption

    // TODO 3: Generate nonce for master key encryption of DEK

    // TODO 4: Encrypt the new DEK using AES-GCM with master key

    // TODO 5: Create DataEncryptionKey struct with encrypted DEK and metadata

    // TODO 6: Set status to "active" and initialize usage counter
```

```
// TODO 7: Zero the plaintext DEK from memory before returning

// Hint: Store creation timestamp for rotation policy decisions


return nil, fmt.Errorf("not implemented")

}

// RotateDataEncryptionKey creates new DEK and marks current as retired

func (e *Engine) RotateDataEncryptionKey() error {

    // TODO 1: Acquire write lock to prevent concurrent rotation

    // TODO 2: Generate new DEK with incremented version number

    // TODO 3: Add new DEK to cache and update current DEK pointer

    // TODO 4: Mark previous DEK as "retired" but keep in cache

    // TODO 5: Log rotation event with timestamps and version numbers

    // TODO 6: Optionally trigger background re-encryption of old secrets

    // TODO 7: Update rotation metrics and schedule next rotation

    // Hint: Use atomic operations for DEK pointer updates


return fmt.Errorf("not implemented")

}

// GetDEKByVersion retrieves and decrypts DEK for specified version

func (e *Engine) GetDEKByVersion(version int) ([]byte, error) {

    // TODO 1: Check DEK cache for requested version

    // TODO 2: If not cached, verify DEK exists and is not revoked

    // TODO 3: Create AES-GCM cipher using master key

    // TODO 4: Decrypt the stored encrypted DEK

    // TODO 5: Add decrypted DEK to cache with LRU eviction

    // TODO 6: Return decrypted DEK (caller must zero when done)
```

```

    // Hint: Implement cache eviction when cache size exceeds limit

    return nil, fmt.Errorf("not implemented")
}

// Shutdown securely cleans up the encryption engine

func (e *Engine) Shutdown() error {
    // TODO 1: Acquire write lock to prevent new operations

    // TODO 2: Zero master key from memory

    // TODO 3: Zero all cached DEKs from memory

    // TODO 4: Clear DEK cache and reset pointers

    // TODO 5: Stop any background rotation processes

    // TODO 6: Log shutdown event for audit trail

    // Hint: Use e.memory.Zero() to securely clear byte slices

    return nil
}

```

### Language-Specific Hints:

- Use `crypto/aes` and `crypto/cipher` packages for AES-GCM implementation
- Always check `cipher.NewGCM()` errors - some AES implementations don't support GCM
- Use `defer` statements to ensure sensitive memory is zeroed even on early returns
- The `sync.RWMutex` allows multiple concurrent decryptions but exclusive encryption
- Use `encoding/json.Marshal` for consistent associated data serialization
- Check random number generation errors - they indicate serious system problems
- Use `copy()` to safely duplicate byte slices containing sensitive data

### Milestone Checkpoint:

After implementing the encryption engine, verify the following behavior:

#### 1. Basic Encryption Test:

```
go test ./internal/encryption/ -run TestBasicEncryption -v
```

BASH

Expected: Encrypt and decrypt a simple secret successfully

## 2. Key Rotation Test:

```
go test ./internal/encryption/ -run TestKeyRotation -v
```

BASH

Expected: Rotate DEK and decrypt old secrets with retired key

## 3. Version Management Test:

```
go test ./internal/encryption/ -run TestVersioning -v
```

BASH

Expected: Create multiple secret versions and retrieve specific versions

## 4. Manual Testing:

```
// Create test program in cmd/test-encryption/  
  
engine, _ := NewEngine(masterKey, policy)  
  
secretVersion, _ := engine.EncryptSecret([]byte("test-secret"), "/app/db", 1)  
  
plaintext, _ := engine.DecryptSecret(secretVersion)  
  
// plaintext should equal []byte("test-secret")
```

GO

## Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
"cipher: message authentication failed"	Associated data mismatch between encrypt/decrypt	Log associated data on both operations	Ensure identical path/version/timestamp formatting
"DEK not found for version X"	Key rotation deleted DEK while secrets still reference it	Check DEK cache contents and version references	Implement reference counting for DEK cleanup
Memory corruption or segfaults	Not zeroing sensitive data after use	Run with race detector: <code>go test -race</code>	Add defer statements to zero byte slices
Slow encryption performance	Regenerating DEKs on every operation	Profile with <code>go test -cpuprofile=cpu.prof</code>	Implement DEK caching and reuse
Nonce collision errors	Weak random number generator or clock issues	Test RNG quality with statistical tests	Use crypto/rand.Reader and verify system entropy

## Authentication and Policy Engine

**Milestone(s):** This section implements Milestone 2 (Access Policies & Authentication), focusing on path-based ACLs, multiple authentication methods, policy evaluation, and comprehensive audit logging.

Think of authentication and authorization like a corporate badge system combined with a sophisticated building access control system. When you arrive at work, you first prove your identity by swiping your badge (authentication), then the system checks which floors, rooms, and resources your role allows you to access (authorization). The security desk maintains logs of every badge swipe and access attempt (audit logging). Our secret management system works similarly — clients must first prove their identity through tokens or certificates, then the system evaluates whether their assigned policies grant access to the requested secret path.

The authentication and policy engine serves as the **gatekeeper** for our secret management system, implementing a zero-trust security model where every request must be authenticated and authorized before accessing any secrets. This engine consists of four interconnected subsystems: authentication methods that verify client identity, a policy definition language that specifies access rules, an authorization engine that evaluates requests against policies, and comprehensive audit logging that tracks every access attempt for security monitoring.

## Authentication Methods

The authentication subsystem implements multiple methods for clients to prove their identity, similar to how a high-security facility might accept both employee badges and visitor certificates. Each authentication method produces a verified identity that the authorization engine can evaluate against policies.

**Token-Based Authentication** serves as our primary authentication method, functioning like a corporate access card that employees carry. When a client successfully authenticates through any method, the system issues a **bearer token** — a cryptographically signed identifier that proves the client's authenticated identity without requiring them to re-authenticate for subsequent requests.

The token structure contains essential identity and authorization information:

Field	Type	Description
ID	string	Unique token identifier for lookups and revocation
Accessor	string	Public handle for token management without exposing ID
Policies	[]string	List of policy names attached to this token
TokenType	string	Either "service" for long-lived or "batch" for short-lived
CreatedAt	time.Time	Token creation timestamp for audit trails
ExpiresAt	*time.Time	Optional expiration time (nil for non-expiring tokens)
LastUsedAt	*time.Time	Most recent usage timestamp for activity monitoring
UsageCount	int64	Number of times token has been used
MaxUses	int	Maximum allowed uses before token expires (0 for unlimited)
Renewable	bool	Whether token can have its TTL extended
ParentToken	string	Token that created this token (for hierarchical revocation)
DisplayName	string	Human-readable identifier for audit logs
Metadata	map[string]string	Additional context about token creation
BoundCIDRs	[]string	IP address restrictions for token usage

## Decision: Bearer Token Authentication

- **Context:** Clients need to authenticate repeatedly for secret access, but re-authentication for every request creates performance bottlenecks and user experience problems
- **Options Considered:** HTTP Basic Auth, JWT tokens, custom bearer tokens
- **Decision:** Custom bearer tokens with server-side state
- **Rationale:** Unlike stateless JWTs, server-side tokens allow immediate revocation and detailed usage tracking. Basic Auth would require credential transmission on every request
- **Consequences:** Enables immediate token revocation and comprehensive audit trails, but requires server-side token storage and lookup overhead

**Mutual TLS (mTLS) Authentication** provides our strongest authentication method, similar to how diplomatic facilities require both photo identification and biometric verification. In mTLS, both the client and server present X.509 certificates to each other, creating a cryptographically verified bidirectional trust relationship.

The mTLS authentication flow operates through these steps:

1. Client initiates TLS connection to the secret management server
2. Server presents its certificate to prove server identity
3. Server requests client certificate as part of TLS handshake
4. Client presents its certificate signed by a trusted Certificate Authority
5. Server validates client certificate chain, expiration, and revocation status
6. Server extracts identity information from certificate Common Name or Subject Alternative Names
7. Server issues a bearer token bound to the certificate identity for subsequent API calls

The identity structure captures authenticated client information regardless of authentication method:

Field	Type	Description
ID	string	Unique identity identifier across all auth methods
Name	string	Human-readable name (certificate CN, username, etc.)
Type	string	Identity type: "certificate", "user", "service", etc.
AuthMethod	string	Which authentication method verified this identity
Policies	[]string	Default policies assigned to this identity
Groups	[]string	Group memberships for policy inheritance
CreatedAt	time.Time	When identity was first registered
LastAuthAt	*time.Time	Most recent authentication timestamp
AuthCount	int64	Total number of authentication attempts
Metadata	map[string]string	Auth-method-specific context (certificate fingerprint, etc.)
Disabled	bool	Whether identity is administratively disabled

The critical security insight here is that authentication only establishes **who** the client is, not **what** they can access. The authentication engine's job ends once it produces a verified identity — the authorization engine handles all access control decisions based on that identity.

**AppRole Authentication** provides a specialized method designed for applications and services rather than human users. Think of AppRole like a combination of an application ID and a temporary access code — the application proves it knows both pieces of information to authenticate.

AppRole authentication uses two credentials:

- **Role ID:** A public identifier that specifies which application role the client claims
- **Secret ID:** A private, time-limited credential that proves the client is authorized to assume that role

This two-factor approach prevents common application security mistakes where developers embed long-lived credentials directly in application code. The Role ID can be embedded in configuration files or environment variables, while the Secret ID must be delivered through a separate, secure channel (such as a deployment system or init container).

**⚠ Pitfall: Secret ID Distribution** Many developers try to solve the Secret ID distribution problem by embedding both Role ID and Secret ID in the same configuration file or environment variables. This defeats the security purpose of AppRole authentication. The Secret ID must be delivered through a separate mechanism — ideally one that provides it just-in-time during application startup and doesn't persist it to disk.

## Policy Definition Language

The policy definition language provides a declarative way to specify **what** authenticated identities can access, using a path-based access control model similar to filesystem permissions but designed specifically for hierarchical secret storage.

**Path-Based Access Control** treats secrets like files in a directory tree, where policies specify access rules for path patterns. Think of this like building security where your badge might grant access to "all rooms on floor 3" or "any conference room" rather than having to list every specific room individually.

A policy consists of one or more rules, where each rule specifies:

Field	Type	Description
Name	string	Unique policy identifier for assignment to tokens/identities
Rules	[]PolicyRule	Ordered list of access rules evaluated sequentially
Description	string	Human-readable policy purpose documentation
CreatedAt	time.Time	Policy creation timestamp for audit trails
UpdatedAt	time.Time	Most recent policy modification timestamp
CreatedBy	string	Identity that created this policy
Version	int	Policy version number for change tracking

Each policy rule defines fine-grained access controls for specific path patterns:

Field	Type	Description
Path	string	Secret path pattern with wildcards (* for segment, ** for recursive)
Capabilities	[]string	Allowed operations: "read", "create", "update", "delete", "list"
RequiredParameters	map[string] []string	Request parameters that MUST be present with allowed values
AllowedParameters	map[string] []string	Request parameters that MAY be present with allowed values
DeniedParameters	map[string] []string	Request parameters that explicitly DENY access
MaxTTL	time.Duration	Maximum TTL for dynamic secrets generated under this path
MinWrappingTTL	time.Duration	Minimum TTL for secret wrapping responses

**Wildcard Pattern Matching** enables flexible path-based access control without requiring explicit rules for every possible secret path. The system supports two wildcard types:

- **Single Segment Wildcard ( \* )**: Matches exactly one path segment
- **Recursive Wildcard ( \*\* )**: Matches zero or more path segments at any depth

Examples of wildcard pattern matching:

- `secret/team-alpha/*` matches `secret/team-alpha/database` but NOT `secret/team-alpha/env/prod`
- `secret/team-alpha/**` matches both `secret/team-alpha/database` AND `secret/team-alpha/env/prod`
- `secret/*/database` matches `secret/team-alpha/database` and `secret/team-beta/database`
- `secret/**/credentials` matches `secret/app1/credentials`, `secret/team/app1/credentials`, etc.

## Decision: Path-Based Access Control with Wildcards

- **Context:** Applications need flexible access control that doesn't require updating policies every time new secrets are added, while maintaining security through least-privilege principles
- **Options Considered:** Explicit path lists, regex patterns, glob-style wildcards
- **Decision:** Glob-style wildcards with single (\*) and recursive (\*\*) patterns
- **Rationale:** Glob patterns provide intuitive syntax familiar to developers while being more predictable than regex. Explicit lists don't scale for dynamic environments
- **Consequences:** Enables scalable access control policies but requires careful wildcard design to prevent overly broad access grants

**Capability-Based Operations** define what actions an identity can perform on matched secret paths. The system implements five core capabilities:

Capability	Description	Required For
read	Retrieve secret values and metadata	GET /v1/secret/data/path
create	Create new secrets at non-existing paths	POST /v1/secret/data/path (new paths only)
update	Modify existing secrets or create new versions	POST /v1/secret/data/path (existing paths)
delete	Soft-delete secrets and versions	DELETE /v1/secret/data/path
list	Enumerate secret paths under a prefix	LIST /v1/secret/metadata/path

**⚠ Pitfall: Overly Broad Wildcard Patterns** A common mistake is using patterns like `secret/**` with `["read", "create", "update", "delete"]` capabilities, which grants unlimited access to all secrets. This defeats the purpose of access control. Instead, design policies with narrow wildcard patterns that match the minimum necessary paths for each application's requirements.

**Parameter-Based Constraints** provide fine-grained control over request parameters, enabling policies to restrict not just what paths can be accessed, but how they can be accessed. This is particularly important for dynamic secrets where policy might allow database access but restrict which database roles can be requested.

Example policy rule with parameter constraints:

```
Path: secret/database/**  
Capabilities: ["read"]  
RequiredParameters:  
  environment: ["staging", "production"]  
AllowedParameters:  
  role: ["readonly", "readwrite"]  
  ttl: ["1h", "8h", "24h"]  
DeniedParameters:  
  role: ["admin", "superuser"]
```

This rule ensures that database secret requests must specify a valid environment, can optionally specify allowed roles and TTL values, but explicitly denies access to administrative database roles.

## Authorization Flow

The authorization engine evaluates every authenticated request against the requester's assigned policies to make allow/deny decisions. Think of this like a security guard consulting multiple access lists to determine if an employee's badge grants access to a specific resource at a specific time.

**Request Evaluation Pipeline** processes every secret access request through a structured sequence of authorization checks:

1. **Request Parsing:** Extract the secret path, HTTP method, and request parameters from the incoming API call
2. **Identity Resolution:** Look up the authenticated identity associated with the request token
3. **Policy Collection:** Gather all policies assigned directly to the identity or inherited through group membership
4. **Rule Matching:** For each policy, find rules where the secret path matches the rule's path pattern
5. **Capability Checking:** Verify the requested operation (read/create/update/delete/list) is included in matching rule capabilities
6. **Parameter Validation:** Check request parameters against rule constraints (required/allowed/denied parameters)
7. **Decision Aggregation:** Combine results from all matching rules to produce final allow/deny decision
8. **Audit Logging:** Record the authorization decision with full context for security monitoring

The authorization engine implements **default deny** behavior — if no policy rule explicitly allows an operation, the request is denied. This ensures that new secrets and paths are protected by default until policies explicitly grant access.

**Policy Evaluation Algorithm** determines access by finding the most permissive rule that matches the request path. The algorithm processes rules in the order they appear in policies, allowing more specific rules to override general patterns:

1. Initialize decision as DENY and collect empty capability set

2. For each policy assigned to the requesting identity:
  - a. For each rule in the policy (in order):
    - i. Check if request path matches rule path pattern using wildcard matching
    - ii. If path matches, check if requested operation is in rule capabilities
    - iii. If operation is allowed, validate all parameter constraints
    - iv. If all checks pass, update decision to ALLOW for this operation
3. Return final decision based on whether any rule explicitly allowed the operation

**Path Matching Implementation** uses efficient prefix matching with wildcard support to minimize evaluation overhead for policies with hundreds of rules. The matching algorithm handles both single-segment ( `*` ) and recursive ( `**` ) wildcards:

1. Split both rule path pattern and request path into segments using "/" delimiter
2. Initialize pattern index and path index to 0
3. While both indices are within their respective arrays:
  - a. If pattern segment is "`*`", advance both indices by 1 (exact segment match)
  - b. If pattern segment is "`**`", find next non-wildcard pattern segment and skip path segments until match found
  - c. If pattern segment is literal, compare with path segment for exact match
  - d. If segments don't match, rule doesn't apply to this path
4. Rule matches if both pattern and path are fully consumed or pattern ends with "`**`"

**⚠ Pitfall: Policy Rule Ordering** The order of rules within a policy matters because the engine evaluates them sequentially and stops at the first matching rule. A common mistake is placing broad wildcard patterns before more specific rules, causing the specific rules to never be evaluated. Always order rules from most specific to least specific within each policy.

**Token Policy Inheritance** allows tokens to inherit policies from multiple sources, providing flexible authorization models for different organizational structures. When evaluating a request, the authorization engine considers policies from:

1. **Direct Token Policies:** Policies explicitly assigned when the token was created
2. **Identity Policies:** Default policies assigned to the authenticated identity
3. **Group Policies:** Policies inherited through group membership hierarchies

The final authorization decision considers the union of all applicable policies — if any inherited policy grants access, the operation is allowed.

## Audit Logging

Comprehensive audit logging provides the security monitoring and compliance foundation for the secret management system. Think of audit logs like security camera recordings that capture every interaction with the vault, enabling forensic analysis of security incidents and demonstrating compliance with access control policies.

**Audit Event Structure** captures complete context for every interaction with the secret management system, ensuring security teams can reconstruct the full timeline of any security event:

Field	Type	Description
Timestamp	time.Time	Precise event occurrence time in UTC
RequestID	string	Unique identifier correlating related log entries
EventType	string	Category: "auth", "secret", "policy", "admin"
Operation	string	Specific action: "login", "read", "create", "delete", etc.
Path	string	Secret or resource path being accessed
Identity	string	Authenticated identity performing the operation
TokenID	string	Token accessor (not ID) for privacy
AuthMethod	string	Authentication method used ("token", "tls-cert", "approle")
SourceIP	string	Client IP address for network-based analysis
UserAgent	string	Client application identifier
RequestMetadata	map[string]interface{}	Additional request context and parameters
Result	string	Operation outcome: "success", "denied", "error"
ErrorCode	string	Specific error code for failed operations
ResponseMetadata	map[string]interface{}	Non-sensitive response context
Duration	time.Duration	Request processing time for performance analysis

**Security Event Categories** organize audit events into logical groups that align with common security monitoring and incident response workflows:

Event Type	Operations	Security Purpose
auth	login, logout, token-create, token-revoke	Track identity authentication and session management
secret	read, create, update, delete, list	Monitor access to sensitive data and detect unauthorized attempts
policy	policy-create, policy-update, policy-delete	Audit changes to access control rules
admin	unseal, seal, leader-election, backup	Track administrative operations affecting system security
dynamic	lease-create, lease-renew, lease-revoke	Monitor dynamic secret lifecycle for credential management

**Tamper-Evident Logging** ensures audit logs cannot be modified or deleted by attackers who compromise the secret management system. The logging subsystem implements several protection mechanisms:

- **Append-Only Storage:** Audit logs are written to append-only files or storage systems that prevent modification of existing entries
- **Remote Syslog:** Critical events are immediately forwarded to external log collection systems outside the secret management infrastructure
- **Log Signing:** Each audit entry includes a cryptographic signature that can detect tampering
- **Sequence Numbering:** Monotonic sequence numbers enable detection of missing log entries

#### Decision: Structured JSON Audit Logging

- **Context:** Security teams need to correlate secret management events with other security data sources and must parse logs programmatically for threat detection
- **Options Considered:** Plain text logs, structured JSON, binary format
- **Decision:** Structured JSON with standardized field names
- **Rationale:** JSON provides schema evolution and integrates easily with SIEM systems, log aggregators, and analysis tools. Plain text is hard to parse; binary format lacks tooling support
- **Consequences:** Enables automated security analysis and correlation but requires more storage space than compact binary formats

**Privacy and Compliance Considerations** balance comprehensive security monitoring with protection of sensitive information. The audit logging system carefully excludes certain data from logs to prevent credential exposure:

Never Logged:

- Actual secret values or encrypted content

- Full token IDs (only public accessors are logged)
- Sensitive request parameters like passwords or private keys
- Client certificate private keys or token secrets

Always Logged:

- Secret paths and metadata (but not values)
- Authentication outcomes and identity information
- Authorization decisions and policy evaluations
- Error conditions and security violations
- Timing and performance metrics

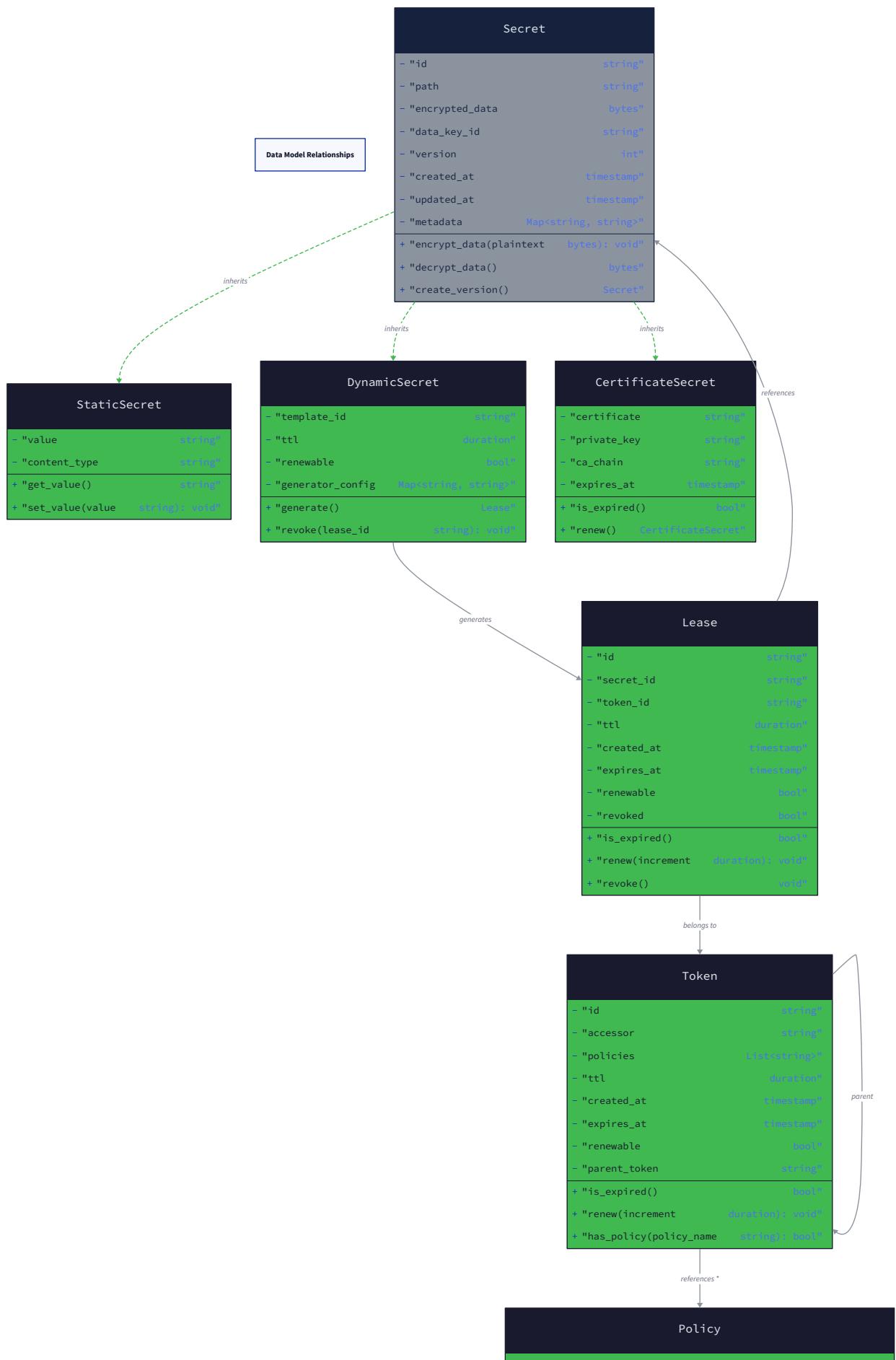
**⚠ Pitfall: Logging Sensitive Data** A common security vulnerability is accidentally logging sensitive information in request parameters or error messages. For example, logging `password=secret123` or including secret values in error responses. The audit system must sanitize all logged data to include only non-sensitive context needed for security monitoring.

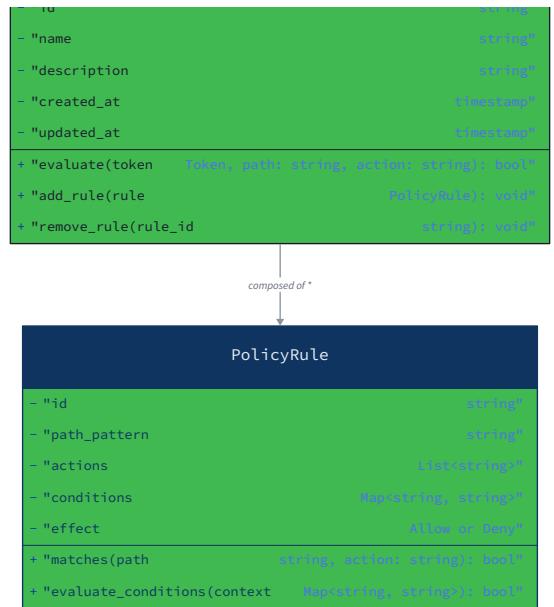
**Performance and Scalability Considerations** ensure audit logging doesn't become a bottleneck for secret management operations. The logging subsystem implements several optimizations:

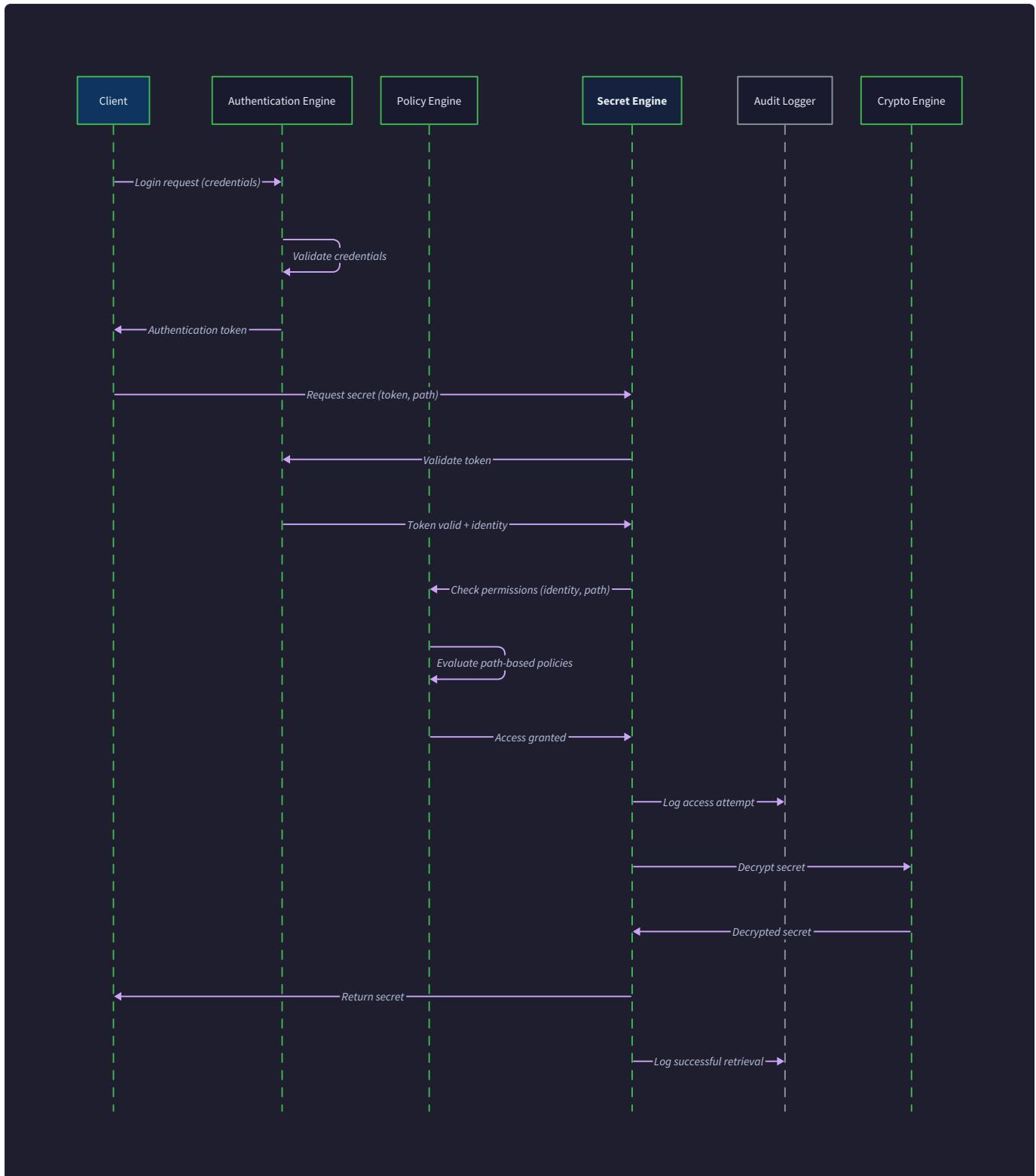
- **Asynchronous Logging:** Audit events are queued and written asynchronously to prevent blocking secret operations
- **Batch Writing:** Multiple audit events are written together to reduce I/O overhead
- **Log Rotation:** Automatic rotation and compression of log files to manage storage growth
- **Sampling:** Non-critical events may be sampled at high request volumes to prevent log flooding

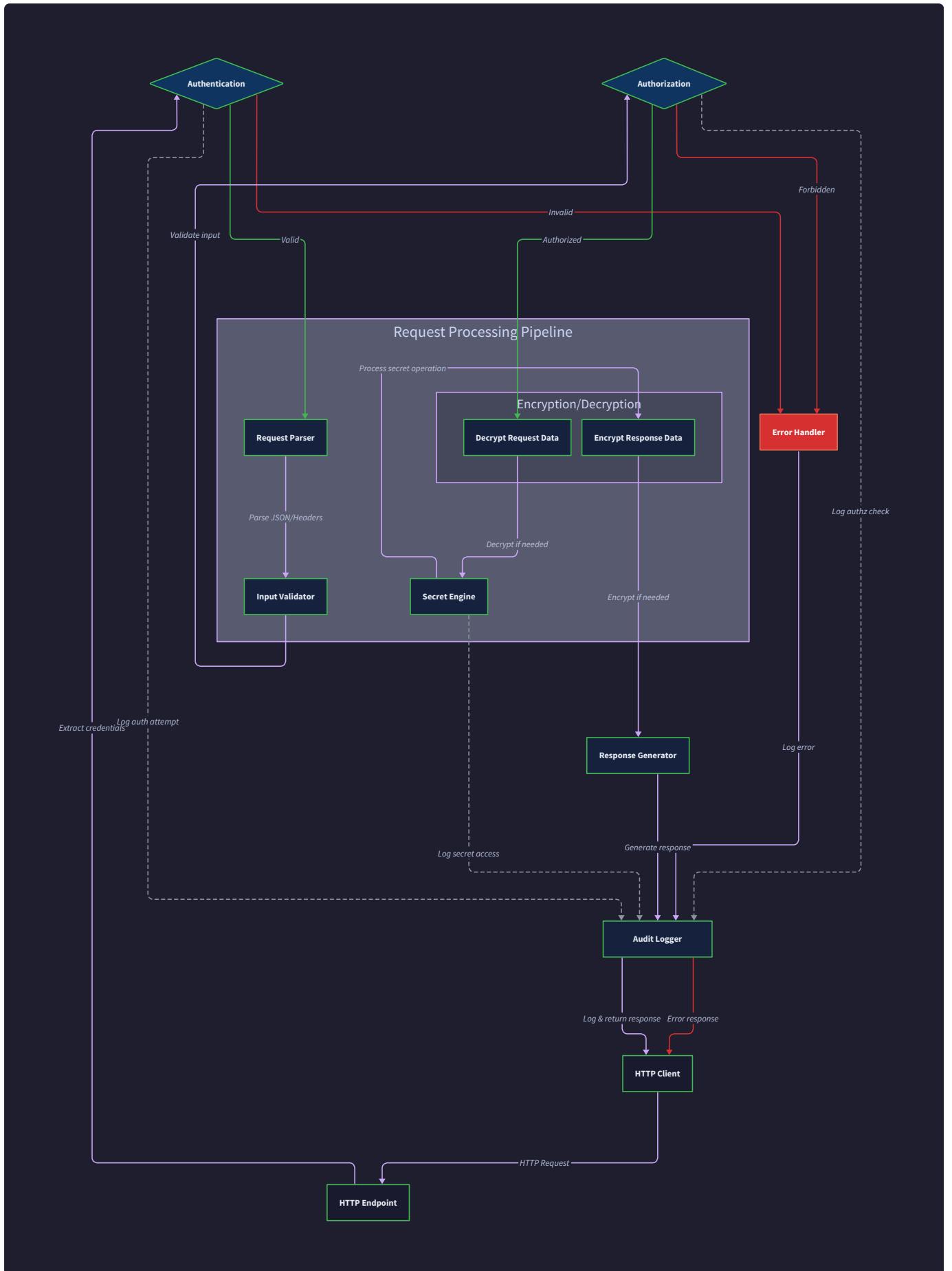
The audit logging system provides configurable log levels to balance detail with performance:

Log Level	Events Included	Use Case
<code>minimal</code>	Authentication, authorization failures, administrative operations	Compliance and basic security monitoring
<code>standard</code>	All secret operations, policy changes, error conditions	Comprehensive security analysis
<code>verbose</code>	Request/response timing, debug information, performance metrics	Troubleshooting and performance optimization









## Common Pitfalls

⚠ **Pitfall: Token Storage Without Expiration** Many implementations create tokens without expiration dates and never clean up the token storage table. This leads to unbounded growth of the token database and potential memory exhaustion. Always implement token TTL with automatic cleanup processes, and consider using short-lived tokens with renewal mechanisms rather than long-lived tokens.

⚠ **Pitfall: Timing Attack Vulnerabilities** When validating tokens or comparing credentials, using standard string comparison functions can leak timing information that allows attackers to gradually guess valid tokens. Always use constant-time comparison functions (like `crypto/subtle.ConstantTimeCompare` in Go) when validating authentication credentials.

⚠ **Pitfall: Policy Rule Conflicts** When multiple policies apply to the same identity, conflicting rules can create security vulnerabilities. For example, one policy might grant `read` access to `secret/**` while another denies access to `secret/admin/*`. The authorization engine must have clear precedence rules — typically "most specific rule wins" or "explicit deny overrides allow" — and these rules must be consistently applied.

⚠ **Pitfall: Insufficient Request Context** Authentication engines often fail to capture sufficient context about the request environment, making forensic analysis difficult. Always log the complete request context including source IP, user agent, request parameters (sanitized), and timing information. This context is critical for detecting credential theft and analyzing security incidents.

⚠ **Pitfall: Group Policy Infinite Recursion** When implementing group-based policy inheritance, circular group memberships can cause infinite recursion during policy evaluation. Always implement cycle detection in group membership resolution, and consider limiting inheritance depth to prevent deep recursive chains that impact performance.

## Implementation Guidance

This subsection provides concrete implementation guidance for building the authentication and policy engine in Go, with a focus on security best practices and production-ready patterns.

## Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Authentication	Header parsing with <code>net/http</code>	JWT middleware with <code>github.com/golang-jwt/jwt</code>
TLS Certificate Handling	<code>crypto/x509</code> and <code>crypto/tls</code>	<code>github.com/cloudflare/cfssl</code> for CA operations
Token Storage	In-memory map with <code>sync.RWMutex</code>	Redis with TTL support
Policy Evaluation	String matching with <code>path/filepath</code>	Compiled regex patterns with <code>regexp</code>
Audit Logging	JSON lines to file with <code>encoding/json</code>	Structured logging with <code>github.com/sirupsen/logrus</code>
Password Hashing	<code>golang.org/x/crypto/argon2</code>	<code>golang.org/x/crypto/scrypt</code> for compatibility

## Recommended File Structure

```
internal/auth/
  engine.go           ← main auth engine and policy evaluation
  engine_test.go      ← auth engine tests
  token.go            ← token creation, validation, and management
  token_test.go       ← token handling tests
  policy.go           ← policy parsing and rule evaluation
  policy_test.go      ← policy evaluation tests
  audit.go            ← audit logging and event capture
  audit_test.go       ← audit logging tests
  types.go            ← shared data structures (Token, Policy, Identity, etc.)

internal/auth/methods/
  token.go           ← bearer token authentication
  mtls.go            ← mutual TLS authentication
  approle.go          ← AppRole authentication method

cmd/auth-test/
  main.go            ← standalone auth testing utility
```

## Authentication Infrastructure (Complete Implementation)

```
// internal/auth/types.go
```

```
package auth
```

```
import (
```

```
    "time"
```

```
)
```

```
// Token represents an authentication token with policies and metadata
```

```
type Token struct {
```

```
    ID          string      `json:"id"`
```

```
    Accessor    string      `json:"accessor"`
```

```
    Policies   []string    `json:"policies"`
```

```
    TokenType  string      `json:"token_type"`
```

```
    CreatedAt  time.Time   `json:"created_at"`
```

```
    ExpiresAt  *time.Time  `json:"expires_at,omitempty"`
```

```
    LastUsedAt *time.Time  `json:"last_used_at,omitempty"`
```

```
    UsageCount int64       `json:"usage_count"`
```

```
    MaxUses    int         `json:"max_uses"`
```

```
    Renewable  bool        `json:"renewable"`
```

```
    ParentToken string     `json:"parent_token"`
```

```
    DisplayName string     `json:"display_name"`
```

```
    Metadata   map[string]string `json:"metadata"`
```

```
    BoundCIDRs []string    `json:"bound_cidrs"`
```

```
}
```

```
// Policy represents an access control policy with rules
```

```
type Policy struct {
```

GO

```

Name      string      `json:"name"`

Rules     []PolicyRule `json:"rules"`

Description string      `json:"description"`

CreatedAt  time.Time   `json:"created_at"`

UpdatedAt   time.Time   `json:"updated_at"`

CreatedBy   string      `json:"created_by"`

Version    int         `json:"version"`

}

// PolicyRule defines access control for a specific path pattern

type PolicyRule struct {

    Path          string      `json:"path"`

    Capabilities  []string    `json:"capabilities"`

    RequiredParameters map[string][]string `json:"required_parameters,omitempty"`

    AllowedParameters map[string][]string `json:"allowed_parameters,omitempty"`

    DeniedParameters map[string][]string `json:"denied_parameters,omitempty"`

    MaxTTL        time.Duration `json:"max_ttl,omitempty"`

    MinWrappingTTL time.Duration `json:"min_wrapping_ttl,omitempty"`

}

// Identity represents an authenticated principal

type Identity struct {

    ID      string      `json:"id"`

    Name    string      `json:"name"`

    Type    string      `json:"type"`

    AuthMethod string    `json:"auth_method"`

    Policies []string    `json:"policies"`

    Groups  []string    `json:"groups"`
}

```

```

   CreatedAt  time.Time      `json:"created_at"`
   LastAuthAt *time.Time     `json:"last_auth_at,omitempty"`
   AuthCount   int64         `json:"auth_count"`
   Metadata    map[string]string `json:"metadata"`
   Disabled    bool          `json:"disabled"`

}

// AuditEvent represents a security event for logging

type AuditEvent struct {

    Timestamp      time.Time      `json:"timestamp"`
    RequestID     string        `json:"request_id"`
    EventType     string        `json:"event_type"`
    Operation      string        `json:"operation"`
    Path           string        `json:"path"`
    Identity       string        `json:"identity"`
    TokenID        string        `json:"token_id"`
    AuthMethod     string        `json:"auth_method"`
    SourceIP       string        `json:"source_ip"`
    UserAgent      string        `json:"user_agent"`
    RequestMetadata map[string]interface{} `json:"request_metadata"`
    Result          string        `json:"result"`
    ErrorCode       string        `json:"error_code"`
    ResponseMetadata map[string]interface{} `json:"response_metadata"`
    Duration        time.Duration `json:"duration"`

}

// Constants for authentication and authorization

const (

```

```
TOKEN_HEADER      = "X-Vault-Token"

DEFAULT_TOKEN_TTL = time.Hour

// Event types

EVENT_AUTH      = "auth"
EVENT_SECRET    = "secret"
EVENT_POLICY    = "policy"
EVENT_ADMIN     = "admin"

// Results

RESULT_SUCCESS  = "success"
RESULT_DENIED   = "denied"
RESULT_ERROR    = "error"

)
```

GO

```
// internal/auth/audit.go

package auth

import (
    "encoding/json"
    "fmt"
    "os"
    "sync"
    "time"
)

// AuditLogger provides tamper-evident audit logging

type AuditLogger struct {
    file      *os.File
    mutex     sync.Mutex
    sequenceNum int64
}

// NewAuditLogger creates a new audit logger writing to the specified file

func NewAuditLogger(filename string) (*AuditLogger, error) {
    file, err := os.OpenFile(filename, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        return nil, fmt.Errorf("failed to open audit log file: %w", err)
    }

    return &AuditLogger{
        file:      file,
        sequenceNum: 0,
```

```
, nil

}

// LogEvent writes an audit event to the log file

func (a *AuditLogger) LogEvent(event AuditEvent) error {

    a.mutex.Lock()

    defer a.mutex.Unlock()

    a.sequenceNum++


    // Add sequence number for tamper detection

    logEntry := struct {

        Sequence int64      `json:"sequence"`

        Event     AuditEvent `json:"event"`

    }{

        Sequence: a.sequenceNum,

        Event:     event,

    }

    data, err := json.Marshal(logEntry)

    if err != nil {

        return fmt.Errorf("failed to marshal audit event: %w", err)

    }

    _, err = a.file.Write	append(data, '\n')

    if err != nil {

        return fmt.Errorf("failed to write audit event: %w", err)

    }

}
```

```
}

// Force write to disk for tamper evidence

return a.file.Sync()

}

// Close closes the audit log file

func (a *AuditLogger) Close() error {

    return a.file.Close()

}
```

## Core Authentication Engine (Skeleton Implementation)

```
// internal/auth/engine.go                                GO

package auth

import (
    "context"
    "crypto/rand"
    "crypto/subtle"
    "encoding/hex"
    "fmt"
    "net/http"
    "strings"
    "sync"
    "time"
)

// Engine handles authentication, authorization, and audit logging

type Engine struct {

    tokens      map[string]*Token
    policies     map[string]*Policy
    identities   map[string]*Identity
    tokenMutex   sync.RWMutex
    policyMutex  sync.RWMutex
    auditLogger  *AuditLogger
}

// NewEngine creates a new authentication engine

func NewEngine(auditLogPath string) (*Engine, error) {
```

```
auditLogger, err := NewAuditLogger(auditLogPath)

if err != nil {
    return nil, fmt.Errorf("failed to create audit logger: %w", err)
}

return &Engine{
    tokens:     make(map[string]*Token),
    policies:   make(map[string]*Policy),
    identities: make(map[string]*Identity),
    auditLogger: auditLogger,
}, nil
}

// AuthenticateRequest extracts and validates authentication from HTTP request

func (e *Engine) AuthenticateRequest(r *http.Request) (*Identity, *Token, error) {

    // TODO 1: Extract token from X-Vault-Token header

    // TODO 2: Validate token exists and is not expired

    // TODO 3: Update token last_used_at and usage_count

    // TODO 4: Look up identity associated with token

    // TODO 5: Check if identity is disabled

    // TODO 6: Validate source IP against token bound_cidrs if configured

    // TODO 7: Return authenticated identity and token

    // Hint: Use constant-time comparison for token validation to prevent timing attacks

    panic("implement me")
}

// AuthorizeRequest evaluates whether identity can perform operation on path
```

```
func (e *Engine) AuthorizeRequest(identity *Identity, operation, path string, params map[string][]string) (bool, error) {

    // TODO 1: Collect all policies assigned to identity (direct + inherited)

    // TODO 2: For each policy, evaluate rules in order

    // TODO 3: For each rule, check if path matches rule.Path pattern

    // TODO 4: If path matches, check if operation is in rule.Capabilities

    // TODO 5: Validate request parameters against rule constraints

    // TODO 6: Return true if any rule explicitly allows, false otherwise

    // Hint: Implement path wildcard matching with * and ** support

    panic("implement me")

}

// CreateToken generates a new authentication token for an identity

func (e *Engine) CreateToken(ctx context.Context, request CreateTokenRequest) (*Token, error) {

    // TODO 1: Generate cryptographically secure random token ID and accessor

    // TODO 2: Validate requested policies exist and identity has permission

    // TODO 3: Calculate expiration time based on TTL and max TTL limits

    // TODO 4: Create token struct with all metadata

    // TODO 5: Store token in token map

    // TODO 6: Log token creation event to audit log

    // Hint: Use crypto/rand for token generation, 32+ bytes recommended

    panic("implement me")

}

// CreatePolicy creates or updates an access control policy

func (e *Engine) CreatePolicy(ctx context.Context, policy *Policy) error {

    // TODO 1: Validate policy name and rule syntax

    // TODO 2: Check for path pattern conflicts with existing policies
```

```
// TODO 3: Validate capability names are recognized

// TODO 4: Store policy in policy map with version increment

// TODO 5: Log policy creation/update to audit log

// Hint: Validate wildcard patterns don't create security holes

panic("implement me")

}

// ValidateTokenConstantTime performs constant-time token comparison

func (e *Engine) ValidateTokenConstantTime(provided, stored string) bool {

    return subtle.ConstantTimeCompare([]byte(provided), []byte(stored)) == 1
}

// matchesPath checks if a request path matches a policy rule path pattern

func (e *Engine) matchesPath(pattern, path string) bool {

    // TODO 1: Split both pattern and path by "/" delimiter

    // TODO 2: Handle single segment wildcard (*) matching

    // TODO 3: Handle recursive wildcard (**) matching

    // TODO 4: Handle literal segment matching

    // TODO 5: Return true if pattern fully matches path

    // Hint: Recursive wildcard matching requires backtracking logic

    panic("implement me")
}

// generateSecureToken creates a cryptographically random token

func (e *Engine) generateSecureToken() (string, error) {

    bytes := make([]byte, 32)

    _, err := rand.Read(bytes)

    if err != nil {

```

```
    return "", err

}

return hex.EncodeToString(bytes), nil

}

type CreateTokenRequest struct {

    Policies      []string

    TTL           time.Duration

    Renewable     bool

    DisplayName   string

    Metadata      map[string]string

}
```

## Path Matching Utility (Complete Implementation)

```
// internal/auth/pathmatch.go                                GO

package auth

import (
    "strings"
)

// PathMatcher provides efficient wildcard path matching for policies

type PathMatcher struct{}


// NewPathMatcher creates a new path matching utility

func NewPathMatcher() *PathMatcher {
    return &PathMatcher{}
}

// Match checks if a path matches a pattern with wildcards

func (pm *PathMatcher) Match(pattern, path string) bool {
    return pm.matchSegments(
        strings.Split(pattern, "/"),
        strings.Split(path, "/"),
        0, 0,
    )
}

// matchSegments recursively matches pattern segments against path segments

func (pm *PathMatcher) matchSegments(pattern, path []string, patternIdx, pathIdx int) bool {
    // If we've consumed both pattern and path, it's a match
    if patternIdx >= len(pattern) && pathIdx >= len(path) {

```

```
        return true

    }

    // If pattern is exhausted but path remains, no match

    if patternIdx >= len(pattern) {

        return false

    }

    // If path is exhausted but pattern has non-recursive wildcards, no match

    if pathIdx >= len(path) {

        // Check if remaining pattern segments are all recursive wildcards

        for i := patternIdx; i < len(pattern); i++ {

            if pattern[i] != "***" {

                return false

            }

        }

        return true

    }

    currentPattern := pattern[patternIdx]

    switch currentPattern {

        case "*":

            // Single segment wildcard - match exactly one path segment

            return pm.matchSegments(pattern, path, patternIdx+1, pathIdx+1)

        case "***":
```

```
// Recursive wildcard - try matching zero or more segments

// First try consuming no path segments (** matches zero)

if pm.matchSegments(pattern, path, patternIdx+1, pathIdx) {

    return true

}

// Then try consuming one path segment and continuing

return pm.matchSegments(pattern, path, patternIdx, pathIdx+1)

}

default:

// Literal segment - must match exactly

if currentPattern == path[pathIdx] {

    return pm.matchSegments(pattern, path, patternIdx+1, pathIdx+1)

}

return false

}

}
```

## Milestone Checkpoints

**Authentication Checkpoint:** After implementing token authentication, test with:

```
# Start your auth engine

go run cmd/auth-test/main.go

# Test token creation

curl -X POST http://localhost:8080/v1/auth/token/create \
-H "X-Vault-Token: root-token" \
-d '{"policies": ["default"], "ttl": "1h"}'

# Expected: JSON response with token ID and accessor

# Verify: Token appears in audit log with creation event
```

BASH

**Policy Evaluation Checkpoint:** Test path-based access control:

```
# Create a policy

curl -X POST http://localhost:8080/v1/sys/policy/test-policy \
-H "X-Vault-Token: root-token" \
-d '{"policy": "path \"secret/app/*\" { capabilities = [\"read\"] }"}'

# Test authorized access

curl -H "X-Vault-Token: <your-token>" \
http://localhost:8080/v1/secret/app/database

# Test unauthorized access (should return 403)

curl -H "X-Vault-Token: <your-token>" \
http://localhost:8080/v1/secret/admin/keys

# Expected: First request succeeds, second returns 403 Forbidden

# Verify: Both attempts appear in audit log with correct results
```

## Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
All requests return 401 Unauthorized	Token lookup failing or tokens not being stored	Check if tokens map contains expected tokens, verify token header extraction	Ensure CreateToken actually stores in tokens map and AuthenticateRequest correctly extracts header
Wildcard policies don't match expected paths	Path matching algorithm incorrect	Add debug logging to matchSegments function showing pattern/path at each step	Test path matching in isolation with unit tests for edge cases
Policy evaluation is slow	Linear search through all policies for each request	Profile policy evaluation with timing measurements	Implement policy indexing by path prefix for faster lookup
Audit logs missing events	Events not being logged or file write failures	Check audit log file permissions and disk space	Ensure every authorization decision calls LogEvent, check file.Sync() errors
Token cleanup not working	Expired tokens accumulating in memory	Check token expiration logic and cleanup goroutine	Implement background cleanup that removes expired tokens from map

## Dynamic Secret Engine

**Milestone(s):** This section implements Milestone 3 (Dynamic Secrets), focusing on on-demand credential generation, lease management, TTL tracking, and automatic revocation across multiple secret backends.

The dynamic secret engine represents a fundamental shift from traditional static secret management to a **credential vending machine** model. Instead of storing long-lived passwords and API keys that accumulate risk over time, this system generates fresh credentials on demand with built-in expiration dates. Think of it like a parking meter that issues time-limited tickets - you get exactly what you need for exactly how long you need it, and the system automatically cleans up when time expires.

### Dynamic Secret Mental Model: Understanding just-in-time credential generation

Before diving into implementation details, it's crucial to understand the mental model that underpins dynamic secret generation. Traditional secret management is like a safety deposit box - you store valuable items (passwords, keys) in a secure location and retrieve them when needed. The problem is that these items never change and accumulate risk the longer they exist.

**Dynamic secrets work more like a hotel key card system.** When you check in, the hotel doesn't give you a master key that works forever. Instead, they program a temporary card that works only for your room and only during your stay. The card is worthless before check-in and automatically stops working at checkout. If someone finds your lost card a week later, it's just plastic.

This mental model extends to database credentials, cloud provider access keys, and API tokens. Instead of creating a shared database user "app\_user" that every application instance uses, the dynamic secret engine creates unique database users like "vault-db-app-20241201-142735" that exist only long enough to serve a specific request or session. When the lease expires, that database user is automatically deleted.

The **credential vending machine** analogy captures another important aspect - the system doesn't pre-generate credentials and store them. Instead, it has the blueprints (role configurations) and generates fresh credentials at request time. This eliminates the storage risk entirely - there's no static credential to steal from the vault's storage backend.

**Key Insight:** Dynamic secrets shift the security model from "protect stored credentials" to "minimize credential lifetime." A compromise can only access resources during the narrow window when credentials are active.

The dynamic secret engine operates on three core principles:

1. **Just-in-time generation:** Credentials are created only when requested, not stored in advance
2. **Lease-based tracking:** Every credential has an explicit expiration time and unique lease ID
3. **Automatic cleanup:** The system takes responsibility for revoking credentials when leases expire

This creates a **blast radius containment** effect. If an application is compromised, the attacker gains access to credentials that automatically expire. If the breach is detected within the lease duration, administrators can revoke specific leases immediately rather than rotating shared credentials across all applications.

## **Secret Backend Plugins: Database and cloud provider credential generation**

The dynamic secret engine uses a plugin architecture where each **secret backend** implements the specific logic for generating, managing, and revoking credentials for a particular service type. This design provides extensibility while maintaining consistent lease management across all credential types.

## Decision: Plugin Architecture for Secret Backends

- **Context:** Different services (databases, cloud providers, SSH) have completely different credential models and APIs for user management
- **Options Considered:** Monolithic engine with hardcoded backends, plugin system with standardized interfaces, service-specific engines
- **Decision:** Plugin system with standardized `SecretBackend` interface
- **Rationale:** Enables extensibility without core engine changes, allows third-party backends, maintains consistent lease semantics across credential types
- **Consequences:** Slightly more complex than monolithic approach, but enables supporting new services without modifying core vault code

Each secret backend plugin implements a standardized interface that abstracts the credential lifecycle:

Method Name	Parameters	Returns	Description
GenerateCredentials	<code>ctx context.Context , roleName string , ttl time.Duration</code>	<code>map[string]interface{} , map[string]interface{} , error</code>	Creates new credentials and returns both the credentials data and revocation metadata
RevokeCredentials	<code>ctx context.Context , revocationData map[string]interface{}</code>	<code>error</code>	Removes or disables the credentials using stored revocation metadata
RenewCredentials	<code>ctx context.Context , revocationData map[string]interface{} , increment time.Duration</code>	<code>error</code>	Extends credential lifetime if backend supports renewal
ValidateRole	<code>ctx context.Context , roleName string , roleConfig map[string]interface{}</code>	<code>error</code>	Validates that role configuration is correct for this backend type

The **database secret backend** serves as the primary example of dynamic credential generation. When configured with connection details and appropriate privileges, it can generate unique database users with specific permissions and automatic cleanup:

#### Database Backend Role Configuration:

Field	Type	Description
Name	string	Unique identifier for this role within the backend
CreationStatements	[]string	SQL statements to execute when creating new database user
RevocationStatements	[]string	SQL statements to execute when cleaning up expired user
RenewStatements	[]string	SQL statements to execute when extending user lifetime (optional)
DefaultTTL	time.Duration	Default lease duration if client doesn't specify
MaxTTL	time.Duration	Maximum lease duration regardless of client request
RenewIncrement	time.Duration	How much time to add during lease renewal
AllowedDomains	[]string	Domain restrictions for cloud provider backends
RoleOptions	map[string]interface{}	Backend-specific configuration options

When a client requests database credentials through the "database/creds/readonly" endpoint, the database backend executes the following sequence:

- Generate unique username:** Create a cryptographically unique username like "vault\_READONLY-20241201-142735-abc123" to avoid collisions
- Execute creation statements:** Run the configured SQL statements, substituting the generated username and a random password
- Test connectivity:** Attempt to authenticate with the new credentials to verify creation succeeded
- Return credential data:** Provide the username and password to the client
- Store revocation metadata:** Save the username and any cleanup information needed for later revocation

#### Database Backend Credential Generation Example:

For a role configured with creation statement:

```
CREATE USER '{{name}}'@'%' IDENTIFIED BY '{{password}}';
GRANT SELECT ON myapp.* TO '{{name}}'@'%';
```

SQL

The backend generates:

- Username: "vault\_READONLY-20241201-142735-abc123"
- Password: 32-character random string

- Executes: `CREATE USER 'vault_READONLY-20241201-142735-abc123'@'%' IDENTIFIED BY 'xK9mP...';`
- Executes: `GRANT SELECT ON myapp.* TO 'vault_READONLY-20241201-142735-abc123'@'%';`

The **cloud provider secret backend** follows similar patterns but generates API keys, access tokens, or service account credentials instead of database users. For AWS, this might involve creating temporary IAM users with specific policies. For Google Cloud, it could generate service account keys with defined roles.

### Decision: Template-Based Statement Configuration

- **Context:** Different databases and versions have varying syntax for user management commands
- **Options Considered:** Hardcoded statements per database type, template system with variable substitution, database-specific plugins
- **Decision:** Template system with `{}{name}` and `{}{password}` variable substitution
- **Rationale:** Provides flexibility for administrators to customize statements for their specific database configuration and security requirements
- **Consequences:** Requires careful template validation to prevent SQL injection, but enables supporting any database with user management capabilities

### Secret Backend Plugin Architecture:

Component	Responsibility	Key Data
<code>SecretBackend</code>	Manages credential lifecycle for specific service type	Backend configuration, role definitions, connection details
<code>BackendRole</code>	Defines credential template and permissions	Creation/revocation statements, TTL limits, role options
<code>CredentialPlugin</code>	Implements service-specific credential generation	Service client connections, API credentials, user management logic

Each backend maintains its own connection pool to the target service to avoid connection overhead during credential generation. The database backend, for example, maintains persistent connections to configured database instances and reuses them across credential requests.

**⚠ Pitfall: Connection Pool Resource Leaks** Many implementations create a new database connection for each credential request, leading to connection exhaustion under load. The database backend should maintain a connection pool and reuse connections across credential operations. Similarly, cloud provider backends should reuse HTTP clients and authenticate once rather than per request.

**⚠ Pitfall: Insufficient Backend Privileges** Backends need elevated privileges to create and delete users, which creates a **privilege escalation** risk if the vault itself is compromised. The backend service account should have only the minimum privileges needed for user lifecycle management. For databases, this means

`CREATE USER`, `DROP USER`, and `GRANT` privileges but not data access. For cloud providers, it means user management permissions but not resource access.

## Lease Management: TTL tracking, renewal, and automatic revocation

The lease management system serves as the **control plane** for all dynamic credentials, tracking their lifecycle from creation through expiration or explicit revocation. Think of it as an **automated parking enforcement system** - it tracks when each parking meter expires and automatically issues tickets (revokes credentials) for expired spots.

Every dynamic credential request creates a **lease** that encapsulates the credential's lifecycle metadata:

### Lease Data Structure:

Field	Type	Description
<code>ID</code>	<code>string</code>	Globally unique lease identifier (UUID)
<code>SecretType</code>	<code>string</code>	Type of secret (database, aws, gcp) for routing to correct backend
<code>BackendPath</code>	<code>string</code>	Which secret backend instance generated this lease
<code>RoleName</code>	<code>string</code>	Role used for credential generation (affects permissions)
<code>CreatedAt</code>	<code>time.Time</code>	When the lease was initially created
<code>ExpiresAt</code>	<code>time.Time</code>	When the lease expires and credentials become invalid
<code>RenewedAt</code>	<code>*time.Time</code>	Last renewal timestamp (nil if never renewed)
<code>RenewalCount</code>	<code>int</code>	How many times this lease has been renewed
<code>MaxTTL</code>	<code>time.Duration</code>	Maximum lifetime regardless of renewals
<code>Renewable</code>	<code>bool</code>	Whether this lease can be extended before expiration
<code>TokenID</code>	<code>string</code>	Authentication token that created this lease (for audit)
<code>RevocationData</code>	<code>map[string]interface{}</code>	Backend-specific data needed for credential cleanup
<code>SecretData</code>	<code>map[string]interface{}</code>	Credential information returned to client (may be empty for security)
<code>Status</code>	<code>string</code>	Current lease state (active, expired, revoked)

The lease management system operates through several coordinated processes:

### Lease Creation Process:

1. **Validate request:** Ensure client has permission for the requested backend path and role
2. **Generate lease ID:** Create globally unique identifier for tracking this credential lifecycle
3. **Calculate expiration:** Determine lease duration based on client request, role defaults, and maximum limits
4. **Invoke backend:** Call the appropriate secret backend to generate actual credentials
5. **Store lease record:** Persist lease metadata for tracking and future revocation
6. **Index by expiration:** Add lease to expiration index for efficient cleanup scanning
7. **Return credentials:** Provide credential data and lease ID to client

#### **Decision: Separate Credential Data from Revocation Data**

- **Context:** Backends need different information for credential generation vs. cleanup, and storing credentials increases attack surface
- **Options Considered:** Store complete credentials in lease, store only revocation data, store both separately
- **Decision:** Store revocation data in lease, return credentials to client without persisting
- **Rationale:** Minimizes stored secrets (credentials exist only during client session), while preserving cleanup capability through revocation metadata
- **Consequences:** Clients must store credentials themselves, but vault storage doesn't contain usable credentials

**Lease Renewal Process:** Renewable leases can be extended before expiration to avoid credential cycling for long-running processes. The renewal process implements several safety checks:

1. **Validate lease existence:** Ensure the lease ID exists and is currently active
2. **Check renewable flag:** Verify that this lease type supports renewal
3. **Enforce maximum TTL:** Ensure renewed lease won't exceed the configured maximum lifetime
4. **Calculate new expiration:** Add the requested increment (bounded by role configuration)
5. **Update lease record:** Modify expiration time, renewal count, and last renewed timestamp
6. **Update expiration index:** Move lease to new expiration time bucket for cleanup tracking
7. **Invoke backend renewal:** Allow backend to extend underlying credentials if needed

#### **Lease Expiration Tracking:**

The lease management system maintains an **expiration index** that groups leases by expiration time for efficient batch processing:

#### **LeaseIndex Data Structure:**

Field	Type	Description
ByExpiration	<code>map[time.Time][]string</code>	Groups lease IDs by expiration timestamp for batch processing
ByToken	<code>map[string][]string</code>	Groups leases by creating token for bulk revocation
ByBackend	<code>map[string][]string</code>	Groups leases by backend path for backend-specific operations
ActiveCount	<code>int64</code>	Running count of active leases for monitoring

The expiration index enables the **lease reaper** process to efficiently find expired leases without scanning the entire lease database:

#### Lease Reaper Algorithm:

1. **Scan expiration index:** Find all lease buckets with expiration times before current time
2. **Batch expired leases:** Collect lease IDs from expired buckets (typically 100-1000 per batch)
3. **Load lease details:** Retrieve full lease records for the expired lease IDs
4. **Group by backend:** Organize leases by backend type for efficient revocation
5. **Revoke credentials:** Call backend revocation methods with stored revocation data
6. **Update lease status:** Mark leases as revoked in storage
7. **Update indexes:** Remove revoked leases from all index structures
8. **Log revocation:** Record successful revocation in audit log for compliance

**Key Insight:** The lease reaper runs as a background process every 30-60 seconds, but lease expiration times are honored by the authorization system immediately. This means expired credentials are denied access even if revocation hasn't completed yet.

**⚠️ Pitfall: Lease Reaper Failure Handling** If the lease reaper process crashes or becomes unable to revoke credentials, expired leases accumulate and credentials remain active past their intended lifetime. The system should implement **revocation retry logic** with exponential backoff and **alerting** when revocation fails repeatedly. Additionally, the authorization system should reject requests using expired leases regardless of backend credential status.

#### Lease Renewal Limits:

Scenario	Original TTL	Max TTL	Renewal Request	Result	Rationale
Normal renewal	1 hour	8 hours	+1 hour after 30 minutes	New expiration: 1.5 hours from creation	Within limits
Excessive renewal	1 hour	8 hours	+24 hours after 30 minutes	New expiration: 8 hours from creation	Capped at max TTL
Expired lease	1 hour	8 hours	+1 hour after 2 hours	Error: lease expired	Cannot renew expired leases
Max TTL reached	8 hours	8 hours	+1 hour after 7 hours	Error: would exceed max TTL	Prevent infinite extension

## Revocation Engine: Cleanup of expired credentials across backends

The revocation engine serves as the **cleanup crew** for the dynamic secret system, ensuring that expired or compromised credentials are properly removed from target systems. Unlike lease management which tracks metadata, the revocation engine actually interacts with external systems to disable database users, delete cloud IAM keys, and clean up other credential artifacts.

Think of the revocation engine as a **building security system that automatically deactivates keycards**.

When an employee leaves or a keycard expires, the system doesn't just mark it as inactive in the database - it actively tells all door readers that the keycard should no longer grant access.

### Revocation Engine Architecture:

The revocation engine operates through a **queue-based system** that provides reliability, retry logic, and priority handling for credential cleanup operations:

### RevocationQueue Data Structure:

Field	Type	Description
LeaseID	string	Unique identifier of the lease being revoked
BackendPath	string	Which secret backend should handle this revocation
RevocationData	map[string]interface{}	Backend-specific data needed for credential cleanup
QueuedAt	time.Time	When this revocation was first queued
AttemptCount	int	How many revocation attempts have been made
NextAttempt	time.Time	When to try revocation again (for failed attempts)
Priority	int	Revocation priority (1=immediate, 2=normal, 3=cleanup)

The revocation engine processes this queue continuously, handling both **automatic expiration** and **explicit revocation** requests:

#### **Revocation Processing Algorithm:**

1. **Dequeue revocation requests:** Pull revocation requests from queue, ordered by priority and next attempt time
2. **Load backend configuration:** Retrieve the appropriate secret backend for this revocation
3. **Execute revocation:** Call backend-specific revocation method with stored revocation data
4. **Handle success:** Remove revocation request from queue and update lease status
5. **Handle failure:** Increment attempt count, calculate next retry time with exponential backoff
6. **Re-queue failed attempts:** Put failed revocations back in queue with updated retry timestamp
7. **Alert on persistent failures:** Send alerts if revocation fails repeatedly (typically after 5-10 attempts)

#### **Decision: Asynchronous Revocation with Retry Queue**

- **Context:** Credential revocation involves network calls to external systems that may be temporarily unavailable
- **Options Considered:** Synchronous revocation blocking lease expiration, fire-and-forget async revocation, queue-based async revocation with retries
- **Decision:** Queue-based asynchronous revocation with exponential backoff retry
- **Rationale:** Provides reliability without blocking lease expiration, handles transient network failures gracefully, enables monitoring and alerting for persistent failures
- **Consequences:** Adds complexity but ensures credentials are eventually revoked even if target systems are temporarily unreachable

#### **Backend-Specific Revocation Strategies:**

Different types of credentials require different revocation approaches, each with unique failure modes and recovery strategies:

Backend Type	Revocation Method	Failure Scenarios	Recovery Strategy
Database	DROP USER statement	Connection timeout, database offline, user doesn't exist	Retry with connection pool, verify user absence
AWS IAM	Delete user API call	API rate limits, permission denied, user already deleted	Retry with backoff, ignore "user not found" errors
Cloud KMS	Disable service account key	Network partition, quota exceeded, key already disabled	Retry different region, ignore "already disabled"
SSH Certificate	Revocation list update	Certificate authority offline, signing key unavailable	Queue for CA recovery, maintain revocation list

### Database Credential Revocation Example:

When revoking database credentials, the database backend executes the configured revocation statements with proper error handling:

### Database Revocation Process:

- Extract username:** Get database username from revocation data (e.g., "vault-readonly-20241201-142735-abc123")
- Terminate active sessions:** Kill any active database sessions for this user to prevent continued access
- Execute revocation statements:** Run configured SQL statements, typically `DROP USER '{{name}}'@'%'`
- Verify deletion:** Query user tables to confirm the user no longer exists
- Handle cleanup errors:** If user doesn't exist, treat as successful (idempotent operation)
- Log revocation result:** Record success or failure details for audit and debugging

**⚠ Pitfall: Revocation Statement Dependencies** Database revocation statements must handle dependencies correctly. For example, MySQL requires dropping user privileges before dropping the user itself. The revocation statements should be ordered to handle these dependencies: `REVOKE ALL PRIVILEGES ON *.* FROM '{{name}}'@'%'; DROP USER '{{name}}'@'%';`

### Revocation Failure Handling:

The revocation engine implements **exponential backoff** to handle transient failures without overwhelming external systems:

### Retry Schedule:

- Attempt 1: Immediate
- Attempt 2: 1 minute delay
- Attempt 3: 2 minute delay
- Attempt 4: 4 minute delay

- Attempt 5: 8 minute delay
- Attempt 6+: 16 minute delay (maximum)

After 10 consecutive failures, the revocation request is moved to a **dead letter queue** for manual intervention, and an alert is generated for the operations team.

### **Emergency Revocation:**

For security incidents, the revocation engine supports **priority revocation** that bypasses normal queue ordering:

### **Revocation Priority Levels:**

1. **Immediate (Priority 1):** Security incident response, suspected compromise - processed within seconds
2. **Normal (Priority 2):** Regular lease expiration - processed within 1-2 minutes
3. **Cleanup (Priority 3):** Background cleanup, orphaned resources - processed during off-peak hours

Priority 1 revocations also trigger **parallel revocation** across multiple backend instances and regions to ensure rapid credential invalidation.

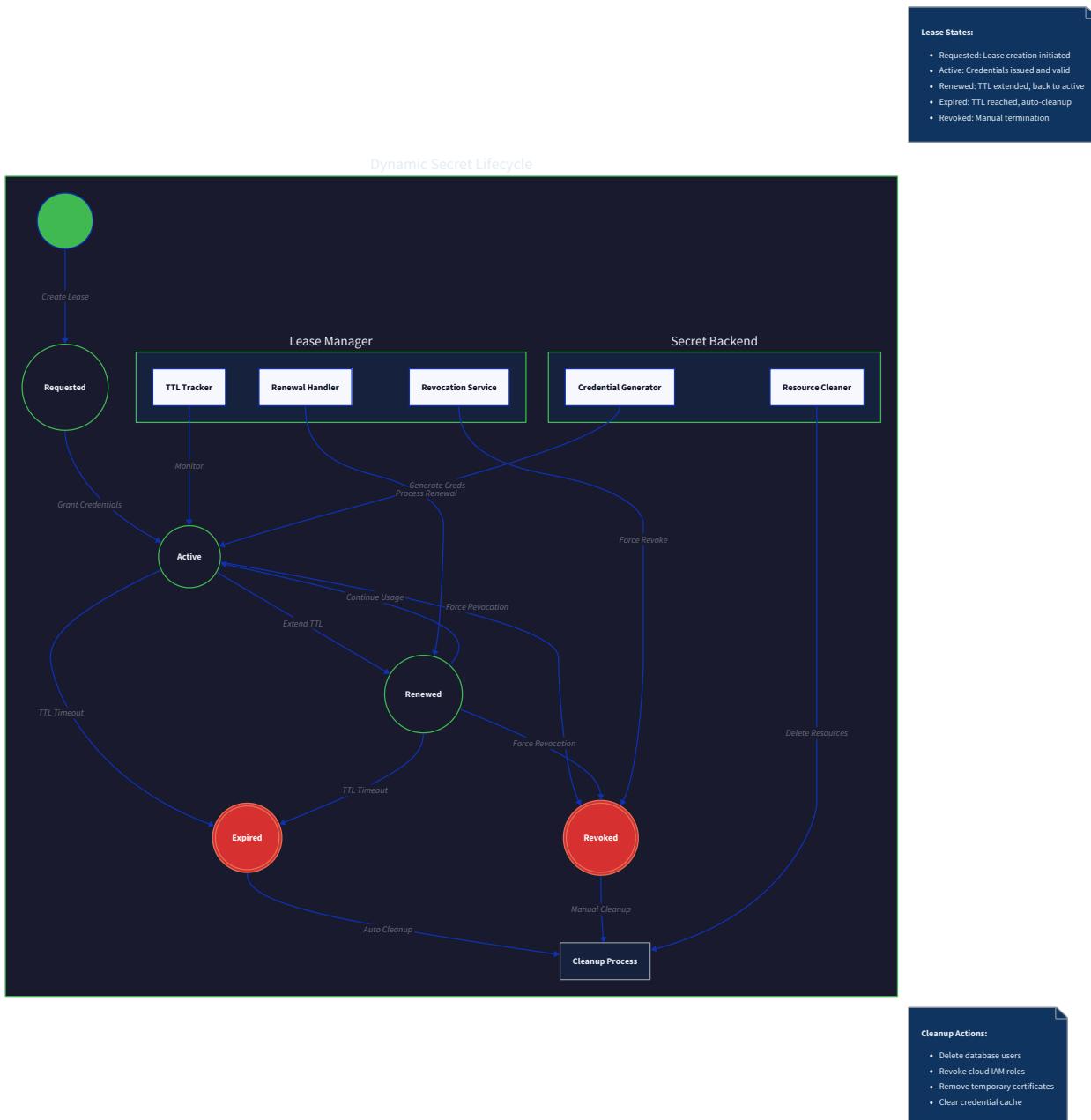
### **Bulk Revocation Operations:**

The revocation engine supports bulk operations for scenarios like token revocation (revoking all leases created by a specific token) or backend rotation (revoking all credentials from a compromised backend):

### **Bulk Revocation Algorithm:**

1. **Identify affected leases:** Query lease index by token ID, backend path, or other criteria
2. **Queue all revocations:** Add revocation requests for all identified leases
3. **Process with concurrency limits:** Revoke credentials in parallel but respect backend rate limits
4. **Track completion:** Monitor bulk revocation progress and report completion status
5. **Handle partial failures:** Continue bulk operation even if individual revocations fail

This enables rapid response to security incidents where entire token families or backend instances need immediate revocation.



## Implementation Guidance

The dynamic secret engine requires careful coordination between lease management, backend plugins, and revocation processing. This implementation guidance provides complete infrastructure code and detailed skeletons for the core learning components.

### A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Lease Storage	In-memory map with periodic disk sync	Embedded database (BadgerDB) with transactions
Revocation Queue	Go channels with worker goroutines	Persistent queue (Redis/database-backed)
Backend Plugins	Direct interface implementation	Plugin system with RPC/gRPC
Time Management	Standard <code>time.Time</code> with tickers	Hierarchical timing wheels for efficiency
Credential Generation	Standard <code>crypto/rand</code> for passwords	HSM integration for key material

## B. Recommended File/Module Structure:

```

internal/dynamic/
    engine.go           ← main dynamic engine
    engine_test.go     ← engine tests
    lease.go           ← lease management
    lease_index.go     ← lease indexing and expiration tracking
    revocation.go      ← revocation engine and queue
backends/
    interface.go        ← SecretBackend interface definition
    database/
        database.go    ← database credential backend
        mysql.go        ← MySQL-specific implementation
        postgres.go     ← PostgreSQL-specific implementation
aws/
    aws.go             ← AWS IAM backend
    registry.go        ← backend plugin registry
internal/
    credential_generator.go ← secure password/key generation
    retry.go           ← exponential backoff utilities
  
```

## C. Infrastructure Starter Code:

### Credential Generator (complete implementation):

GO

```
// internal/dynamic/internal/credential_generator.go

package internal

import (
    "crypto/rand"
    "fmt"
    "math/big"
    "strings"
    "time"
)

const (
    // Character sets for password generation
    PasswordChars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
    UsernameChars = "abcdefghijklmnopqrstuvwxyz0123456789"
    DefaultPasswordLength = 32
    DefaultUsernameLength = 20
)

// CredentialGenerator provides secure random credential generation

type CredentialGenerator struct{}


// GeneratePassword creates a cryptographically secure random password

func (g *CredentialGenerator) GeneratePassword(length int) (string, error) {
    if length <= 0 {
        length = DefaultPasswordLength
    }

    password := make([]byte, length)
```

```
for i := range password {

    charIndex, err := rand.Int(rand.Reader, big.NewInt(int64(len>PasswordChars)))

    if err != nil {

        return "", fmt.Errorf("failed to generate random character: %w", err)

    }

    password[i] = PasswordChars[charIndex.Int64()]

}

return string(password), nil
}

// GenerateUsername creates a unique username with timestamp and random suffix

func (g *CredentialGenerator) GenerateUsername(prefix string) (string, error) {

    timestamp := time.Now().Format("20060102-150405")

    // Generate random suffix

    suffix := make([]byte, 6)

    for i := range suffix {

        charIndex, err := rand.Int(rand.Reader, big.NewInt(int64(lenUsernameChars)))

        if err != nil {

            return "", fmt.Errorf("failed to generate random suffix: %w", err)

        }

        suffix[i] = UsernameChars[charIndex.Int64()]

    }

    // Format: vault-prefix-20241201-142735-abc123

    username := fmt.Sprintf("vault-%s-%s-%s", prefix, timestamp, string(suffix))
}
```

```

// Ensure username doesn't exceed common database limits (32-64 chars)

if len(username) > 32 {

    // Truncate prefix if needed

    maxPrefix := 32 - len(timestamp) - len(suffix) - len("vault--")

    if len(prefix) > maxPrefix && maxPrefix > 0 {

        prefix = prefix[:maxPrefix]

        username = fmt.Sprintf("vault-%s-%s-%s", prefix, timestamp, string(suffix))

    }

}

return username, nil
}

// SubstituteTemplate replaces {{name}} and {{password}} in template strings

func (g *CredentialGenerator) SubstituteTemplate(template, name, password string) string {

    result := strings.ReplaceAll(template, "{{name}}", name)

    result = strings.ReplaceAll(result, "{{password}}", password)

    return result
}

```

### Retry Logic (complete implementation):

GO

```
// internal/dynamic/internal/retry.go

package internal

import (
    "context"
    "fmt"
    "math"
    "time"
)

// RetryConfig defines exponential backoff parameters

type RetryConfig struct {
    InitialDelay time.Duration
    MaxDelay     time.Duration
    Multiplier   float64
    MaxAttempts  int
}

// DefaultRetryConfig provides sensible defaults for revocation retries

var DefaultRetryConfig = RetryConfig{
    InitialDelay: 1 * time.Minute,
    MaxDelay:     16 * time.Minute,
    Multiplier:   2.0,
    MaxAttempts:  10,
}

// CalculateDelay computes next retry delay using exponential backoff

func (c RetryConfig) CalculateDelay(attemptNumber int) time.Duration {
    if attemptNumber <= 1 {
```

```
    return 0 // First attempt is immediate

}

delay := float64(c.InitialDelay) * math.Pow(c.Multiplier, float64(attemptNumber-2))

if time.Duration(delay) > c.MaxDelay {

    delay = float64(c.MaxDelay)

}

return time.Duration(delay)

}

// ShouldRetry determines if another attempt should be made

func (c RetryConfig) ShouldRetry(attemptNumber int) bool {

    return attemptNumber < c.MaxAttempts

}

// RetryOperation executes operation with exponential backoff

func RetryOperation(ctx context.Context, config RetryConfig, operation func() error) error {

    var lastErr error

    for attempt := 1; attempt <= config.MaxAttempts; attempt++ {

        if attempt > 1 {

            delay := config.CalculateDelay(attempt)

            select {

                case <-time.After(delay):

                    // Continue to retry

                case <-ctx.Done():

                    return fmt.Errorf("retry cancelled: %w", ctx.Err())

```

```
    }

}

if err := operation(); err != nil {

    lastErr = err

    if !config.ShouldRetry(attempt) {

        break

    }

    continue

}

return nil // Success

}

return fmt.Errorf("operation failed after %d attempts: %w", config.MaxAttempts, lastErr)
}
```

#### D. Core Logic Skeleton Code:

**Dynamic Engine (core implementation skeleton):**

GO

```
// internal/dynamic/engine.go

package dynamic

import (
    "context"
    "fmt"
    "sync"
    "time"

    "your-project/internal/dynamic/backends"
    "your-project/internal/dynamic/internal"
)

// Engine manages dynamic secret generation and lease tracking

type Engine struct {

    backends      map[string]backends.SecretBackend
    leases        map[string]*Lease
    leaseIndex    *LeaseIndex
    revocationQueue chan *RevocationQueue
    credGen        *internal.CredentialGenerator

    // Synchronization
    leaseMutex     sync.RWMutex

    // Background workers
    reaperTicker  *time.Ticker
    stopChan      chan struct{}
}
```

```

// NewEngine creates a new dynamic secret engine

func NewEngine() *Engine {
    return &Engine{
        backends:      make(map[string]backends.SecretBackend),
        leases:       make(map[string]*Lease),
        leaseIndex:    NewLeaseIndex(),
        revocationQueue: make(chan *RevocationQueue, 1000),
        credGen:       &internal.CredentialGenerator{},
        stopChan:      make(chan struct{}),
    }
}

// RegisterBackend adds a secret backend at the specified path

func (e *Engine) RegisterBackend(path string, backend backends.SecretBackend) error {
    // TODO 1: Validate that path is not already registered

    // TODO 2: Store backend in backends map

    // TODO 3: Initialize backend if it has an Init method

    // Hint: Use path as key in backends map

    return nil
}

// GenerateSecret creates new dynamic credentials with lease tracking

func (e *Engine) GenerateSecret(ctx context.Context, backendPath, roleName string, ttl time.Duration, tokenID string) (*Lease, map[string]interface{}, error) {
    // TODO 1: Look up backend by backendPath in backends map

    // TODO 2: Generate unique lease ID using crypto/rand or UUID library

    // TODO 3: Calculate lease expiration time (created + ttl)

    // TODO 4: Call backend.GenerateCredentials(ctx, roleName, ttl)
}

```

```
// TODO 5: Create Lease struct with all required fields

// TODO 6: Store lease in leases map with leaseMutex write lock

// TODO 7: Add lease to expiration index for reaper processing

// TODO 8: Return lease and credentials (credentials are not stored)

// Hint: backend.GenerateCredentials returns (credentials, revocationData, error)

return nil, nil, nil

}

// RenewLease extends an existing lease if renewable

func (e *Engine) RenewLease(ctx context.Context, leaseID string, increment time.Duration)
(*Lease, error) {

    // TODO 1: Acquire read lock and look up lease by ID

    // TODO 2: Validate lease exists and is currently active

    // TODO 3: Check if lease.Renewable is true

    // TODO 4: Calculate new expiration (current + increment)

    // TODO 5: Ensure new expiration doesn't exceed lease.MaxTTL from creation time

    // TODO 6: Upgrade to write lock and update lease expiration fields

    // TODO 7: Update lease index with new expiration time

    // TODO 8: Call backend.RenewCredentials if backend supports it

    // Hint: New expiration = min(current + increment, created + maxTTL)

    return nil, nil

}

// RevokeLease immediately revokes a lease and queues credential cleanup

func (e *Engine) RevokeLease(ctx context.Context, leaseID string) error {

    // TODO 1: Look up lease and validate it exists

    // TODO 2: Update lease status to "revoked"

    // TODO 3: Remove lease from expiration index
```

```

// TODO 4: Create RevocationQueue entry with priority 1 (immediate)

// TODO 5: Send revocation request to revocation queue channel

// TODO 6: Return success immediately (async revocation)

// Hint: Don't wait for actual credential cleanup - queue it

return nil

}

// startLeaseReaper begins background process to clean up expired leases

func (e *Engine) startLeaseReaper() {

    // TODO 1: Create ticker that runs every 30 seconds

    // TODO 2: Start goroutine that selects on ticker and stopChan

    // TODO 3: On each tick, call e.processExpiredLeases()

    // TODO 4: Store ticker in e.reaperTicker for cleanup

    // Hint: Use time.NewTicker(30 * time.Second)

}

// processExpiredLeases finds expired leases and queues them for revocation

func (e *Engine) processExpiredLeases() {

    // TODO 1: Get current time and find all leases expired before now

    // TODO 2: Use lease index to efficiently find expired lease IDs

    // TODO 3: For each expired lease, create RevocationQueue entry

    // TODO 4: Queue revocation requests (priority 2 for normal expiration)

    // TODO 5: Update lease status to "expired"

    // TODO 6: Remove from expiration index

    // Hint: Process in batches of 100-500 to avoid memory spikes

}

// startRevocationWorkers begins background workers to process revocation queue

```

```

func (e *Engine) startRevocationWorkers(numWorkers int) {

    // TODO 1: Start numWorkers goroutines

    // TODO 2: Each worker should select on revocationQueue and stopChan

    // TODO 3: For each revocation request, call e.processRevocation()

    // TODO 4: Handle panics in worker goroutines with recovery

    // Hint: Use worker pool pattern with shared revocation queue

}

// processRevocation executes actual credential cleanup with retry logic

func (e *Engine) processRevocation(revReq *RevocationQueue) {

    // TODO 1: Look up backend by revReq.BackendPath

    // TODO 2: Call backend.RevokeCredentials with revReq.RevocationData

    // TODO 3: If successful, mark revocation complete and return

    // TODO 4: If failed, increment revReq.AttemptCount

    // TODO 5: Calculate next retry time using exponential backoff

    // TODO 6: If under max attempts, re-queue for retry

    // TODO 7: If max attempts exceeded, move to dead letter queue and alert

    // Hint: Use internal.DefaultRetryConfig for backoff calculation

}

```

### **Lease Index (core data structure skeleton):**

GO

```
// internal/dynamic/lease_index.go

package dynamic

import (
    "sync"
    "time"
)

// LeaseIndex provides efficient lookup of leases by various criteria

type LeaseIndex struct {

    ByExpiration map[time.Time][]string
    ByToken      map[string][]string
    ByBackend    map[string][]string
    ActiveCount  int64

    mutex sync.RWMutex
}

// NewLeaseIndex creates an empty lease index

func NewLeaseIndex() *LeaseIndex {
    return &LeaseIndex{
        ByExpiration: make(map[time.Time][]string),
        ByToken:      make(map[string][]string),
        ByBackend:    make(map[string][]string),
        ActiveCount:  0,
    }
}

// AddLease indexes a new lease by all relevant criteria
```

```

func (idx *LeaseIndex) AddLease(lease *Lease) {

    // TODO 1: Acquire write lock

    // TODO 2: Add lease.ID to ByExpiration[lease.ExpiresAt] slice

    // TODO 3: Add lease.ID to ByToken[lease.TokenID] slice

    // TODO 4: Add lease.ID to ByBackend[lease.BackendPath] slice

    // TODO 5: Increment ActiveCount

    // Hint: Initialize slices if map key doesn't exist yet

}

// RemoveLease removes a lease from all indexes

func (idx *LeaseIndex) RemoveLease(lease *Lease) {

    // TODO 1: Acquire write lock

    // TODO 2: Remove lease.ID from ByExpiration[lease.ExpiresAt] slice

    // TODO 3: Remove lease.ID from ByToken[lease.TokenID] slice

    // TODO 4: Remove lease.ID from ByBackend[lease.BackendPath] slice

    // TODO 5: Decrement ActiveCount

    // TODO 6: Clean up empty slices/map entries to prevent memory leaks

    // Hint: Use slice filtering to remove specific lease ID

}

// GetExpiredLeases returns all lease IDs that expired before the given time

func (idx *LeaseIndex) GetExpiredLeases(before time.Time) []string {

    // TODO 1: Acquire read lock

    // TODO 2: Iterate through ByExpiration map keys (expiration times)

    // TODO 3: Collect lease ID slices where expiration time <= before

    // TODO 4: Flatten collected slices into single lease ID slice

    // TODO 5: Return deduplicated list of expired lease IDs

    // Hint: Use make([]string, 0, estimatedSize) for efficiency
}

```

```

    return nil

}

// GetLeasesByToken returns all lease IDs created by the specified token

func (idx *LeaseIndex) GetLeasesByToken(tokenID string) []string {
    // TODO 1: Acquire read lock

    // TODO 2: Look up tokenID in ByToken map

    // TODO 3: Return copy of lease ID slice (don't return internal slice)

    // TODO 4: Return empty slice if token not found

    // Hint: Use append([]string(nil), slice...) to copy slice

    return nil
}

```

## E. Language-Specific Hints:

- Use `sync.RWMutex` for lease storage to allow concurrent reads while protecting writes
- Use `crypto/rand` for secure lease ID generation: `uuid.NewRandom()` or similar
- Use `time.NewTicker()` for periodic lease reaper execution
- Use channel-based worker pools for revocation processing: `make(chan *RevocationQueue, bufferSize)`
- Use `context.Context` throughout for cancellation and timeouts
- Use `database/sql` with connection pooling for database backends
- Store sensitive data like database passwords in `SecretData` map, not as struct fields
- Use `json` tags on structs for configuration serialization

## F. Milestone Checkpoint:

After implementing the dynamic secret engine:

### What to run:

```
go test ./internal/dynamic/...
go run cmd/server/main.go &

curl -H "X-Vault-Token: $TOKEN" -X POST http://localhost:8443/v1/database/config/mydb \
-d '{"connection_url": "user:pass@tcp(localhost:3306)/", "allowed_roles": ["readonly"]}' 

curl -H "X-Vault-Token: $TOKEN" -X POST http://localhost:8443/v1/database/roles/readonly \
-d '{"creation_statements": ["CREATE USER {{name}} IDENTIFIED BY {{password}}", "GRANT SELECT ON myapp.* TO {{name}}"], "default_ttl": "1h", "max_ttl": "24h"}'

curl -H "X-Vault-Token: $TOKEN" -X GET http://localhost:8443/v1/database/creds/readonly
```

### Expected output:

```
{                                                 JSON
  "lease_id": "database/creds/readonly/abc123-def456-789012",
  "renewable": true,
  "lease_duration": 3600,
  "data": {
    "username": "vault_READONLY-20241201-142735-xyz789",
    "password": "A1b2C3d4E5f6G7h8I9j0K1l2M3n405p6"
  }
}
```

### What to verify:

- Database should contain new user with generated username
- Lease should appear in lease index and be tracked for expiration
- After lease TTL expires, database user should be automatically removed
- Lease renewal should extend expiration time without creating new user

### Signs something is wrong:

- Credentials returned but no database user created → Check backend connection and privileges
- User created but not removed after expiration → Check lease reaper and revocation queue processing
- Lease renewal fails → Verify MaxTTL calculation and lease status checks
- High memory usage → Check for lease index cleanup and slice memory leaks

# Unsealing and High Availability

**Milestone(s):** This section implements Milestone 4 (Unsealing & High Availability), focusing on Shamir's secret sharing for master key protection, seal/unseal operations, distributed consensus for high availability, and cloud KMS integration for automated unsealing.

The final milestone transforms our secret management system from a single-node service into a production-ready, highly available cluster that can operate even when some nodes fail. Think of this transformation like converting a personal safe in your home office into a bank's vault system with multiple security officers, where several officers must work together to open the vault, but the bank remains operational even if some officers are unavailable.

This milestone introduces two critical capabilities that work together to provide enterprise-grade security and availability. First, **Shamir's secret sharing** protects the master key by splitting it into multiple shares, ensuring that no single person or compromised system can access all secrets. Second, **high availability clustering** ensures the secret management service remains accessible even during node failures, network partitions, or maintenance windows.

The unsealing mechanism addresses a fundamental security principle: the master key should never exist in plaintext on persistent storage, and the system should start in a "sealed" state that requires human intervention or trusted external systems to become operational. This provides protection against scenarios like stolen hard drives, compromised backups, or unauthorized system restarts.

The high availability features transform our system from a single point of failure into a distributed service that can tolerate node failures while maintaining consistency. This is critical for production environments where secret management downtime can cascade into application outages across an entire infrastructure.

## Shamir's Secret Sharing: Splitting Master Key into Shares for Security

Shamir's secret sharing provides the cryptographic foundation for protecting our master key through a threshold scheme that distributes trust across multiple parties. Think of it like a safety deposit box that requires multiple keys held by different bank officers - no single officer can access the contents alone, but any sufficient subset of officers working together can open the box.

The mathematical elegance of Shamir's scheme lies in polynomial interpolation over finite fields. To protect a secret with a threshold of  $t$  shares from a total of  $n$  shares, we construct a random polynomial of degree  $t-1$  where the secret is the y-intercept (constant term). Each share represents a point on this polynomial, and any  $t$  points are sufficient to reconstruct the original polynomial and recover the secret. Fewer than  $t$  points reveal no information about the secret due to the randomness of the higher-order coefficients.

## Decision: Shamir's Secret Sharing for Master Key Protection

- **Context:** Master key must be protected against single-point-of-compromise while remaining accessible for legitimate operations
- **Options Considered:**
  - Store encrypted master key with password
  - Hardware Security Module (HSM) only
  - Shamir's secret sharing with configurable threshold
- **Decision:** Implement Shamir's secret sharing with 5 shares and 3-of-5 threshold as default
- **Rationale:** Provides cryptographically sound distribution of trust, tolerates loss of 2 shares, requires collaboration of multiple operators, and works without specialized hardware
- **Consequences:** Enables true zero-trust master key storage but requires coordination of multiple operators during unsealing operations

Shamir Parameter	Type	Description	Default Value
Total Shares	int	Total number of key shares generated during initialization	5
Threshold	int	Minimum number of shares required to reconstruct master key	3
Share Length	int	Length of each share in bytes	33 (1 byte x-coordinate + 32 bytes y-coordinate)
Prime Field	big.Int	Prime number defining the finite field for polynomial arithmetic	$2^{256} - 189$
Share Format	string	Encoding format for human-readable shares	Base64 with checksums

The share generation process follows these mathematical steps during system initialization:

1. **Generate Random Coefficients:** Create `threshold - 1` random coefficients for polynomial of degree `threshold - 1`, ensuring the constant term equals the master key
2. **Evaluate Polynomial:** For each share `i` from 1 to `total_shares`, evaluate the polynomial `P(i)` to get the y-coordinate of share `i`
3. **Encode Shares:** Format each share as `(x=i, y=P(i))` pair with checksums and base64 encoding for human handling
4. **Verify Reconstruction:** Test that any `threshold` subset of shares correctly reconstructs the original master key
5. **Secure Distribution:** Present shares to operators through secure channels, ensuring no single operator receives multiple shares

The reconstruction process reverses this during unsealing operations:

1. **Collect Threshold Shares:** Gather at least `threshold` valid shares from operators through authenticated input channels
2. **Parse and Validate:** Decode base64 shares, verify checksums, and extract  $(x, y)$  coordinate pairs for each share
3. **Lagrange Interpolation:** Use Lagrange interpolation formula to reconstruct polynomial coefficients from the provided points
4. **Extract Secret:** Evaluate reconstructed polynomial at  $x=0$  to recover the master key (constant term)
5. **Validate Reconstruction:** Verify reconstructed key matches expected format and can decrypt a test data encryption key

The `ShamirManager` handles the cryptographic operations while maintaining security invariants:

ShamirManager Method	Parameters	Returns	Description
GenerateShares	masterKey []byte, threshold int, totalShares int	([]Share, error)	Splits master key into shares using polynomial over finite field
ReconstructSecret	shares []Share	([]byte, error)	Reconstructs master key from threshold number of valid shares
ValidateShare	share Share	bool	Verifies share format, checksum, and coordinate validity
SecureInput	prompt string	(Share, error)	Collects share from operator with secure terminal input (no echo)
FormatShare	x int, y []byte	string	Encodes coordinate pair as base64 string with checksum
ParseShare	shareData string	(Share, error)	Decodes and validates share from base64 string representation

Share security depends on several critical implementation details. Each share must include a cryptographic checksum to prevent typos or corruption during manual entry. The finite field arithmetic must use constant-time operations to prevent timing attacks during reconstruction. Share storage locations should be physically and logically separated - storing all shares on the same backup system defeats the security purpose.

The critical security insight is that Shamir's scheme provides information-theoretic security: even with unlimited computational power, fewer than threshold shares reveal zero information about the secret. This is stronger than computational security based on hard mathematical problems.

#### Common Pitfalls in Shamir Implementation:

**⚠ Pitfall: Storing Multiple Shares Together** Many implementations defeat Shamir's security by storing multiple shares in the same location (backup system, configuration management, etc.). This creates a single point of compromise that can gather enough shares to reconstruct the master key. Instead, ensure shares are distributed to different operators, stored in separate physical locations, and never aggregated in any single system or backup.

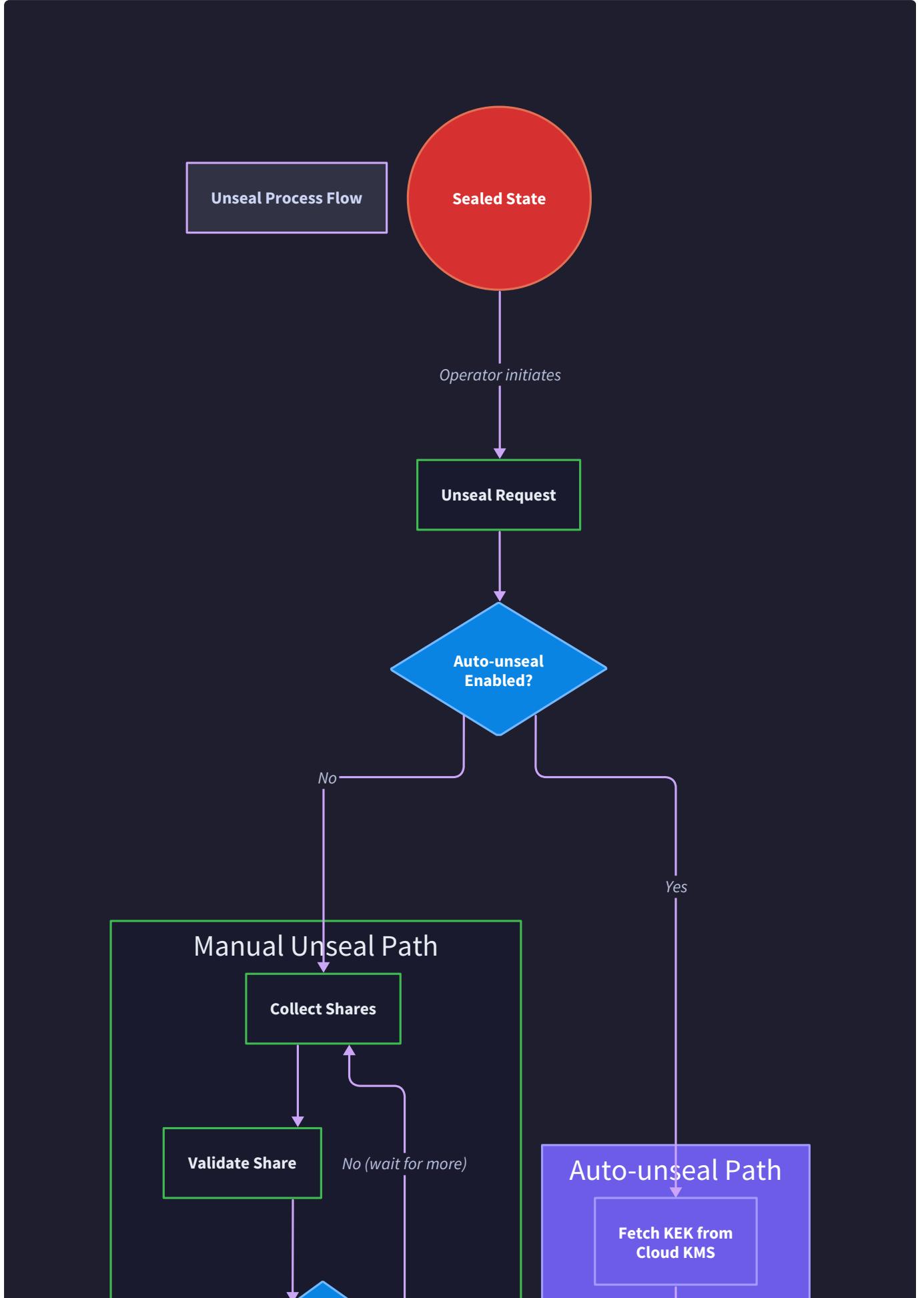
**⚠ Pitfall: Insufficient Randomness in Polynomial Coefficients** Using weak random number generation for polynomial coefficients can make the scheme vulnerable to attacks. The coefficients must be cryptographically random and uniformly distributed over the finite field. Use `/dev/urandom` or equivalent cryptographic RNG, never `math/rand` or similar pseudo-random generators.

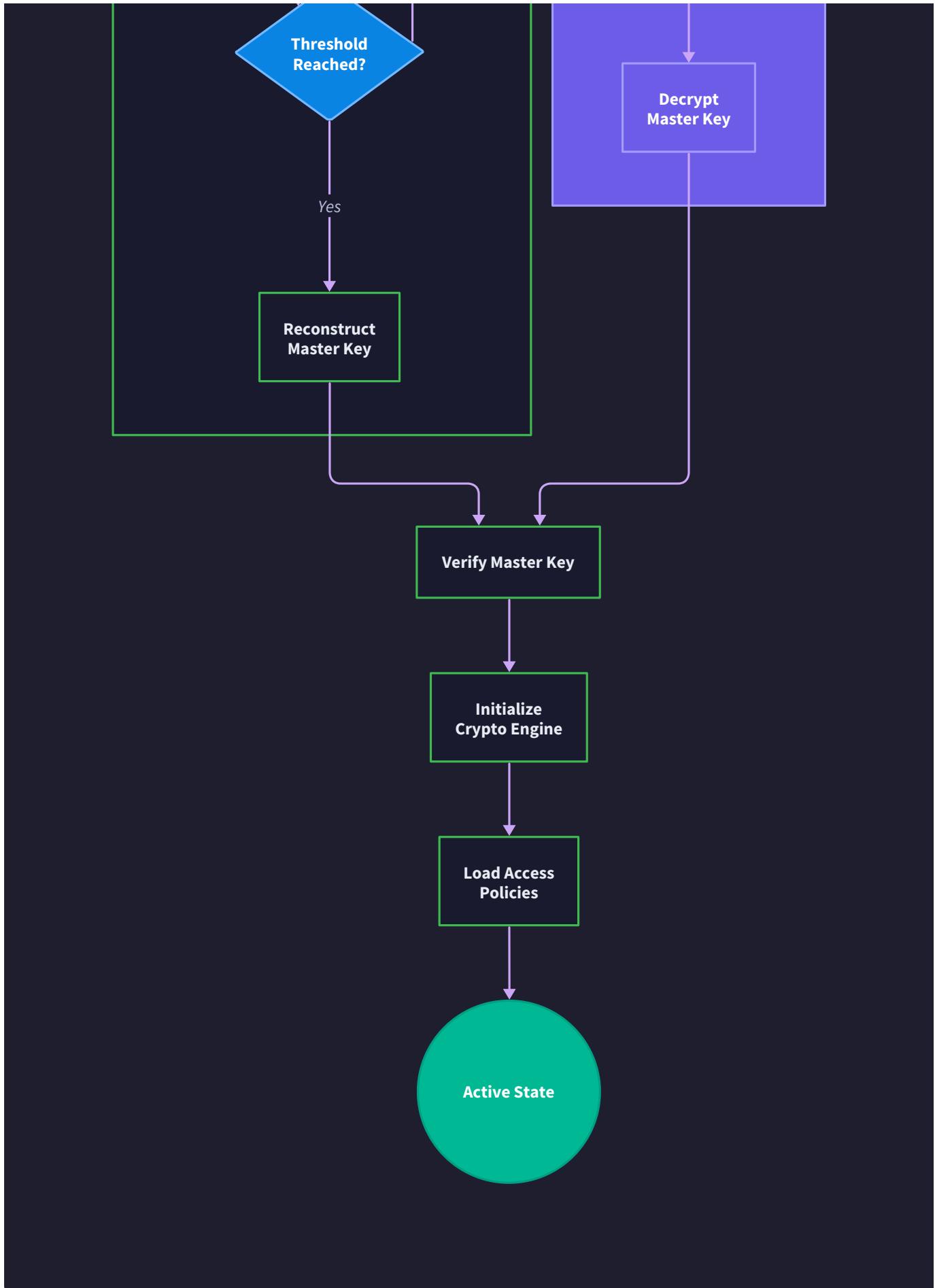
**⚠ Pitfall: Polynomial Degree Mismatch** A polynomial of degree `t-1` requires exactly `t` points for unique reconstruction. Using degree `t` requires `t+1` points, which changes the threshold unexpectedly. Ensure the polynomial degree equals `threshold - 1` and that exactly `threshold` shares are required for reconstruction.

**⚠ Pitfall: Memory Persistence of Reconstructed Key** After reconstructing the master key from shares, the key must be cleared from memory when no longer needed. Go's garbage collector cannot guarantee memory clearing, so use explicit zeroing of byte slices and consider using `mlock()` to prevent swapping to disk during reconstruction operations.

## Seal and Unseal Operations: System Startup and Key Reconstruction Process

The seal and unseal mechanism implements a security state machine that protects the system's cryptographic capabilities when not in active use. Think of this like a bank vault that automatically locks itself during non-business hours and requires multiple officers to unlock it each morning - the vault provides maximum security by default and only becomes accessible through deliberate, authenticated actions.





When sealed, our secret management system enters a hardened state where all cryptographic operations are

disabled, the master key is absent from memory, and only basic administrative operations remain available. This provides defense against a wide range of attack scenarios: memory dumps cannot reveal the master key,

compromised processes cannot access secrets, and even insider threats require coordination with multiple other operators.

The unsealing process transforms the system from this secure but non-functional state into full operational capability by reconstructing the master key from operator-provided shares and loading the cryptographic engines. This state transition must be carefully orchestrated to maintain security properties while providing operational convenience.

### Decision: Sealed-by-Default Security Model

- **Context:** System must protect secrets even when storage, memory, or processes are compromised
- **Options Considered:**
  - Always-available model with encrypted master key
  - Manual seal/unseal with Shamir's shares
  - Automatic unseal with external key management service
- **Decision:** Implement sealed-by-default with both manual and automatic unseal options
- **Rationale:** Provides maximum security against storage compromise, memory extraction, and unauthorized restarts while allowing operational flexibility
- **Consequences:** Requires operator intervention after restarts but provides strongest security posture and compliance benefits

The `UnsealManager` coordinates the state transitions and maintains security invariants:

UnsealManager Field	Type	Description
isSealed	bool	Current seal state - true blocks all secret operations
sealMutex	sync.RWMutex	Protects state transitions and concurrent operation checks
masterKey	[]byte	Reconstructed master key (nil when sealed)
shamirConfig	ShamirConfig	Threshold and total share configuration from initialization
collectedShares	map[int]Share	Shares collected during current unseal attempt
autoUnsealConfig	*AutoUnsealConfig	Configuration for cloud KMS auto-unseal (optional)
sealTime	time.Time	Timestamp when system was last sealed
unsealProgress	UnsealProgress	Tracks share collection progress for monitoring

The seal operation immediately transitions the system to a secure state through these steps:

1. **Acquire Write Lock:** Take exclusive lock on seal mutex to prevent concurrent operations during state transition

2. **Stop Secret Operations:** Set sealed flag to true, causing all secret API endpoints to return "sealed" errors immediately
3. **Clear Cryptographic Material:** Zero the master key byte slice in memory and set pointer to nil
4. **Shutdown Engines:** Stop encryption, authentication, and dynamic secret engines, clearing any cached keys or credentials
5. **Persist Seal State:** Write seal state marker to storage to maintain sealed state across restarts
6. **Log Security Event:** Record seal operation in audit log with timestamp, operator identity, and reason

The unseal operation reverses this process but requires careful validation at each step:

1. **Validate Unseal Preconditions:** Check that system is currently sealed, required shares have been collected, and no conflicting operations are in progress
2. **Reconstruct Master Key:** Use Shamir's reconstruction algorithm on collected shares to recover the master key, validating mathematical consistency
3. **Verify Master Key:** Test reconstructed key by attempting to decrypt a known data encryption key, ensuring the reconstruction was successful
4. **Initialize Engines:** Start encryption engine with reconstructed master key, load data encryption keys, and prepare cryptographic operations
5. **Clear Collected Shares:** Zero share data from memory to prevent future reconstruction attempts using same shares
6. **Update System State:** Set sealed flag to false and remove seal state marker from persistent storage
7. **Log Security Event:** Record successful unseal in audit log with participating operator count and timestamp

Unseal State	Description	Allowed Operations	Next States
Sealed	No master key in memory, all secret operations blocked	Health checks, unseal share submission, status queries	Unsealing, AutoUnsealing
Unsealing	Collecting shares for manual unseal	Share submission, unseal status, cancel unseal	Sealed, Active
AutoUnsealing	Attempting automatic unseal with external KMS	Status queries, health checks	Sealed, Active
Active	Master key available, all operations functional	All secret operations, seal command	Sealed

The share collection process during unsealing provides feedback to operators while maintaining security:

UnsealProgress Field	Type	Description
SharesRequired	int	Total number of shares needed (threshold value)
SharesCollected	int	Number of valid shares received so far
SharesRemaining	int	Additional shares needed to complete unsealing
CollectedShareIDs	[]int	X-coordinates of shares collected (not the share values)
LastShareTime	time.Time	When most recent share was submitted
UnsealStartTime	time.Time	When first share collection began

The unseal API provides a stateful interface for operators to submit shares incrementally:

```
POST /sys/unseal
{
  "share": "base64-encoded-share-with-checksum",
  "reset": false // optional: reset collected shares and start over
}

Response:
{
  "sealed": true/false,
  "progress": {
    "shares_required": 3,
    "shares_collected": 1,
    "shares_remaining": 2
  },
  "unseal_time": "2024-01-15T10:30:45Z" // only present when unsealed
}
```

### Error Handling During Unseal Operations:

The unseal process must handle various failure scenarios gracefully while maintaining security properties:

Failure Mode	Detection Method	Recovery Action	Security Impact
Invalid Share Format	Base64 decode failure, checksum mismatch	Reject share, maintain collection state	None - invalid input ignored
Duplicate Share	X-coordinate matches previously collected share	Reject duplicate, continue collection	None - prevents operator confusion
Insufficient Shares	Reconstruction attempted with < threshold shares	Return error, maintain collection state	None - mathematical impossibility
Reconstruction Failure	Lagrange interpolation produces invalid key	Clear all shares, return to sealed state	High - indicates tampering or corruption
Key Validation Failure	Reconstructed key cannot decrypt test DEK	Clear shares, seal system, alert	High - indicates key corruption or attack

The unsealing process represents a critical security boundary. Once shares are collected and the master key is reconstructed, the system transitions from maximum security (sealed) to operational capability (unsealed). This transition must be atomic and auditable.

#### Common Pitfalls in Seal/Unseal Implementation:

**⚠ Pitfall: Race Conditions During State Transitions** Concurrent seal and unseal operations, or secret requests during state transitions, can lead to inconsistent state or security vulnerabilities. Always use proper locking (read locks for checking state, write locks for changing state) and ensure all secret operations check the sealed state under lock protection.

**⚠ Pitfall: Persistent Share Storage** Some implementations cache collected shares to disk "for convenience" during multi-step unseal operations. This completely undermines Shamir's security by creating a single point where multiple shares can be compromised. Shares must only exist in memory during the unseal process and be immediately zeroed afterward.

**⚠ Pitfall: Insufficient Master Key Validation** After reconstructing the master key from shares, some implementations skip validation and assume the reconstruction was correct. This can lead to accepting corrupted or malicious keys. Always validate the reconstructed key by testing it against known encrypted data before transitioning to unsealed state.

**⚠ Pitfall: Seal State Inconsistency** The seal state must be consistent across memory flags, persistent storage markers, and actual cryptographic engine state. Inconsistencies can lead to security bypasses or operational failures. Ensure all seal/unseal operations update all state locations atomically and include recovery logic for partial state updates.

## High Availability and Consensus: Leader Election and Data Replication Strategies

High availability transforms our secret management system from a single-node service into a distributed cluster that continues operating despite node failures, network partitions, and maintenance activities. Think of this like a hospital emergency room with multiple doctors on duty - if one doctor becomes unavailable, the others continue providing care without interruption, and there's a clear protocol for who takes charge during critical situations.

The fundamental challenge in distributed secret management lies in maintaining strong consistency while providing high availability. Unlike eventual consistency systems where temporary disagreement is acceptable, secret management requires that all nodes have identical views of policies, secrets, and leases. A policy change that allows access on one node but not another could create security vulnerabilities or operational confusion.

Our approach implements a **leader-follower architecture** with **Raft consensus** to ensure linearizable consistency across all cluster operations. One node serves as the elected leader and handles all write operations, while follower nodes can serve read requests and automatically promote to leader if the current leader fails. This provides both consistency guarantees and fault tolerance.

### Decision: Raft Consensus for Cluster Coordination

- **Context:** Need strong consistency for security policies while providing high availability and partition tolerance
- **Options Considered:**
  - Multi-master with conflict resolution (eventual consistency)
  - Paxos-based consensus with separate coordination service
  - Raft consensus integrated into secret management nodes
- **Decision:** Implement Raft consensus directly within secret management nodes
- **Rationale:** Raft provides understandable strong consistency, integrates well with our existing architecture, and avoids external dependencies for coordination
- **Consequences:** Enables strongly consistent cluster operations but requires majority of nodes to be available for write operations

The `ClusterManager` handles all distributed coordination and maintains cluster membership:

ClusterManager Field	Type	Description
nodeID	string	Unique identifier for this cluster node
raftNode	*raft.Raft	Raft consensus implementation handling leader election and log replication
transport	raft.Transport	Network transport for inter-node Raft messages
logStore	raft.LogStore	Persistent storage for Raft log entries
stableStore	raft.StableStore	Persistent storage for Raft metadata (current term, voted for)
snapshotter	raft.SnapshotStore	Handles creation and restoration of cluster state snapshots
peerSet	[]string	Current cluster membership configuration
leadershipChan	chan bool	Notifies when this node gains or loses leadership
isLeader	bool	Current leadership status of this node
lastContact	time.Time	Timestamp of last successful contact with cluster leader

The Raft implementation manages three distinct roles that nodes can occupy:

Node Role	Responsibilities	Message Handling	State Transitions
Leader	Processes all write requests, sends heartbeats, replicates log entries	Accepts client requests, sends AppendEntries to followers	Can become follower on higher term or loss of majority
Follower	Serves read requests, forwards writes to leader, participates in elections	Responds to AppendEntries and RequestVote messages	Can become candidate on leader timeout
Candidate	Requests votes during leader election	Sends RequestVote, counts responses	Becomes leader with majority votes or follower with higher term

### Leader Election Process:

The leader election algorithm ensures exactly one leader exists at any time while handling various failure scenarios:

- 1. Detect Leader Failure:** Followers monitor heartbeat messages from leader, starting election timer if no contact within election timeout (150-300ms randomized)

2. **Become Candidate:** Node increments term number, votes for itself, transitions to candidate role, and resets election timer with random component
3. **Request Votes:** Send RequestVote messages to all other cluster members containing candidate term, node ID, and last log entry information
4. **Evaluate Vote Responses:** Collect votes from other nodes, checking that response term matches request term and vote is granted
5. **Achieve Majority:** If candidate receives votes from majority of nodes (including self), transition to leader role and begin sending heartbeats
6. **Handle Split Vote:** If no candidate achieves majority before timeout, increment term and restart election with new randomized timeout

### **Data Replication and Log Management:**

All cluster state changes flow through the Raft log to ensure consistent ordering and durability across nodes:

Raft Log Entry Field	Type	Description
Index	uint64	Sequential position in the Raft log (monotonically increasing)
Term	uint64	Leader term when entry was created (for detecting stale leaders)
Type	LogEntryType	Type of operation (SecretCreate, PolicyUpdate, TokenRevoke, etc.)
Data	[]byte	Serialized operation data containing all information needed to apply change
Timestamp	time.Time	When leader created this log entry
ClientID	string	Identifier of client that initiated this operation (for deduplication)
Checksum	[]byte	Integrity check for log entry corruption detection

The log replication process ensures all nodes maintain identical state:

1. **Receive Client Request:** Leader validates request, assigns unique log index, and creates log entry with current term
2. **Append to Local Log:** Leader writes entry to local Raft log with fsync for durability before proceeding
3. **Replicate to Followers:** Send AppendEntries messages containing new log entry to all follower nodes
4. **Wait for Majority:** Leader waits for successful acknowledgments from majority of cluster (including self) before committing
5. **Apply to State Machine:** Once majority confirms, leader applies operation to local state machine and marks entry as committed
6. **Notify Client:** Return success response to client only after operation is committed and applied

**7. Propagate Commit:** Next AppendEntries messages inform followers that entry is committed, triggering application to their state machines

#### **Snapshot Management for Log Compaction:**

Raft logs grow continuously and must be periodically compacted to prevent unbounded storage growth:

Snapshot Component	Description	Contents
State Machine Image	Complete secret management state at specific log index	All secrets, policies, tokens, leases, and dynamic secret backends
Last Included Index	Log index of last entry included in snapshot	Used to determine which log entries can be safely deleted
Last Included Term	Term of last entry included in snapshot	Required for maintaining Raft invariants during snapshot installation
Cluster Configuration	Membership and peer information	Node IDs and addresses for cluster reconfiguration
Encryption Context	Current key versions and rotation state	Necessary for continuing encryption operations after restore

#### **High Availability Deployment Topology:**

A production cluster typically consists of 3 or 5 nodes deployed across multiple failure domains:

Cluster Size	Fault Tolerance	Split-Brain Protection	Performance Characteristics
3 nodes	1 node failure	Requires 2/3 majority	Good performance, minimal resource usage
5 nodes	2 node failures	Requires 3/5 majority	Higher fault tolerance, more replication overhead
7+ nodes	3+ node failures	Requires $(n/2)+1$ majority	Maximum fault tolerance, significant replication cost

The cluster handles read and write operations differently to balance consistency and availability:

Operation Type	Leader Behavior	Follower Behavior	Consistency Guarantee
Write Operations	Process locally, replicate via Raft, respond after majority commit	Forward to leader, return leader response	Linearizable (strongest)
Read Operations	Serve immediately from local state	Serve from local state with staleness bounds	Eventually consistent by default
Consistent Reads	Process locally after confirming leadership	Forward to leader for consistency guarantee	Linearizable when requested

### Network Partition Handling:

Network partitions represent one of the most challenging failure modes for distributed systems:

Partition Scenario	Majority Partition Behavior	Minority Partition Behavior	Recovery Process
Clean Split	Continue serving requests normally	Enter read-only mode, reject writes	Rejoin cluster, sync from majority
Flapping Network	May experience leadership churn	Cannot form stable quorum	Use jitter and backoff in elections
Isolated Leader	Step down if cannot contact majority	Elect new leader from remaining nodes	Former leader rejoins as follower

The key insight for partition tolerance is that Raft chooses consistency over availability - a minority partition becomes read-only rather than risk split-brain scenarios that could compromise security policies.

### Common Pitfalls in High Availability Implementation:

**⚠ Pitfall: Split-Brain Scenarios** Allowing multiple nodes to accept write operations simultaneously can lead to conflicting policies, duplicated secrets, or security vulnerabilities. Always ensure only the elected Raft leader processes writes, and that leadership changes require majority consensus. Use fencing mechanisms to prevent former leaders from processing requests after losing leadership.

**⚠ Pitfall: Unbounded Log Growth** Raft logs grow continuously and can consume all available disk space if not properly managed. Implement regular snapshotting with configurable thresholds (e.g., every 10,000 entries or 100MB), and ensure old log entries are safely deleted after successful snapshot creation.

**⚠ Pitfall: Clock Synchronization Dependencies** While Raft doesn't require synchronized clocks for correctness, significant clock skew can cause operational issues like incorrect lease expiration or audit log ordering. Use NTP or similar time synchronization, and design timeout values to be robust against reasonable clock drift ( $\pm 100\text{ms}$  typically).

**⚠ Pitfall: Inadequate Failure Detection** Setting heartbeat intervals too long delays failure detection and increases unavailability windows, while too short intervals create unnecessary network traffic and false positives. Use randomized election timeouts (150-300ms) and heartbeat intervals around 50ms for good balance between responsiveness and stability.

## Auto-Unseal Integration: Cloud KMS Integration for Automated Unsealing

Auto-unseal integration addresses the operational challenge of manual unsealing in production environments while maintaining security properties through external key management services. Think of this like a bank vault that can automatically unlock using a secure communication channel with the bank's central security office - it provides operational convenience without compromising the fundamental security model of keeping the master key protected.

Traditional manual unsealing requires human operators to input Shamir shares after every system restart, which creates operational burden and potential availability issues. Auto-unseal solves this by encrypting the master key with a key encryption key (KEK) managed by an external, highly available key management service. The system can then automatically retrieve and use the KEK to decrypt its master key during startup, eliminating the need for human intervention while maintaining the security property that the master key is never stored in plaintext.

The critical security insight is that auto-unseal shifts the trust boundary rather than eliminating it - instead of trusting human operators with Shamir shares, we trust the external KMS provider with the key encryption key. This trade-off makes sense in cloud environments where the KMS provider offers hardware security modules, audit logging, and fine-grained access controls that may exceed what we can achieve with manual processes.

### Decision: Hybrid Manual and Auto-Unseal Support

- **Context:** Production environments need automated recovery while maintaining option for air-gapped or manual control scenarios
- **Options Considered:**
  - Manual Shamir shares only (maximum security, operational burden)
  - Auto-unseal only (operational convenience, external dependencies)
  - Hybrid approach supporting both methods
- **Decision:** Implement both manual and auto-unseal with configuration-driven selection
- **Rationale:** Provides flexibility for different deployment scenarios, allows migration between methods, and maintains compatibility with security requirements that mandate manual control
- **Consequences:** Increases implementation complexity but provides maximum deployment flexibility and migration paths

The `AutoUnsealManager` handles integration with external key management services:

<b>AutoUnsealManager Field</b>	<b>Type</b>	<b>Description</b>
provider	KMSProvider	Interface to specific KMS service (AWS KMS, Azure KeyVault, etc.)
keyID	string	External KMS key identifier for master key encryption
encryptedMasterKey	[]byte	Master key encrypted with KMS key, stored locally
authConfig	KMSAuthConfig	Authentication configuration for KMS service access
retryConfig	RetryConfig	Backoff and timeout settings for KMS operations
healthChecker	*KMSHealthChecker	Monitors KMS service availability and key accessibility
lastUnsealTime	time.Time	Timestamp of most recent successful auto-unseal
failureCount	int64	Count of consecutive auto-unseal failures for alerting

### Key Management Service Provider Interface:

The KMS provider abstraction allows integration with multiple external key management services:

<b>KMSProvider Method</b>	<b>Parameters</b>	<b>Returns</b>	<b>Description</b>
Encrypt	plaintext []byte, keyID string	([]byte, error)	Encrypts data using specified KMS key
Decrypt	ciphertext []byte, keyID string	([]byte, error)	Decrypts data using specified KMS key
GenerateDataKey	keyID string, keySize int	(plaintext []byte, encrypted []byte, error)	Generates new data encryption key
DescribeKey	keyID string	(*KeyMetadata, error)	Retrieves key information and access policies
ListKeys	maxResults int	([]string, error)	Lists available keys for key discovery
HealthCheck	keyID string	error	Verifies key exists and is accessible

### AWS KMS Integration Implementation:

AWS KMS represents the most common auto-unseal scenario and demonstrates the integration pattern:

AWS KMS Configuration	Type	Description	Example Value
Region	string	AWS region containing the KMS key	us-west-2
KeyID	string	KMS key identifier (ARN, alias, or key ID)	alias/vault-unseal-key
AccessKeyId	string	AWS access key for API authentication	AKIAIOSFODNN7EXAMPLE
SecretAccessKey	string	AWS secret key (should use IAM roles instead)	wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
SessionToken	string	AWS session token for temporary credentials	Optional for STS credentials
Endpoint	string	Custom KMS endpoint for testing or regions	<a href="https://kms.us-west-2.amazonaws.com">https://kms.us-west-2.amazonaws.com</a>

The AWS KMS provider implements the interface using the AWS SDK:

```
type AWSKMSProvider struct {
    client    *kms.KMS
    region   string
    keyID    string
    retries  int
    timeout  time.Duration
}
```

GO

### Auto-Unseal Process Flow:

The automated unsealing process combines local cryptographic operations with external KMS calls:

- 1. Initialize KMS Client:** Configure authentication credentials, endpoint URLs, and retry policies for the external KMS service
- 2. Retrieve Encrypted Master Key:** Load the locally stored encrypted master key from persistent storage (created during initial seal with auto-unseal)
- 3. Authenticate with KMS:** Present credentials to KMS service and verify access to the configured key encryption key

4. **Decrypt Master Key:** Call KMS decrypt operation with encrypted master key, receiving plaintext master key in response
5. **Validate Decrypted Key:** Verify decrypted master key has correct format and can successfully decrypt a test data encryption key
6. **Initialize Cryptographic Engines:** Start encryption, authentication, and dynamic secret engines using the decrypted master key
7. **Clear Temporary Data:** Zero the plaintext master key from local variables while leaving it in the encryption engine
8. **Update System State:** Transition from sealed to unsealed state and begin serving secret management requests

#### **Failure Handling and Degradation:**

Auto-unseal operations can fail due to network issues, authentication problems, or KMS service outages:

Failure Type	Symptoms	Recovery Strategy	Fallback Options
Network Connectivity	Timeout errors, connection refused	Exponential backoff retry with circuit breaker	Manual unseal if operators available
Authentication Failure	401/403 errors from KMS service	Credential refresh, IAM role validation	Alternative credentials or manual unseal
Key Not Found	404 errors, key disabled/deleted	Alert operators, check key status	Restore from backup or manual unseal
Rate Limiting	429 errors, quota exceeded	Implement backoff and jitter	Queue unseal requests or manual unseal
KMS Service Outage	500+ errors, service unavailable	Extended retry with exponential backoff	Manual unseal to maintain availability

The retry mechanism implements exponential backoff with jitter to avoid thundering herd problems:

Retry Attempt	Base Delay	Jitter Range	Max Delay	Total Elapsed
1	1 second	±200ms	1.2s	1.2s
2	2 seconds	±400ms	2.4s	3.6s
3	4 seconds	±800ms	4.8s	8.4s
4	8 seconds	±1.6s	9.6s	18s
5+	30 seconds	±6s	36s	54s+

#### **Security Considerations for Auto-Unseal:**

Auto-unseal introduces new attack vectors that must be carefully managed:

Security Risk	Description	Mitigation Strategy
KMS Key Compromise	External KMS key is compromised or misused	Use KMS key policies, audit logging, key rotation
Credential Exposure	AWS keys or similar credentials leaked	Use IAM roles, temporary credentials, credential rotation
Network Interception	KMS API calls intercepted or modified	Use TLS, certificate pinning, request signing
Replay Attacks	Old decrypt requests replayed to bypass controls	Use nonces, timestamps, and request uniqueness
Privilege Escalation	Overly broad KMS permissions exploited	Principle of least privilege, specific key access only

The KMS key policy should restrict access to only the secret management service:

---

{

JSON

```

    "Version": "2012-10-17",

    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {"AWS": "arn:aws:iam::ACCOUNT:role/vault-unseal-role"},
            "Action": ["kms:Decrypt", "kms:DescribeKey"],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "kms.REGION.amazonaws.com"
                }
            }
        }
    ]
}

```

## Monitoring and Observability:

Auto-unseal operations require comprehensive monitoring to detect failures and security issues:

Metric Category	Key Metrics	Alerting Thresholds
Success Rate	Successful unseals / Total attempts	< 95% success rate
Latency	Time from unseal start to completion	> 30 seconds
Error Patterns	KMS error types and frequencies	> 5 auth failures/hour
Credential Health	Time until credential expiration	< 24 hours remaining
Key Accessibility	KMS key describe success rate	Any failures

## Common Pitfalls in Auto-Unseal Implementation:

**⚠️ Pitfall: Storing KMS Credentials in Configuration** Hard-coding AWS access keys or similar credentials in configuration files creates a security vulnerability that defeats the purpose of external key management. Always

use IAM roles, instance profiles, or other credential-less authentication methods when possible.

**⚠ Pitfall: Insufficient KMS Error Handling** KMS services can return various error types (throttling, authentication, key not found) that require different handling strategies. Implement specific retry logic for retriable errors while immediately failing for authentication issues that require operator intervention.

**⚠ Pitfall: Missing Fallback to Manual Unseal** If auto-unseal fails and no manual unseal option is available, the system becomes completely inaccessible until the external dependency is resolved. Always maintain the ability to collect Shamir shares manually as a fallback recovery mechanism.

#### **⚠ Pitfall: Overly Broad KMS Permissions**

Granting broad KMS permissions like `kms : *` or access to all keys increases the blast radius if credentials are compromised. Use fine-grained permissions that allow access only to the specific key needed for unsealing operations.

## **Implementation Guidance**

The unsealing and high availability components represent the most complex part of our secret management system, involving distributed systems concepts, cryptographic operations, and external service integration. This implementation guidance provides the foundation and structure needed to build these components incrementally.

### **Technology Recommendations:**

Component	Simple Option	Advanced Option
Shamir's Secret Sharing	Custom implementation with math/big	Hashicorp go-shamir library
Raft Consensus	Hashicorp raft library	etcd raft implementation
KMS Integration	AWS SDK with simple retry	Multi-cloud abstraction layer
Network Transport	HTTP/1.1 with connection pooling	gRPC with load balancing
State Storage	BoltDB embedded database	Distributed storage backend

### **Recommended File Structure:**

```
internal/
  unseal/
    shamir.go          ← Shamir's secret sharing implementation
    shamir_test.go     ← Cryptographic correctness tests
    manager.go         ← Seal/unseal state machine
    manager_test.go   ← State transition tests
  cluster/
    raft.go           ← Raft consensus integration
    raft_test.go      ← Leader election and replication tests
    transport.go      ← Network transport for Raft messages
    snapshot.go       ← State machine snapshot handling
  kms/
    interface.go      ← KMS provider interface definition
    aws.go            ← AWS KMS implementation
    azure.go          ← Azure Key Vault implementation
    mock.go           ← Mock provider for testing
    autounseal.go     ← Auto-unseal coordinator
server/
  cluster_server.go  ← HTTP handlers for cluster operations
  unseal_handlers.go ← API endpoints for seal/unseal
cmd/
  vault-server/
    main.go          ← Entry point with cluster startup
config/
  cluster-example.yaml ← Example cluster configuration
```

## Shamir's Secret Sharing Foundation:

```
package unseal

import (
    "crypto/rand"
    "errors"
    "fmt"
    "math/big"
)

// ShamirConfig defines parameters for secret sharing

type ShamirConfig struct {
    Threshold    int      // Minimum shares required for reconstruction
    TotalShares  int      // Total number of shares to generate
    PrimeField   *big.Int // Prime number defining finite field
}

// Share represents a single Shamir share with coordinates

type Share struct {
    X          int      // X-coordinate (share number)
    Y          []byte   // Y-coordinate (share value)
    Checksum  []byte   // Integrity check for the share
}

// ShamirManager handles secret sharing operations

type ShamirManager struct {
    config ShamirConfig
}

// NewShamirManager creates a new manager with given configuration
```

GO

```

func NewShamirManager(threshold, totalShares int) (*ShamirManager, error) {

    // TODO 1: Validate threshold <= totalShares and both > 0

    // TODO 2: Generate or load appropriate prime field for arithmetic

    // TODO 3: Initialize ShamirManager with validated configuration

    // Hint: Use a 256-bit prime like 2^256 - 189 for the field

}

// GenerateShares splits secret into threshold shares

func (sm *ShamirManager) GenerateShares(secret []byte) ([]Share, error) {

    // TODO 1: Convert secret bytes to big integer for polynomial arithmetic

    // TODO 2: Generate threshold-1 random coefficients for polynomial

    // TODO 3: Evaluate polynomial at x=1,2,...,totalShares to create shares

    // TODO 4: Add checksums to each share for integrity validation

    // TODO 5: Test reconstruction with threshold shares before returning

    // Hint: Polynomial is P(x) = secret + c1*x + c2*x^2 + ... + c(t-1)*x^(t-1)

}

// ReconstructSecret rebuilds secret from threshold shares

func (sm *ShamirManager) ReconstructSecret(shares []Share) ([]byte, error) {

    // TODO 1: Validate we have at least threshold shares

    // TODO 2: Verify checksums on all provided shares

    // TODO 3: Use Lagrange interpolation to reconstruct polynomial coefficients

    // TODO 4: Evaluate polynomial at x=0 to get secret (constant term)

    // TODO 5: Convert big integer result back to byte slice

    // Hint: Lagrange formula: sum over i of (y_i * product over j≠i of (0-x_j)/(x_i-x_j))

}

```

### Seal/Unseal State Machine:

```
package unseal

import (
    "sync"
    "time"
)

// UnsealManager coordinates system seal/unseal operations

type UnsealManager struct {

    isSealed        bool
    sealMutex       sync.RWMutex
    masterKey       []byte
    shamirManager   *ShamirManager
    collectedShares map[int]Share
    autoUnseal      *AutoUnsealManager

    // State tracking
    sealTime        time.Time
    unsealProgress  UnsealProgress
}

// UnsealProgress tracks share collection during unsealing

type UnsealProgress struct {

    SharesRequired  int          `json:"shares_required"`
    SharesCollected int          `json:"shares_collected"`
    SharesRemaining int          `json:"shares_remaining"`
    CollectedShareIDs []int      `json:"collected_share_ids"`
    LastShareTime   time.Time   `json:"last_share_time"`
}
```

GO

```
        UnsealStartTime time.Time `json:"unseal_start_time"`

    }

// NewUnsealManager creates manager with given Shamir configuration

func NewUnsealManager(threshold, totalShares int) (*UnsealManager, error) {

    // TODO 1: Create ShamirManager with provided parameters

    // TODO 2: Initialize UnsealManager in sealed state

    // TODO 3: Set up empty share collection map

    // TODO 4: Initialize progress tracking structures

}

// Seal immediately transitions system to sealed state

func (um *UnsealManager) Seal() error {

    // TODO 1: Acquire write lock to prevent concurrent operations

    // TODO 2: Set sealed flag to true to block secret operations

    // TODO 3: Zero and nil the master key to clear from memory

    // TODO 4: Clear any collected shares from memory

    // TODO 5: Record seal time and log security event

    // Hint: Use explicit zeroing like for i := range um.masterKey { um.masterKey[i] = 0 }

}

// SubmitShare adds a share to the unsealing process

func (um *UnsealManager) SubmitShare(shareData string) (*UnsealProgress, error) {

    // TODO 1: Parse and validate the provided share data

    // TODO 2: Check if share X-coordinate is already collected (reject duplicates)

    // TODO 3: Add share to collection and update progress tracking

    // TODO 4: If threshold reached, attempt reconstruction and unsealing

    // TODO 5: Return current progress status regardless of completion
```

```
// Hint: Always return progress to show operator current state
}

// IsSealed returns current seal status (thread-safe)

func (um *UnsealManager) IsSealed() bool {
    // TODO 1: Acquire read lock to safely check state
    // TODO 2: Return current sealed flag value
    // Hint: Use RLock/RUnlock for concurrent read access
}
```

### Raft Cluster Integration:

```
package cluster

import (
    "time"

    "github.com/hashicorp/raft"
)

// ClusterManager handles distributed consensus and replication

type ClusterManager struct {

    nodeID      string

    raftNode    *raft.Raft

    transport   raft.Transport

    logStore    raft.LogStore

    stableStore raft.StableStore

    snapshots   raft.SnapshotStore

    // Leadership tracking

    isLeader     bool

    leadershipChan chan bool

    lastContact  time.Time

}

// ClusterConfig defines cluster membership and networking

type ClusterConfig struct {

    NodeID      string `yaml:"node_id"`

    BindAddress string `yaml:"bind_address"`

    DataDir     string `yaml:"data_dir"`

    Peers       []string `yaml:"peers"`
}
```

GO

```
// Raft timing parameters

HeartbeatTimeout    time.Duration `yaml:"heartbeat_timeout"`
ElectionTimeout    time.Duration `yaml:"election_timeout"`
CommitTimeout      time.Duration `yaml:"commit_timeout"`
LeaderLeaseTimeout time.Duration `yaml:"leader_lease_timeout"`

}

// NewClusterManager initializes Raft consensus for the node

func NewClusterManager(config ClusterConfig) (*ClusterManager, error) {

    // TODO 1: Create BoltDB stores for Raft log and stable storage

    // TODO 2: Set up TCP transport for inter-node communication

    // TODO 3: Configure Raft timing parameters from config

    // TODO 4: Initialize Raft node with stores and transport

    // TODO 5: Set up leadership change notification channel

    // Hint: Use raft.Config with reasonable defaults for timing parameters

}

// Start begins Raft consensus participation

func (cm *ClusterManager) Start() error {

    // TODO 1: Bootstrap cluster if this is the first node

    // TODO 2: Join existing cluster if peers are configured

    // TODO 3: Start leadership monitoring goroutine

    // TODO 4: Begin accepting client requests based on leadership status

    // Hint: Check if stable store is empty to determine bootstrap vs join

}

// Apply submits operation to Raft for consensus and replication
```

```
func (cm *ClusterManager) Apply(operation []byte, timeout time.Duration) error {

    // TODO 1: Check if this node is current leader

    // TODO 2: Create Raft log entry with operation data

    // TODO 3: Submit to Raft and wait for majority commitment

    // TODO 4: Return error if operation fails consensus or times out

    // Hint: Only leaders can accept write operations in Raft

}

// IsLeader returns current leadership status

func (cm *ClusterManager) IsLeader() bool {

    // TODO 1: Return cached leadership status

    // TODO 2: Consider checking Raft state for authoritative answer

    // Hint: Cache leadership to avoid expensive Raft state checks

}
```

#### Auto-Unseal with AWS KMS:

```
package kms
```

```
import (
    "context"
    "time"
    "github.com/aws/aws-sdk-go/service/kms"
)
```

```
// KMSProvider defines interface for key management services
```

```
type KMSProvider interface {
```

```
    Encrypt(ctx context.Context, plaintext []byte, keyID string) ([]byte, error)
```

```
    Decrypt(ctx context.Context, ciphertext []byte, keyID string) ([]byte, error)
```

```
    HealthCheck(ctx context.Context, keyID string) error
```

```
}
```

```
// AWSKMSProvider implements KMS interface for AWS
```

```
type AWSKMSProvider struct {
```

```
    client *kms.KMS
```

```
    region string
```

```
    keyID string
```

```
    retries int
```

```
    timeout time.Duration
```

```
}
```

```
// AutoUnsealManager coordinates automatic unsealing with external KMS
```

```
type AutoUnsealManager struct {
```

```
    provider KMSProvider
```

```
    keyID string
```

```
    encryptedMasterKey []byte
```

GO

```
retryConfig      RetryConfig

healthChecker    *KMSHealthChecker

// Monitoring

lastUnsealTime   time.Time

failureCount     int64

}

// NewAWSKMSProvider creates AWS KMS client with configuration

func NewAWSKMSProvider(region, keyID string) (*AWSKMSProvider, error) {

    // TODO 1: Create AWS session with region and credentials

    // TODO 2: Initialize KMS client with session

    // TODO 3: Configure retry behavior and timeouts

    // TODO 4: Validate key exists and is accessible

    // Hint: Use AWS SDK v1 for simpler credential handling

}

// Encrypt encrypts data using AWS KMS key

func (aws *AWSKMSProvider) Encrypt(ctx context.Context, plaintext []byte, keyID string)
([]byte, error) {

    // TODO 1: Create KMS encrypt request with data and key ID

    // TODO 2: Call KMS encrypt API with context for cancellation

    // TODO 3: Handle AWS-specific errors (throttling, auth, etc.)

    // TODO 4: Return ciphertext blob from KMS response

    // Hint: AWS returns base64-encoded ciphertext that includes metadata

}

// Decrypt decrypts data using AWS KMS
```

```

func (aws *AWSKMSProvider) Decrypt(ctx context.Context, ciphertext []byte, keyID string)
([]byte, error) {

    // TODO 1: Create KMS decrypt request with ciphertext blob

    // TODO 2: Call KMS decrypt API (key ID embedded in ciphertext)

    // TODO 3: Implement exponential backoff for retriable errors

    // TODO 4: Return plaintext from successful decrypt response

    // Hint: KMS ciphertext includes key ID, so explicit keyID parameter not needed

}

// AttemptAutoUnseal tries to unseal using KMS-encrypted master key

func (am *AutoUnsealManager) AttemptAutoUnseal(ctx context.Context) error {

    // TODO 1: Load encrypted master key from local storage

    // TODO 2: Call KMS provider decrypt with retry logic

    // TODO 3: Validate decrypted key format and test against known DEK

    // TODO 4: Initialize encryption engines with recovered master key

    // TODO 5: Clear plaintext key from local variables after use

    // Hint: Implement circuit breaker pattern for repeated KMS failures

}

```

### Milestone Checkpoint for Unsealing and High Availability:

After implementing this milestone, your system should demonstrate the following capabilities:

- 1. Shamir Share Generation:** Run `./vault-server generate-shares -threshold=3 -total=5` to create master key shares
- 2. Manual Unsealing:** Start server in sealed mode, submit shares via API until threshold reached and system unseals
- 3. Cluster Formation:** Start multiple nodes that discover each other and elect a leader for write operations
- 4. Auto-Unseal:** Configure AWS KMS integration and verify automatic unsealing after restart
- 5. Failure Recovery:** Stop leader node and verify follower promotion maintains service availability

### Expected test outcomes:

```

# Test Shamir correctness

go test ./internal/unseal -run TestShamirReconstruction

# Should pass with various threshold combinations

# Test seal/unseal state machine

go test ./internal/unseal -run TestSealUnsealCycle

# Should handle concurrent operations correctly

# Test cluster consensus

go test ./internal/cluster -run TestLeaderElection

# Should elect leader and replicate operations

# Test auto-unseal integration

go test ./internal/kms -run TestAutoUnseal

# Should unseal using mocked KMS responses

```

BASH

## Debugging Tips:

Symptom	Likely Cause	Diagnosis	Fix
Shares don't reconstruct original key	Polynomial degree mismatch or field arithmetic error	Check that polynomial degree = threshold-1, verify field operations	Use consistent prime field, validate coefficient generation
System hangs during unseal	Deadlock in seal mutex or blocking KMS call	Check goroutine stacks, verify lock ordering	Use context timeouts, avoid nested locking
Cluster nodes can't find each other	Network configuration or Raft transport issues	Check bind addresses, firewall rules, DNS resolution	Verify peer addresses match actual network interfaces
Frequent leadership changes	Network instability or timing parameter issues	Monitor Raft logs for election triggers, check network latency	Increase election timeout, improve network stability
Auto-unseal fails intermittently	KMS service throttling or credential expiration	Check AWS CloudTrail logs, monitor KMS API responses	Implement exponential backoff, refresh credentials

## Interactions and Data Flow

**Milestone(s):** This section spans all four milestones, demonstrating how the components developed in each milestone interact to process requests. It shows authentication and authorization flows (Milestone 2), secret encryption/decryption operations (Milestone 1), dynamic secret generation (Milestone 3), and how the system operates in sealed/unsealed states (Milestone 4).

Think of the secret management system as a sophisticated corporate bank with multiple security checkpoints, automated vaults, and background maintenance systems. When a client makes a withdrawal request (secret retrieval), their identity gets verified at the front desk (authentication), their access permissions get checked against the account rules (authorization), the secure vault gets unlocked with the right keys (encryption), and background systems handle account maintenance like expiring temporary accounts (lease management). This section maps out all these interaction flows and background processes.

The secret management system operates through three primary interaction patterns: synchronous request/response flows that handle client operations, asynchronous background processes that maintain system health, and internal component communication that coordinates distributed operations. Each pattern has distinct characteristics, error handling requirements, and performance considerations that shape the overall system behavior.

## REST API Design

The REST API serves as the primary interface between clients and the secret management system, following a path-based URL structure that mirrors the internal policy hierarchy. The API design philosophy emphasizes consistency, security, and discoverability while maintaining compatibility with HashiCorp Vault's client ecosystem.

### API Endpoint Structure

The API follows a hierarchical path structure where each segment represents a different system concern. Think of the URL structure like a filing system in a corporate office - the first folder indicates the department (auth, secret, sys), the second folder indicates the specific service or mount point, and subsequent folders represent the actual resources being accessed. This hierarchy enables both human operators and automated tools to understand access patterns and implement appropriate security policies.

Endpoint Category	Path Pattern	Purpose	Authentication Required
Authentication	/v1/auth/{method}/{operation}	Login and token operations	Partial (method-specific)
Secret Operations	/v1/secret/{path}	Static secret CRUD operations	Yes (bearer token)
Dynamic Secrets	/v1/{backend}/{operation}	Dynamic credential generation	Yes (bearer token)
System Operations	/v1/sys/{operation}	Administrative and status operations	Yes (elevated privileges)
Health/Status	/v1/sys/health	System status and readiness checks	No

The authentication endpoints handle the initial identity verification process, accepting different credential types based on the configured authentication methods. Secret operation endpoints manage static secrets with full CRUD capabilities, supporting both key-value operations and metadata management. Dynamic secret endpoints are mounted at configurable paths and provide credential generation specific to each backend type. System operation endpoints handle administrative tasks like policy management, unsealing, and configuration.

## Request and Response Format Standards

All API interactions use JSON for request and response bodies, with consistent error handling and metadata structure across all endpoints. The response format follows a standardized envelope that provides both the requested data and contextual information needed for client decision-making.

### Standard Response Envelope:

Field	Type	Description
request_id	string	Unique identifier for request tracking and audit correlation
lease_id	string	Lease identifier for renewable/revocable responses (dynamic secrets)
renewable	boolean	Whether this response can be renewed before expiration
lease_duration	integer	Seconds until this response expires (0 for non-expiring)
data	object	The actual response payload specific to the operation
warnings	array	Non-fatal issues that clients should be aware of
auth	object	Authentication information for login responses
wrap_info	object	Response wrapping metadata for secure token delivery

## Error Response Format:

Field	Type	Description
errors	array	Human-readable error messages for debugging
error_type	string	Machine-readable error category for client handling
request_id	string	Request identifier for correlation with audit logs

**Design Insight:** The response envelope pattern provides clients with all necessary information to handle credential lifecycle management without requiring additional API calls. The lease metadata enables clients to proactively renew credentials before expiration, while the request ID enables correlation between client operations and server audit logs for security investigations.

## Authentication Header Requirements

The API uses bearer token authentication carried in HTTP headers, following OAuth 2.0 conventions while supporting Vault-specific token features. Think of the authentication header like an electronic keycard that employees use to access different areas of a corporate building - the card identifies the person and contains access permissions that are checked at each secured door.

Header Name	Format	Purpose	Required When
X-Vault-Token	Bearer token string	Primary authentication credential	All authenticated requests
X-Vault-Namespace	Namespace path	Multi-tenancy isolation (future extension)	Multi-tenant deployments
X-Vault-Request-ID	UUID string	Client-provided request correlation	Optional (server generates if missing)
X-Vault-Wrap-TTL	Duration string	Response wrapping TTL request	When requesting wrapped responses

## Token Types and Behavior:

Token Type	Characteristics	Use Case
Service	Long-lived, renewable, bound to policies	Application authentication
Batch	Short-lived, encrypted, lightweight	High-throughput scenarios
Orphan	No parent relationship, manual lifecycle	Root access and emergency recovery

## Decision: Bearer Token Authentication

- **Context:** Need to balance security with client implementation simplicity across diverse application environments
- **Options Considered:**
  - HTTP Basic Auth with username/password
  - JWT tokens with embedded claims
  - Bearer tokens with server-side validation
- **Decision:** Bearer tokens with server-side state and validation
- **Rationale:** Provides immediate revocation capability, supports fine-grained policy evaluation, and enables audit logging with full context
- **Consequences:** Requires server-side token storage and validation, but enables precise access control and immediate security response

## Request Processing Pipeline

The request processing pipeline transforms incoming HTTP requests through multiple security and business logic stages before generating responses. Think of this pipeline like airport security screening - each checkpoint validates different aspects of the passenger (request) and their credentials, with the ability to reject the request at any stage if security requirements aren't met.

The pipeline operates as a series of filters, where each stage can either pass the request to the next stage, return an error response, or modify the request context for downstream processing. This design enables consistent security policy enforcement while maintaining performance through early rejection of invalid requests.

### Stage 1: Request Parsing and Validation

The initial stage handles HTTP-level parsing and basic request validation before any security processing begins. This stage protects the system from malformed requests and provides early feedback for client debugging.

#### Request Parsing Process:

1. **HTTP Method and Path Extraction:** The server extracts the HTTP method and URL path, validating that the method is supported for the requested endpoint. Unsupported methods return HTTP 405 Method Not Allowed with an Allow header indicating supported methods.
2. **Header Validation:** All required headers are validated for presence and format correctness. The `Content-Type` header must be `application/json` for request bodies, and the `Accept` header should include `application/json` for response formatting.
3. **Body Size and Format Validation:** Request bodies are limited to prevent memory exhaustion attacks. JSON parsing occurs with strict validation - malformed JSON immediately returns HTTP 400 Bad Request

with parsing error details.

4. **Path Parameter Extraction:** URL path segments are parsed and validated according to the endpoint's parameter requirements. Special characters are URL-decoded and validated against allowed character sets to prevent injection attacks.
5. **Query Parameter Processing:** Query parameters are parsed and validated according to endpoint specifications. Parameters like `version` for secret operations and `ttl` for dynamic secrets undergo type and range validation.

#### Request Validation Table:

Validation Type	Check Performed	Error Response	Recovery Action
Content-Type	Must be application/json for POST/PUT	415 Unsupported Media Type	Client must set correct header
Body Size	Must be under 1MB limit	413 Payload Too Large	Client must reduce request size
JSON Syntax	Must parse as valid JSON	400 Bad Request with parse error	Client must fix JSON formatting
Required Fields	Must contain all mandatory fields	400 Bad Request with missing fields	Client must include required data
Field Types	Must match expected data types	400 Bad Request with type errors	Client must correct data types

## Stage 2: Authentication and Identity Resolution

The authentication stage verifies client identity and resolves the authenticated entity to internal identity structures. This stage implements the bearer token authentication model while supporting extensibility for additional authentication methods.

#### Authentication Process Flow:

1. **Token Extraction:** The `X-Vault-Token` header is extracted and validated for proper format. Missing tokens result in HTTP 401 Unauthorized for protected endpoints, while malformed tokens return HTTP 400 Bad Request with format guidance.
2. **Token Lookup and Validation:** The token is looked up in the token store using constant-time comparison to prevent timing attacks. Invalid or expired tokens immediately return HTTP 403 Forbidden with appropriate error details.
3. **Token Metadata Resolution:** Valid tokens have their associated metadata loaded, including policies, TTL information, usage constraints, and parent token relationships. This metadata drives subsequent authorization decisions.

4. **Identity Construction:** An `Identity` object is constructed containing all relevant information about the authenticated entity, including resolved policies from direct assignment and group membership.
5. **Usage Tracking and Limits:** Token usage is recorded for audit purposes and usage limits are enforced. Tokens that exceed their maximum usage count are immediately revoked.

#### **Authentication Methods Comparison:**

Method	Token Format	Validation Approach	Performance	Security Level
Token	Random string	Server-side lookup	Fast (memory/cache)	High (immediate revocation)
JWT	Signed JSON	Cryptographic validation	Faster (no lookup)	Medium (revocation lag)
mTLS	X.509 Certificate	Certificate chain validation	Moderate	Very High (crypto proof)

#### **Decision: Server-Side Token Validation**

- **Context:** Need immediate token revocation capability for security incidents while maintaining high request throughput
- **Options Considered:**
  - JWT tokens with embedded claims and expiration
  - Server-side token store with fast lookup
  - Hybrid approach with cached validation
- **Decision:** Server-side token store with memory caching
- **Rationale:** Enables immediate revocation for security response, supports complex usage tracking, and provides full audit context
- **Consequences:** Requires server-side state management but provides superior security control and immediate policy updates

### **Stage 3: Authorization and Policy Evaluation**

The authorization stage determines whether the authenticated identity has permission to perform the requested operation on the specified resource path. This implements the path-based access control system with wildcard matching and parameter validation.

#### **Policy Evaluation Algorithm:**

1. **Path Normalization:** The requested path is normalized by removing trailing slashes, resolving relative references, and converting to canonical form. This prevents authorization bypass through path manipulation.
2. **Policy Collection:** All policies associated with the authenticated identity are collected, including directly assigned policies and policies inherited through group membership. Policy collection follows a breadth-first pattern to ensure complete coverage.

3. **Rule Matching:** Each policy rule is evaluated against the normalized request path using wildcard matching. Rules are processed in order of specificity, with more specific patterns taking precedence over general patterns.
4. **Capability Verification:** The requested operation is checked against the capabilities granted by matching policy rules. Operations must be explicitly allowed - the system implements default deny with explicit allow semantics.
5. **Parameter Validation:** Request parameters are validated against policy constraints including allowed values, required parameters, and denied parameters. This provides fine-grained control over operation characteristics.
6. **TTL and Constraint Application:** Maximum TTL limits and other constraints from matching policies are applied to the request context. These constraints are enforced during secret operations and lease generation.

#### Policy Rule Evaluation Table:

Rule Type	Pattern Example	Matches	Grants	Constraints Applied
Exact Path	secret/myapp/db	Exact path only	Specified capabilities	All policy constraints
Single Wildcard	secret/myapp/*	Direct children only	Specified capabilities	All policy constraints
Recursive Wildcard	secret/myapp/**	All descendants	Specified capabilities	All policy constraints
Parameter Filter	secret/+config	Paths with parameters	Conditional on parameters	Parameter-specific limits

**Critical Security Insight:** The policy evaluation must be performed on every request, even for cached or frequently accessed secrets. Authorization state can change independently of authentication state through policy updates, token revocation, or identity modifications. Caching authorization decisions creates security vulnerabilities where access continues after permissions are revoked.

## Stage 4: Secret Operations and Business Logic

The final stage implements the actual business logic for secret operations, including encryption, decryption, lease management, and audit logging. This stage coordinates between multiple system components to fulfill the client request while maintaining security and consistency guarantees.

#### Secret Retrieval Process:

1. **Path Resolution:** The secret path is resolved to determine the storage backend and any path transformations required. Mount point processing handles backend-specific path mapping and parameter extraction.

2. **Encryption Context Preparation:** For encrypted secrets, the encryption context is prepared including the data encryption key version, authentication data, and any path-specific encryption parameters.
3. **Storage Operation:** The appropriate storage backend is invoked to retrieve the encrypted secret data. Storage errors are classified as temporary (retry) or permanent (fail immediately) based on error type.
4. **Decryption and Data Processing:** Retrieved secrets undergo decryption using the envelope encryption system. Version resolution ensures the correct data encryption key is used for each secret version.
5. **Response Preparation:** The decrypted secret data is formatted into the standard response envelope with appropriate metadata including version information, lease details, and renewal capabilities.
6. **Audit Event Generation:** Comprehensive audit events are generated capturing the complete request context, operation outcome, and any security-relevant details for compliance and security monitoring.

### **Dynamic Secret Generation Process:**

1. **Backend Resolution:** The dynamic secret request is routed to the appropriate secret backend based on the mount point. Backend-specific role and configuration validation occurs at this stage.
2. **Credential Generation:** The secret backend generates new credentials according to the role configuration. This may involve database user creation, cloud IAM role assumption, or certificate generation.
3. **Lease Creation and Tracking:** A lease is created to track the generated credentials including expiration time, renewal capabilities, and revocation data needed for cleanup.
4. **Credential Delivery:** Generated credentials are packaged into the response with lease metadata. Sensitive credential data is never logged or stored in plaintext.
5. **Background Lease Registration:** The lease is registered with the lease management system for tracking expiration and enabling background revocation processing.

## **Background Processes**

The secret management system operates several critical background processes that maintain security, performance, and operational health independently of client request processing. Think of these background processes like the maintenance staff in a corporate building - they work behind the scenes to ensure security systems function properly, expired access badges are deactivated, and facility operations continue smoothly without disrupting daily business operations.

These processes implement the "defense in depth" principle by providing multiple independent layers of security enforcement and system maintenance. Each process operates on its own schedule with appropriate error handling and retry logic to ensure system reliability even during partial failures.

### **Lease Management and Revocation Engine**

The lease management system tracks all dynamic secrets throughout their lifecycle and ensures proper cleanup when credentials expire or are explicitly revoked. This system prevents credential accumulation that could create security vulnerabilities or resource exhaustion in target systems.

## Lease Tracking Data Structure:

Component	Purpose	Update Frequency	Persistence Requirements
Lease Index by Expiration	Fast lookup of expired leases	Real-time on creation/renewal	Memory with periodic snapshots
Lease Index by Token	Token revocation support	Real-time on creation	Memory with write-through
Lease Index by Backend	Backend-specific operations	Real-time on creation	Memory with write-through
Revocation Queue	Retry failed revocations	As needed for failures	Persistent storage required

## Lease Reaper Process:

The lease reaper operates as a continuous background process that identifies expired leases and initiates their revocation. Think of the lease reaper like a security guard who regularly patrols the building to check for expired access badges and deactivate them to prevent unauthorized access.

- Expiration Scanning:** Every 30 seconds, the reaper queries the lease index for leases that have expired within the last scan interval. The query includes a small buffer (5 seconds) to account for clock skew and processing delays.
- Batch Processing:** Expired leases are processed in batches to optimize backend operations and reduce system load. Batch size is configurable but defaults to 100 leases to balance throughput and memory usage.
- Revocation Queuing:** Each expired lease is queued for revocation with appropriate priority based on the credential type and security sensitivity. High-priority credentials (root database users, admin cloud roles) are processed immediately.
- Progress Tracking:** The reaper maintains state about its last successful scan to ensure no leases are missed during system restarts or failures. This state is persisted to storage and recovered on startup.
- Error Handling:** Failed revocation attempts are logged and retried according to exponential backoff policies. Persistent failures are escalated to dead letter queues for manual investigation.

## Revocation Worker Process:

Multiple revocation workers operate concurrently to process the revocation queue and perform actual credential cleanup in target systems. The number of workers is configurable based on expected load and backend capacity.

- Queue Processing:** Workers continuously poll the revocation queue for pending items, using priority ordering to ensure high-priority revocations are processed first. Queue polling includes exponential backoff to reduce system load during idle periods.

2. **Backend Invocation:** For each revocation item, the appropriate secret backend is invoked with the stored revocation data. Backends implement credential-specific cleanup logic such as database user deletion or cloud IAM role revocation.
3. **Retry Logic with Circuit Breaker:** Failed revocation attempts are retried according to configurable policies with exponential backoff. Circuit breaker patterns prevent cascade failures when backends are experiencing issues.
4. **Success Confirmation:** Successfully revoked credentials are removed from all tracking indexes and their lease records are marked as revoked for audit purposes. Success confirmation may include verification queries to ensure cleanup completion.
5. **Failure Escalation:** Revocation attempts that exceed maximum retry counts are moved to dead letter queues with detailed error information for operator investigation. These failures generate alert notifications for immediate attention.

#### **Revocation Retry Configuration:**

Attempt	Delay	Total Time	Action
1	Immediate	0s	Direct revocation attempt
2	30 seconds	30s	First retry with backend health check
3	2 minutes	2m 30s	Second retry with extended timeout
4	8 minutes	10m 30s	Third retry with debug logging
5	32 minutes	42m 30s	Final retry with operator notification
Failed	Dead letter	N/A	Manual intervention required

#### **Decision: Separate Lease Reaper and Revocation Workers**

- **Context:** Need to balance timely credential revocation with system stability under varying load conditions
- **Options Considered:**
  - Single-threaded process handling both detection and revocation
  - Multi-threaded process with shared queue
  - Separate processes with persistent queue
- **Decision:** Separate processes communicating through persistent queue
- **Rationale:** Enables independent scaling, fault isolation, and priority handling while maintaining processing guarantees
- **Consequences:** Requires additional coordination and queue management but provides superior reliability and performance characteristics

## Token Maintenance and Cleanup

The token maintenance system handles token lifecycle management including expiration, usage limit enforcement, and orphaned token cleanup. This system prevents token accumulation and enforces security policies related to token usage patterns.

### Token Expiration Processing:

1. **Expiration Detection:** A background process scans the token store every minute to identify tokens that have exceeded their TTL. The scan uses indexed queries to minimize performance impact on active token operations.
2. **Grace Period Handling:** Expired tokens enter a short grace period (30 seconds) during which they are marked as expired but not immediately deleted. This prevents race conditions where in-flight requests fail due to token expiration timing.
3. **Revocation Cascade:** When parent tokens are revoked, all child tokens are automatically revoked in a cascading pattern. The revocation process maintains referential integrity and prevents orphaned authentication state.
4. **Audit Event Generation:** Token revocation events are logged with comprehensive context including revocation reason, token metadata, and any associated leases that require cleanup.

### Token Usage Limit Enforcement:

1. **Usage Tracking:** Every token authentication increments the token's usage counter in a thread-safe manner. Usage tracking includes protection against race conditions that could allow usage limit bypasses.
2. **Limit Validation:** Before authentication succeeds, the token's usage count is compared against its maximum usage limit. Tokens that would exceed their limit are immediately revoked and the authentication fails.
3. **Usage Pattern Analysis:** Token usage patterns are analyzed to detect anomalous behavior such as rapid usage spikes that might indicate credential compromise or client implementation errors.

### Orphaned Token Cleanup:

1. **Parent Relationship Validation:** Tokens with parent relationships are validated to ensure their parent tokens still exist and are valid. Orphaned tokens from deleted parents are queued for cleanup.
2. **Policy Reference Validation:** Tokens referencing policies that no longer exist are identified and either updated with replacement policies or revoked if no suitable replacement exists.
3. **Identity Reference Validation:** Tokens associated with deleted or disabled identities are automatically revoked to prevent access after identity removal.

## Encryption Key Rotation and Migration

The encryption key management system handles periodic rotation of data encryption keys and migration of secrets to new key versions. This process maintains security by limiting the exposure time of any single

encryption key while ensuring seamless access to historical secret versions.

#### **Automatic Key Rotation Process:**

1. **Rotation Schedule Evaluation:** The key rotation scheduler evaluates configured rotation policies including maximum key age, maximum usage count, and external rotation triggers. Rotation policies can be configured per mount point or globally.
2. **New Key Generation:** When rotation is triggered, a new data encryption key is generated using cryptographically secure random number generation. The new key is encrypted with the current master key and stored in the key store.
3. **Key Version Management:** The new key is assigned the next sequential version number and marked as the active key for new encryptions. Previous key versions remain available for decryption but are no longer used for new operations.
4. **Migration Planning:** The rotation process identifies existing secrets encrypted with old key versions and creates a migration plan. Migration is prioritized based on secret access frequency and age of the encryption key.

#### **Background Secret Re-encryption:**

1. **Migration Queue Processing:** A background process continuously works through the migration queue, re-encrypting secrets with the current data encryption key version. Processing rate is throttled to avoid impacting operational performance.
2. **Version Preservation:** During re-encryption, all secret versions are preserved with their original metadata. Only the encryption key version and encrypted data are updated while maintaining version history.
3. **Atomic Updates:** Secret re-encryption is performed atomically to prevent data corruption. The original secret version is preserved until the new version is successfully written and verified.
4. **Progress Tracking:** Migration progress is tracked per mount point and globally to provide operational visibility into key rotation status. Progress information is exposed through monitoring APIs.
5. **Verification and Rollback:** Re-encrypted secrets undergo verification by attempting decryption with the new key version. Failed verifications trigger automatic rollback to the previous version and error reporting.

#### **Key Retirement and Cleanup:**

1. **Usage Monitoring:** Old key versions are monitored for usage patterns. Keys that are no longer actively used for decryption operations become candidates for retirement.
2. **Retirement Timeline:** Keys follow a configured retirement timeline where they progress from active to deprecated to retired status. Each status change has configurable time delays to ensure operational safety.
3. **Final Cleanup:** Retired keys that are no longer referenced by any secrets are securely deleted from the key store. Key deletion includes secure memory wiping and audit event generation.

**Security Principle:** Key rotation operates on the principle of "blast radius containment" - limiting the amount of data encrypted with any single key version reduces the impact of potential key compromise while maintaining backward compatibility for legitimate access.

## Health Monitoring and Alerting

The health monitoring system continuously evaluates system components and operational metrics to detect degraded performance, security issues, or impending failures. This system provides early warning of problems and enables proactive operational response.

### Component Health Checks:

Component	Health Metrics	Check Frequency	Alert Thresholds
Storage Backend	Response time, error rate, connection count	30 seconds	>500ms response, >5% errors
Encryption Engine	Key availability, encryption/decryption performance	60 seconds	Key lookup failures, >100ms operations
Authentication Engine	Token validation performance, policy evaluation time	30 seconds	>50ms validation, >10% failures
Dynamic Secret Backends	Credential generation success rate, cleanup success	2 minutes	>10% generation failures, >5% cleanup failures
Lease Management	Revocation queue depth, processing rate	60 seconds	>1000 queued items, <90% success rate

### Operational Metrics Collection:

- Request Processing Metrics:** Comprehensive timing and success rate metrics for all API endpoints, categorized by operation type, authentication method, and client identity. Metrics include percentile distributions for performance analysis.
- Security Event Metrics:** Aggregated counts of authentication failures, authorization denials, and other security-relevant events. These metrics enable detection of potential attacks or misconfigurations.
- Resource Usage Metrics:** Memory usage, storage consumption, network utilization, and other resource metrics that indicate system capacity and potential scaling needs.
- Background Process Metrics:** Performance and health metrics for all background processes including lease reapers, token cleanup, and key rotation processes.

### Alert Generation and Escalation:

1. **Threshold-Based Alerts:** Automated alerts are generated when metrics exceed configured thresholds. Alert severity is determined by the magnitude of threshold exceedance and the criticality of the affected component.
2. **Trend-Based Alerts:** Statistical analysis of metric trends identifies gradual degradation that might not trigger threshold alerts but indicates developing issues requiring attention.
3. **Correlation Analysis:** Multiple related metrics are analyzed together to distinguish between transient issues and systematic problems. Correlation reduces alert noise while improving detection accuracy.
4. **Escalation Policies:** Alerts follow configured escalation policies with increasing severity and notification scope based on alert duration and system impact.

## Implementation Guidance

The interactions and data flow components require careful attention to HTTP server implementation, middleware processing, and background task management. The following guidance provides a foundation for building these interaction patterns with appropriate error handling and performance characteristics.

## Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	Go net/http with gorilla/mux	Go with gin-gonic or fiber frameworks
JSON Processing	Go encoding/json	Go with easyjson for performance
Background Jobs	Go goroutines with sync primitives	Go with machinery or async job queues
Metrics Collection	Go expvar with custom collectors	Prometheus client library
Request Logging	Go log/slog with structured output	Structured logging with logrus or zap
HTTP Middleware	Custom middleware functions	Go middleware libraries (negroni, alice)

## File Structure Organization

```
internal/server/
  server.go           ← HTTP server setup and configuration
  middleware.go       ← Authentication, authorization, logging middleware
  handlers.go         ← HTTP request handlers for each endpoint
  response.go         ← Standard response formatting utilities
  validation.go        ← Request validation helpers

internal/api/
  auth.go             ← Authentication endpoint handlers
  secrets.go          ← Secret operation endpoint handlers
  dynamic.go          ← Dynamic secret endpoint handlers
  system.go           ← System administration endpoint handlers

internal/background/
  lease_reaper.go    ← Lease expiration detection and cleanup
  token_cleaner.go   ← Token lifecycle maintenance
  key_rotator.go     ← Encryption key rotation management
  health_monitor.go  ← System health checking and alerting

internal/pipeline/
  request.go          ← Request processing pipeline stages
  context.go          ← Request context and metadata handling
  audit.go            ← Audit event generation and logging
```

## Core HTTP Server Infrastructure

This complete HTTP server foundation handles basic server setup, graceful shutdown, and middleware composition. Use this as your starting point and add endpoint-specific handlers.

```
package server

import (
    "context"
    "encoding/json"
    "fmt"
    "log/slog"
    "net/http"
    "time"

    "github.com/gorilla/mux"
)

// Complete HTTP server with middleware pipeline and graceful shutdown

type SecretServer struct {

    config      ServerConfig

    httpServer *http.Server

    storage     Backend

    encryption *Engine

    auth        *Engine

    dynamic     *Engine

    unseal      *UnsealManager

    isSealed    bool

    sealMutex   sync.RWMutex

    logger      *slog.Logger
}

// NewSecretServer creates a fully configured HTTP server with all middleware
```

GO

```
func NewSecretServer(cfg ServerConfig) (*SecretServer, error) {

    server := &SecretServer{

        config: cfg,
        logger: slog.New(slog.NewJSONHandler(os.Stdout, nil)),
    }

    // Initialize all engines based on configuration

    if err := server.initializeEngines(); err != nil {
        return nil, fmt.Errorf("failed to initialize engines: %w", err)
    }

    // Setup HTTP router with all endpoints and middleware

    router := server.setupRouter()

    server.httpServer = &http.Server{

        Addr:         fmt.Sprintf(":%d", cfg.Server.Port),
        Handler:      router,
        ReadTimeout:  cfg.Server.ReadTimeout,
        WriteTimeout: cfg.Server.WriteTimeout,
        TLSConfig:    server.buildTLSConfig(),
    }

    return server, nil
}

// Start begins HTTP server operation with TLS

func (s *SecretServer) Start() error {
```

```
s.logger.Info("Starting secret management server",
    "port", s.config.Server.Port,
    "tls", s.config.Server.TLSCertPath != "")

// Start background processes

go s.startBackgroundProcesses()

// Start HTTP server with TLS

if s.config.Server.TLSCertPath != "" {
    return s.httpServer.ListenAndServeTLS(
        s.config.Server.TLSCertPath,
        s.config.Server.TLSKeyPath,
    )
}

return s.httpServer.ListenAndServe()
}

// Shutdown gracefully stops the server and background processes

func (s *SecretServer) Shutdown(ctx context.Context) error {
    s.logger.Info("Shutting down secret management server")

    // Stop background processes first

    s.stopBackgroundProcesses()

    // Gracefully shutdown HTTP server

    return s.httpServer.Shutdown(ctx)
}
```

```
// setupRouter configures all HTTP routes and middleware pipeline

func (s *SecretServer) setupRouter() http.Handler {
    router := mux.NewRouter()

    // Apply middleware in correct order (outermost first)
    router.Use(s.loggingMiddleware)
    router.Use(s.corsMiddleware)
    router.Use(s.sealCheckMiddleware)

    // API v1 routes
    v1 := router.PathPrefix("/v1").Subrouter()

    // Public endpoints (no authentication)
    v1.HandleFunc("/sys/health", s.handleHealth).Methods("GET")
    v1.HandleFunc("/sys/seal-status", s.handleSealStatus).Methods("GET")

    // Authentication endpoints (partial auth required)
    auth := v1.PathPrefix("/auth").Subrouter()
    auth.HandleFunc("/token/create", s.handleCreateToken).Methods("POST")
    auth.HandleFunc("/token/lookup-self", s.handleTokenLookup).Methods("GET")

    // Protected endpoints (full authentication required)
    protected := v1.NewRoute().Subrouter()
    protected.Use(s.authenticationMiddleware)
    protected.Use(s.authorizationMiddleware)

    // Secret operations
```

```
protected.HandleFunc("/secret/{path:.?}", s.handleSecretGet).Methods("GET")

protected.HandleFunc("/secret/{path:.?}", s.handleSecretPut).Methods("POST", "PUT")

protected.HandleFunc("/secret/{path:.?}", s.handleSecretDelete).Methods("DELETE")

// Dynamic secret operations

protected.HandleFunc("/database/creds/{role}", s.handleDynamicSecret).Methods("GET")

protected.HandleFunc("/aws/creds/{role}", s.handleDynamicSecret).Methods("GET")

// System administration

protected.HandleFunc("/sys/policy/{name}", s.handlePolicyOperations).Methods("GET",
"POST", "DELETE")

return router

}
```

## Request Processing Pipeline Implementation

The request processing pipeline handles the multi-stage validation, authentication, and authorization flow. Each stage can terminate the request early or enrich the request context for subsequent stages.

```
package pipeline
```

GO

```
// RequestContext carries all information through the processing pipeline
```

```
type RequestContext struct {

    RequestID      string
    StartTime      time.Time
    Identity       *Identity
    Token          *Token
    Operation      string
    Path           string
    Parameters     map[string][]string
    ClientIP       string
    UserAgent      string
    Authorized     bool
    AuditEvents    []AuditEvent
}
```

```
// ProcessingStage represents a single stage in the request pipeline
```

```
type ProcessingStage func(ctx context.Context, reqCtx *RequestContext, r *http.Request) error
```

```
// RequestProcessor coordinates the multi-stage request processing pipeline
```

```
type RequestProcessor struct {

    stages []ProcessingStage
    audit  *AuditLogger
    logger *slog.Logger
}
```

```
// NewRequestProcessor creates a processor with all standard stages configured
```

```
func NewRequestProcessor(audit *AuditLogger) *RequestProcessor {
    processor := &RequestProcessor{
        audit: audit,
        logger: slog.New(slog.NewJSONHandler(os.Stdout, nil)),
    }

    // Configure standard processing stages in order
    processor.stages = []ProcessingStage{
        processor.parseAndValidateStage,
        processor.authenticateStage,
        processor.authorizeStage,
        processor.auditStage,
    }

    return processor
}

// ProcessRequest executes all pipeline stages and returns processing result
func (p *RequestProcessor) ProcessRequest(ctx context.Context, r *http.Request) (*RequestContext, error) {
    reqCtx := &RequestContext{
        RequestID: generateRequestID(),
        StartTime: time.Now(),
        ClientIP:  extractClientIP(r),
        UserAgent: r.UserAgent(),
    }

    // Execute each processing stage in sequence
```

```
for i, stage := range p.stages {

    if err := stage(ctx, reqCtx, r); err != nil {

        // Log pipeline failure with stage information

        p.logger.Error("Request processing failed",

            "request_id", reqCtx.RequestID,
            "stage", i,
            "error", err,
            "duration", time.Since(reqCtx.StartTime))

        // Generate failure audit event

        p.auditFailure(reqCtx, err)

        return reqCtx, err
    }
}

return reqCtx, nil
}

// parseAndValidateStage handles HTTP parsing and basic request validation

func (p *RequestProcessor) parseAndValidateStage(ctx context.Context, reqCtx
*RequestContext, r *http.Request) error {

    // TODO 1: Extract HTTP method and validate against allowed methods for endpoint

    // TODO 2: Parse URL path and extract path parameters using mux.Vars(r)

    // TODO 3: Validate Content-Type header for requests with body (POST/PUT)

    // TODO 4: Parse and validate JSON request body with size limits

    // TODO 5: Extract query parameters and validate types/ranges

    // TODO 6: Store parsed operation and path in RequestContext

    // Hint: Use r.Body with io.LimitReader to prevent memory exhaustion
}
```

```
    return nil

}

// authenticateStage verifies client identity and resolves authentication

func (p *RequestProcessor) authenticateStage(ctx context.Context, reqCtx *RequestContext, r *http.Request) error {

    // TODO 1: Extract X-Vault-Token header from request

    // TODO 2: Validate token format and handle missing tokens for public endpoints

    // TODO 3: Look up token in token store using constant-time comparison

    // TODO 4: Validate token expiration and usage limits

    // TODO 5: Load token metadata including policies and identity information

    // TODO 6: Construct Identity object with resolved policies and groups

    // TODO 7: Update token usage count and last used timestamp

    // Hint: Use ValidateTokenConstantTime to prevent timing attacks

    return nil
}

// authorizeStage evaluates policies and determines access permissions

func (p *RequestProcessor) authorizeStage(ctx context.Context, reqCtx *RequestContext, r *http.Request) error {

    // TODO 1: Skip authorization for public endpoints (health, seal-status)

    // TODO 2: Collect all policies associated with authenticated identity

    // TODO 3: Evaluate each policy rule against normalized request path

    // TODO 4: Check requested operation against capabilities in matching rules

    // TODO 5: Validate request parameters against policy constraints

    // TODO 6: Apply maximum TTL and other policy constraints to request

    // TODO 7: Set Authorized flag in RequestContext based on evaluation result

    // Hint: Use PathMatcher.Match for wildcard pattern evaluation

    return nil
```

```
}

// auditStage generates audit events for all request processing

func (p *RequestProcessor) auditStage(ctx context.Context, reqCtx *RequestContext, r *http.Request) error {

    // TODO 1: Create AuditEvent with comprehensive request information

    // TODO 2: Include authentication and authorization results

    // TODO 3: Add timing information and request metadata

    // TODO 4: Determine audit event type based on operation category

    // TODO 5: Write audit event to configured audit log

    // Hint: Generate audit events even for failed requests for security monitoring

    return nil
}
```

## Background Process Infrastructure

Background processes require careful lifecycle management, error handling, and coordination. This infrastructure provides the foundation for lease management, token cleanup, and health monitoring.

```
package background
```

GO

```
// BackgroundManager coordinates all background processes with lifecycle management
```

```
type BackgroundManager struct {
```

```
    processes []BackgroundProcess
```

```
    stopChan chan struct{}
```

```
    wg sync.WaitGroup
```

```
    logger *slog.Logger
```

```
}
```

```
// BackgroundProcess defines the interface for all background processes
```

```
type BackgroundProcess interface {
```

```
    Name() string
```

```
    Start(stopChan <-chan struct{}) error
```

```
    HealthCheck() error
```

```
}
```

```
// LeaseReaper handles expired lease detection and revocation queuing
```

```
type LeaseReaper struct {
```

```
    engine *Engine
```

```
    storage Backend
```

```
    interval time.Duration
```

```
    batchSize int
```

```
    lastScan time.Time
```

```
    logger *slog.Logger
```

```
}
```

```
// NewLeaseReaper creates a lease reaper with configured scan interval
```

```
func NewLeaseReaper(engine *Engine, storage Backend) *LeaseReaper {
```

```
        return &LeaseReaper{  
  
            engine:    engine,  
  
            storage:   storage,  
  
            interval:  30 * time.Second,  
  
            batchSize: 100,  
  
            logger:    slog.New(slog.NewJSONHandler(os.Stdout, nil)),  
        }  
    }  
  
    // Start begins the lease reaper background process  
  
    func (lr *LeaseReaper) Start(stopChan <-chan struct{}) error {  
  
        ticker := time.NewTicker(lr.interval)  
  
        defer ticker.Stop()  
  
  
        lr.logger.Info("Starting lease reaper", "interval", lr.interval)  
  
  
        for {  
  
            select {  
  
                case <-stopChan:  
  
                    lr.logger.Info("Stopping lease reaper")  
  
                    return nil  
  
  
                case <-ticker.C:  
  
                    if err := lr.processExpiredLeases(); err != nil {  
  
                        lr.logger.Error("Lease reaper processing failed", "error", err)  
  
                        // Continue processing - don't stop on errors  
                    }  
            }  
        }  
    }  
}
```

```
        }

    }

}

// processExpiredLeases finds and queues expired leases for revocation

func (lr *LeaseReaper) processExpiredLeases() error {

    // TODO 1: Query storage for leases expired since last scan time

    // TODO 2: Process expired leases in batches to manage memory usage

    // TODO 3: For each expired lease, create revocation queue entry

    // TODO 4: Queue revocation requests with appropriate priority

    // TODO 5: Update last scan time to current time

    // TODO 6: Log processing statistics (leases found, queued, errors)

    // Hint: Include small buffer time to account for clock skew

    return nil
}

// RevocationWorker processes the revocation queue and performs credential cleanup

type RevocationWorker struct {

    workerID      int
    engine        *Engine
    retryConfig   RetryConfig
    queue         chan *RevocationQueue
    stopChan      chan struct{}
    logger        *slog.Logger
}

// NewRevocationWorker creates a worker that processes revocation requests

func NewRevocationWorker(workerID int, engine *Engine, queue chan *RevocationQueue) *RevocationWorker {
```

```
return &RevocationWorker{  
  
    workerID: workerID,  
  
    engine: engine,  
  
    retryConfig: RetryConfig{  
  
        InitialDelay: 30 * time.Second,  
  
        MaxDelay:     30 * time.Minute,  
  
        Multiplier:   2.0,  
  
        MaxAttempts:  5,  
  
    },  
  
    queue: queue,  
  
    logger: slog.New(slog.NewJSONHandler(os.Stdout, nil)),  
  
}  
  
}  
  
// Start begins processing revocation requests from the queue  
  
func (rw *RevocationWorker) Start(stopChan <-chan struct{}) error {  
  
    rw.stopChan = stopChan  
  
    rw.logger.Info("Starting revocation worker", "worker_id", rw.workerID)  
  
  
  
    for {  
  
        select {  
  
            case <-stopChan:  
  
                rw.logger.Info("Stopping revocation worker", "worker_id", rw.workerID)  
  
                return nil  
  
  
  
            case revReq := <-rw.queue:  
  
                if err := rw.processRevocation(revReq); err != nil {  

```

```

        rw.logger.Error("Revocation processing failed",
                         "worker_id", rw.workerID,
                         "lease_id", revReq.LeaseID,
                         "error", err)

    }

}

}

}

// processRevocation handles a single revocation request with retry logic

func (rw *RevocationWorker) processRevocation(revReq *RevocationQueue) error {
    // TODO 1: Look up secret backend for the lease being revoked

    // TODO 2: Attempt credential revocation using backend-specific logic

    // TODO 3: Handle revocation success by removing lease from indexes

    // TODO 4: Handle revocation failure with exponential backoff retry

    // TODO 5: Move persistently failing revocations to dead letter queue

    // TODO 6: Update revocation statistics and monitoring metrics

    // Hint: Use circuit breaker pattern to prevent cascade failures

    return nil
}

```

## Milestone Checkpoints

After implementing the interactions and data flow components, verify the system behavior with these checkpoints:

### **Authentication Pipeline Verification:**

```
# Test valid token authentication                                                 BASH

curl -H "X-Vault-Token: valid-token-here" https://localhost:8443/v1/secret/test

# Expected: HTTP 200 with secret data or HTTP 404 if secret doesn't exist

# Signs of problems: HTTP 401 (auth failure), HTTP 500 (server error)

# Test invalid token handling

curl -H "X-Vault-Token: invalid-token" https://localhost:8443/v1/secret/test

# Expected: HTTP 403 Forbidden with error message

# Signs of problems: HTTP 500 (auth system failure), HTTP 200 (security bypass)
```

### Authorization Policy Verification:

```
# Test path-based policy enforcement                                              BASH

curl -H "X-Vault-Token: limited-token" https://localhost:8443/v1/secret/forbidden/path

# Expected: HTTP 403 Forbidden with policy denial message

# Signs of problems: HTTP 200 (policy bypass), HTTP 500 (policy evaluation failure)

# Verify policy wildcard matching

curl -H "X-Vault-Token: wildcard-token" https://localhost:8443/v1/secret/allowed/subpath

# Expected: HTTP 200 or 404 based on secret existence

# Signs of problems: HTTP 403 (wildcard not working), HTTP 500 (pattern matching failure)
```

### Background Process Verification:

```
# Check background process health
curl https://localhost:8443/v1/sys/health

# Expected: JSON with background process status information

# Look for: lease_reaper_status: "healthy", revocation_workers: "active"

# Signs of problems: process_status: "failed", high error counts

# Monitor lease cleanup

# Create dynamic secret, wait for expiration, verify cleanup

# Check logs for: "lease expired and queued for revocation"

# Check target system for: credential actually removed
```

### Common Implementation Issues:

**⚠ Pitfall: Missing Request Context Propagation** Forgetting to pass the request context through all pipeline stages causes loss of audit information and makes debugging impossible. Each stage must receive and can modify the RequestContext object, and all audit events must reference the same request ID for correlation.

**⚠ Pitfall: Blocking Background Processes** Background processes that don't properly handle the stop channel can prevent graceful shutdown and cause resource leaks. Always use select statements with the stop channel and ensure goroutines terminate cleanly when requested.

**⚠ Pitfall: Inconsistent Error Response Format** HTTP endpoints that return different error formats make client implementation difficult and break compatibility with Vault clients. Use a consistent error response structure across all endpoints with appropriate HTTP status codes.

**⚠ Pitfall: Missing Constant-Time Token Comparison** Using standard string comparison for token validation enables timing attacks where attackers can determine valid token prefixes by measuring response times. Always use constant-time comparison functions for any security-sensitive string comparisons.

## Error Handling and Edge Cases

**Milestone(s):** This section applies to all four milestones, with particular emphasis on Milestone 4 (High Availability) for cluster-related failures. Error handling strategies are built progressively through each milestone.

A robust secret management system must operate reliably even when components fail, networks partition, or external systems become unavailable. Think of this as designing a bank vault system that continues to protect

assets and serve authorized customers even during earthquakes, power outages, or attempted break-ins. The system must gracefully degrade functionality while maintaining its core security guarantees, never compromising secrets even in catastrophic failure scenarios.

This section examines the primary failure modes that can affect a secret management system, the detection and recovery strategies for each failure type, and how the system maintains partial functionality when full operation is impossible. The key insight is that security requirements never relax during failures - if anything, the system must become more conservative and protective when operating in degraded states.

## System Failure Modes

Understanding potential failure modes is critical for building resilient secret management systems. Each failure mode requires specific detection mechanisms, recovery procedures, and fallback strategies. The system must be designed with the principle of "assume breach" - any component can fail at any time, and the system must continue to protect secrets.

### Network Partitions and Communication Failures

Network partitions represent one of the most challenging failure modes for distributed secret management systems. Unlike simple network outages where nodes are clearly unavailable, partitions create split-brain scenarios where subsets of nodes can communicate internally but cannot reach other subsets. This is analogous to a bank having multiple branches that can operate independently but cannot communicate with headquarters - each branch must decide whether to continue serving customers or temporarily halt operations.

Failure Scenario	Symptoms	Detection Method	Immediate Response
Network partition between cluster nodes	Raft consensus timeouts, leader election failures	Heartbeat monitoring, consensus operation failures	Transition non-leader nodes to read-only mode
Client-to-server network issues	HTTP request timeouts, connection refused errors	Client-side timeout detection, load balancer health checks	Client retry with exponential backoff, circuit breaker activation
Storage backend network failures	Database connection timeouts, cloud storage API errors	Backend health check failures, operation timeouts	Switch to cached data, enable read-only mode
KMS provider network issues	Auto-unseal failures, key operation timeouts	KMS health check timeouts, encryption operation failures	Manual unseal fallback, defer key operations

The system implements several strategies to handle network-related failures. For cluster communication, the Raft consensus protocol provides built-in partition tolerance by requiring a majority quorum for write operations. When a partition occurs, only the subset containing a majority of nodes can continue accepting write requests,

while minority partitions automatically transition to read-only mode. This prevents the dangerous split-brain scenario where multiple subsets accept conflicting writes.

### Design Insight: Conservative Degradation

When facing ambiguous failure conditions, the system always chooses the more restrictive option. A node that cannot confirm it's part of the majority partition will refuse write operations rather than risk creating inconsistency. This "fail-safe" approach prioritizes data integrity over availability.

Client-facing network failures require careful retry logic to avoid overwhelming a recovering system. The system implements exponential backoff with jitter to spread retry attempts over time. Circuit breaker patterns prevent cascading failures by temporarily halting requests to unresponsive backends, allowing them time to recover.

### Decision: Network Partition Handling Strategy

- **Context:** Network partitions can create split-brain scenarios where multiple nodes accept conflicting writes, leading to data inconsistency
- **Options Considered:**
  - Allow all nodes to continue operations independently
  - Require strict majority quorum for any operations
  - Implement Byzantine fault tolerance with complex consensus
- **Decision:** Raft consensus with majority quorum requirement and read-only degradation for minority partitions
- **Rationale:** Raft provides proven partition tolerance with simpler implementation than Byzantine consensus, while majority quorum prevents split-brain scenarios without complex coordination
- **Consequences:** Enables consistent behavior during partitions but requires majority of nodes for write availability, increasing minimum cluster size requirements

### Storage System Failures

Storage failures can range from temporary connectivity issues to permanent data corruption. The secret management system must handle these failures without exposing unencrypted secrets or losing critical metadata. Think of this as a library system where individual books might become damaged, entire shelves might become inaccessible, or the card catalog might become corrupted - the library must continue operating while protecting its collection.

Storage Failure Type	Detection Signals	Data at Risk	Recovery Strategy
Disk corruption	Checksum validation failures, read errors	Individual secret versions	Restore from backup, mark corrupted versions as unavailable
Database connection loss	Connection timeouts, SQL errors	All persisted data temporarily inaccessible	Retry with exponential backoff, switch to cached data
Storage backend crash	Health check failures, persistent connection errors	All stored secrets and metadata	Failover to backup storage, restore from replicated data
Encryption key corruption	Decryption failures, key validation errors	Secrets encrypted with corrupted key	Key recovery from shares, re-encryption with new key
Transaction log corruption	WAL read errors, sequence validation failures	Recent writes may be lost	Replay from last good checkpoint, rebuild from snapshots

The system implements multiple layers of protection against storage failures. At the data level, all secrets include cryptographic checksums to detect corruption. The envelope encryption design isolates failures - if a data encryption key becomes corrupted, only secrets encrypted with that specific key are affected, while other secrets remain accessible.

For database connectivity issues, the system maintains read-through caches for frequently accessed secrets and policies. This allows continued read operations during temporary storage outages. Write operations are queued with persistent storage to ensure no requests are lost during brief connectivity issues.

The most critical storage failure involves corruption of the master key or key shares. The system addresses this through redundant storage of encrypted key shares across multiple locations and automated integrity checking. If key corruption is detected, the system can reconstruct the master key from remaining shares and re-encrypt affected data.

### Critical Security Principle: Fail Closed

When storage integrity cannot be verified, the system denies access rather than potentially serving corrupted or malicious data. A corrupted secret is treated the same as a missing secret - the request fails rather than returning potentially compromised data.

## Encryption Key Corruption and Loss

Key management failures represent catastrophic scenarios for a secret management system. Unlike other data that can be restored from backups, encryption keys are unique - losing the master key means permanently losing access to all encrypted secrets. This is analogous to losing the combination to a bank vault - even if the vault and its contents are physically intact, they become inaccessible.

Key Failure Scenario	Probability	Impact Severity	Detection Method	Recovery Options
Master key corruption in memory	Low	High	Decryption operation failures	Reconstruct from Shamir shares
Single key share corruption	Medium	Low	Share validation during reconstruction	Use remaining shares, regenerate corrupted share
Multiple key share corruption	Low	Critical	Insufficient shares for reconstruction	Restore shares from secure backup, emergency key escrow
KMS provider key deletion	Low	Critical	Auto-unseal failures, KMS API errors	Manual unseal with backup shares, key recovery procedures
Data encryption key corruption	Medium	Medium	Secret decryption failures for specific key version	Re-encrypt affected secrets with new DEK

The Shamir's secret sharing implementation provides redundancy for master key protection. The system can tolerate corruption of up to `threshold - 1` shares while maintaining full functionality. When share corruption is detected, the system can regenerate new shares from the reconstructed master key and distribute them to replace corrupted copies.

For data encryption keys, the system maintains a key derivation tree that allows regeneration of corrupted DEKs from the master key and cryptographic parameters. This ensures that temporary DEK corruption doesn't result in permanent secret loss, though it may require re-encryption operations.

The most sophisticated protection involves integration with hardware security modules (HSMs) or cloud key management services for key escrow. These external systems provide independent key recovery mechanisms when all local shares become unavailable.

## Decision: Key Recovery Strategy Design

- **Context:** Encryption key loss or corruption could result in permanent data loss, requiring robust recovery mechanisms
- **Options Considered:**
  - Rely solely on Shamir shares for all key recovery
  - Implement key escrow with trusted third party
  - Use deterministic key derivation from user passwords
- **Decision:** Combination of Shamir shares for primary recovery with optional KMS escrow for disaster scenarios

- **Rationale:** Shamir shares provide offline recovery without external dependencies, while KMS escrow offers additional protection for catastrophic local failures without introducing routine external dependencies
- **Consequences:** Enables robust key recovery with multiple fallback options but increases operational complexity and introduces potential external dependencies for disaster recovery

## Service Dependencies and External Systems

Secret management systems rely on numerous external services for full functionality. Database backends, cloud KMS providers, network time services, and certificate authorities can all experience outages or degraded performance. The system must continue protecting secrets and serving authorized requests even when these dependencies become unavailable.

Dependency Type	Failure Impact	Degraded Functionality	Fallback Strategy
Database backend	Cannot persist new secrets or leases	Read from cache, reject writes	Local storage, backup database
Cloud KMS provider	Auto-unseal unavailable, cannot encrypt new DEKs	Manual unseal required, read-only mode	Cached DEKs, manual key management
Time service (NTP)	Token expiration checks unreliable, lease management affected	Conservative expiration handling	Local clock with skew tolerance
Certificate Authority	Cannot issue new client certificates, mTLS validation issues	Token authentication only, cached certificate validation	Cached CA certificates, extended validity periods
Dynamic secret backends	Cannot generate new credentials, revocation may fail	Static secret access only	Cached credentials, manual cleanup

The system implements graceful degradation for each dependency type. When the database backend becomes unavailable, the system continues serving read requests from in-memory caches while queuing write operations for later processing. This maintains availability for secret retrieval operations while ensuring write requests aren't lost.

Cloud KMS failures affect auto-unseal capabilities but don't prevent normal operations once the system is unsealed. The system falls back to manual unsealing procedures and can continue operating with locally managed encryption keys. New DEK generation may be deferred until KMS connectivity is restored.

Dynamic secret backend failures have the most complex implications, as they affect both credential generation and revocation. The system implements queue-based revocation with persistent retry logic to ensure credentials are eventually cleaned up even if backends are temporarily unavailable.

## Recovery Strategies

Effective recovery strategies ensure the secret management system can restore full functionality after failures while maintaining security guarantees throughout the recovery process. Recovery procedures must be thoroughly documented, regularly tested, and executable under stress conditions. Think of these as emergency procedures for a nuclear power plant - they must be simple enough to execute correctly during crisis situations while maintaining all safety protocols.

### Backup and Restore Procedures

Comprehensive backup strategies protect against both data loss and complete system failures. The secret management system requires specialized backup procedures due to the sensitive nature of the data and the complex relationships between encrypted secrets, encryption keys, and access control metadata.

Backup Component	Backup Frequency	Encryption Requirements	Restoration Priority
Encrypted secret data	Hourly incremental, daily full	Separate encryption from operational keys	High - core system functionality
Shamir key shares	Immediate after generation, stored offline	Hardware security modules or air-gapped systems	Critical - required for any secret access
Access policies and tokens	Daily full, immediate for policy changes	Standard encryption, integrity protection	Medium - affects authorization
Audit logs	Continuous streaming, hourly checkpoints	Write-once storage, tamper-evident sealing	Low - forensics and compliance
System configuration	After each change, weekly full	Configuration-specific encryption	Medium - system restoration

The backup strategy implements the principle of "separation of concerns" - secrets and keys are backed up using different procedures, storage locations, and encryption methods. This ensures that compromise of a backup system doesn't provide access to both encrypted data and the keys needed to decrypt it.

Secret data backups include all encrypted content with full version history but exclude encryption keys. These backups can be stored in standard backup systems since the data remains encrypted. The backup process verifies data integrity through cryptographic checksums and maintains metadata about backup consistency points.

Key share backups require the highest security measures. Shamir shares are distributed across multiple secure locations, with each location storing only a subset of shares insufficient for key reconstruction. Physical security measures include safe deposit boxes, HSMs, or geographically distributed key escrow services.

## Security Principle: Backup Separation

No single backup system or location contains both encrypted secrets and the keys needed to decrypt them. Even if an attacker compromises backup infrastructure, they cannot access secret content without also compromising separate key storage systems.

Restoration procedures follow carefully orchestrated sequences to ensure security throughout the recovery process. The system must be unsealed before secret restoration can begin, requiring collection of sufficient key shares. Once unsealed, data restoration proceeds in dependency order: storage backend, encryption keys, secret data, access policies, and finally audit log replay.

## Decision: Backup Strategy Architecture

- **Context:** Secret management systems require specialized backup procedures due to encryption key separation and security requirements
- **Options Considered:**
  - Traditional database backups with encrypted tablespace
  - Application-level export with separate key and data streams
  - Continuous replication with encrypted transport
- **Decision:** Application-level export with cryptographically separated key and data backup streams
- **Rationale:** Provides maximum control over encryption boundaries, enables independent key and data recovery procedures, and supports cross-platform restoration without database dependencies
- **Consequences:** Enables flexible disaster recovery options with strong security isolation but requires custom backup tooling and more complex restoration procedures

## Disaster Recovery Procedures

Disaster recovery goes beyond simple backup restoration to address scenarios where entire data centers, regions, or operational teams become unavailable. The procedures must be executable by skeleton staff with minimal specialized knowledge while maintaining all security protocols.

Disaster Scenario	Recovery Time Objective	Recovery Point Objective	Required Resources	Automation Level
Single server failure	5 minutes	0 data loss	Standby server, current backups	Fully automated
Data center outage	30 minutes	1 hour data loss	Alternate data center, recent backups	Semi-automated
Regional disaster	4 hours	4 hours data loss	Geographically distributed backups, emergency team	Manual with automation tools
Complete system compromise	24 hours	24 hours data loss	Clean environment, offline backups, new hardware	Fully manual
Key personnel unavailable	8 hours	1 hour data loss	Cross-trained staff, documented procedures	Manual with guidance

Each disaster scenario has specific recovery procedures optimized for the expected conditions. Single server failures leverage automated failover with standby systems that continuously replicate data. The standby promotion process includes unsealing procedures and health verification before accepting production traffic.

Data center outages require coordination between backup data centers and restoration of service at alternate locations. The procedures include network reconfiguration, DNS updates, and verification that all security boundaries remain intact during the transition.

Regional disasters invoke the most comprehensive recovery procedures, potentially requiring reconstruction of the entire system infrastructure. These procedures emphasize security verification at each step, ensuring that restored systems haven't been compromised during the recovery process.

The most critical disaster scenario involves complete system compromise, where attackers may have accessed both operational systems and backup infrastructure. Recovery procedures assume that all online systems are potentially compromised and require building new infrastructure from verified offline backups.

### Operational Principle: Recovery Rehearsal

Disaster recovery procedures are practiced quarterly with rotating team members to ensure knowledge transfer and procedure validation. Each rehearsal identifies process improvements and updates documentation to reflect current system state.

Recovery procedures include specific verification steps to confirm system integrity after restoration. These include cryptographic verification of restored data, access control validation, and security boundary testing. The system doesn't return to full operation until all verification procedures complete successfully.

## Data Consistency and Integrity Validation

After any failure and recovery event, the system must verify that all data remains consistent and uncorrupted. This process goes beyond simple backup restoration to include cryptographic verification, cross-reference validation, and security boundary testing.

Validation Type	Verification Method	Expected Duration	Failure Response
Cryptographic integrity	SHA-256 checksums for all secret versions	10 minutes per 10k secrets	Mark corrupted secrets as unavailable, restore from backup
Encryption key validation	Decrypt sample secrets with all key versions	5 minutes per key version	Regenerate corrupted keys, re-encrypt affected secrets
Access policy consistency	Policy rule parsing and conflict detection	2 minutes per 100 policies	Restore policy from backup, audit recent changes
Audit log continuity	Sequence number validation and timestamp verification	1 minute per day of logs	Investigate gap causes, restore missing log segments
Dynamic secret lease validity	Backend connectivity and credential verification	30 seconds per active lease	Revoke unverifiable leases, regenerate credentials

The validation process follows a systematic approach that verifies each system component independently before testing integrated functionality. Cryptographic verification confirms that all secrets can be successfully decrypted and that their content matches expected checksums. This catches both corruption and malicious tampering.

Encryption key validation involves attempting to decrypt sample secrets using each available key version. Failed decryptions indicate key corruption and trigger key recovery procedures. The system maintains a small set of test secrets specifically for this validation process.

Access policy validation parses all policy definitions to ensure they contain valid syntax and don't create security vulnerabilities such as overly broad wildcard permissions. The system also checks for policy conflicts that might create ambiguous authorization decisions.

## Decision: Post-Recovery Validation Strategy

- **Context:** After system recovery, comprehensive validation must confirm data integrity and security without disrupting restored service
- **Options Considered:**
  - Background validation with immediate service restoration
  - Complete validation before service restoration
  - Incremental validation with gradual service enablement
- **Decision:** Incremental validation with gradual service enablement, starting with read-only access

- **Rationale:** Balances service availability with security verification, allows early detection of validation failures without full service disruption, provides ability to halt service expansion if issues are discovered
- **Consequences:** Enables faster partial service restoration with controlled risk exposure but requires more complex service state management during recovery periods

## Graceful Degradation

When the secret management system cannot operate at full capacity due to failures or resource constraints, it must gracefully reduce functionality while maintaining its core security guarantees. Think of this as a hospital operating on emergency power - certain services may be suspended, but life-critical functions continue with uncompromised safety standards.

### Read-Only Mode Operation

Read-only mode represents the most common degraded operation state, typically triggered by storage backend failures, cluster partition scenarios, or planned maintenance. In this mode, the system continues serving existing secrets to authorized clients while refusing all operations that would modify system state.

Operation Type	Read-Only Behavior	Error Response	Client Impact
Secret retrieval	Full functionality maintained	None - normal operation	No impact on existing secret access
Secret creation/update	Rejected with explicit error	HTTP 503 Service Unavailable	Applications must retry or use cached values
Dynamic secret generation	Rejected for new requests	HTTP 503 Service Unavailable	Applications must use existing credentials longer
Policy changes	Rejected with explicit error	HTTP 503 Service Unavailable	Administrative operations deferred
Token creation	Rejected for new requests	HTTP 503 Service Unavailable	Authentication uses existing tokens only
Lease renewal	Allowed if cached locally	Partial functionality	Existing leases continue, new leases unavailable

The transition to read-only mode is triggered automatically when the system detects conditions that make write operations unsafe or impossible. Common triggers include loss of majority quorum in a clustered deployment, storage backend connectivity issues, or explicit administrative commands during maintenance windows.

During read-only mode, the system maintains full audit logging for all operations, including failed write attempts. This ensures complete visibility into system access patterns and helps identify when normal operations can safely resume. The audit system operates from local buffers and queues entries for later persistence when write access is restored.

Critical to read-only mode operation is the handling of dynamic secrets and lease renewals. Existing leases can be renewed using cached policy information, preventing premature expiration of active credentials. However, new dynamic secret generation is suspended since it requires write access to create lease records and configure backend credentials.

### Operational Principle: Transparent Degradation

Clients experience read-only mode as increased latency for write operations rather than service unavailability. The system returns explicit error codes and retry guidance, allowing well-designed clients to implement appropriate fallback behavior.

The system provides detailed status information during read-only mode, including the specific failure conditions that triggered degradation and estimated time until full service restoration. This information helps operations teams prioritize recovery efforts and helps client applications make informed decisions about retry strategies.

### Reduced Functionality Scenarios

Beyond simple read-only mode, the system may need to operate with various subsystems disabled or degraded. Each reduced functionality scenario maintains a specific subset of capabilities while clearly documenting what operations are unavailable and why.

Degraded Subsystem	Available Functionality	Unavailable Functionality	Typical Causes
Dynamic secret engine	Static secret access, policy evaluation	New credential generation, lease renewals	Backend connectivity issues, credential store failures
Authentication engine	Token validation from cache	New token creation, policy updates	Auth database unavailable, policy store corruption
Encryption engine	Decrypt cached DEKs	Key rotation, new DEK generation	KMS unavailable, master key access issues
Audit logging	Operation execution continues	Compliance reporting, log analysis	Log storage full, audit database unavailable
Cluster consensus	Local node operations	Multi-node coordination, leadership changes	Network partitions, node failures

Dynamic secret engine degradation significantly impacts applications that rely on just-in-time credential generation. The system extends existing lease expiration times where possible and provides clear guidance about when normal credential generation will resume. Applications must implement fallback mechanisms for extended use of existing credentials.

Authentication engine degradation affects new user onboarding and policy changes but allows continued operation for existing authenticated clients. Token validation continues using cached policy information, though

policy updates are deferred until full functionality is restored.

Encryption engine degradation prevents key rotation activities and new secret encryption but allows continued access to existing secrets using cached decryption keys. This scenario often occurs during KMS provider outages and can be sustained for extended periods without impacting secret retrieval operations.

### **Decision: Degraded Mode Granularity**

- **Context:** Different system components can fail independently, requiring fine-grained control over which functionality remains available during partial failures
- **Options Considered:**
  - Binary availability (full service or complete shutdown)
  - Component-level degradation (individual subsystems can be disabled)
  - Operation-level degradation (specific operations can be disabled independently)
- **Decision:** Component-level degradation with clear capability boundaries between subsystems
- **Rationale:** Provides meaningful partial functionality without excessive complexity, maps naturally to system architecture, allows operations teams to understand and manage degraded states effectively
- **Consequences:** Enables continued service during partial failures with predictable behavior but requires careful dependency management and clear error reporting to clients

### **Emergency Security Procedures**

When the system detects potential security breaches or operating conditions that might compromise secret safety, it implements emergency security procedures that prioritize data protection over service availability. These procedures represent the system's final defense mechanisms when normal security controls may be insufficient.

Security Event	Detection Triggers	Emergency Response	Service Impact
Potential key compromise	Multiple decryption failures, unusual key access patterns	Immediate key rotation, revoke all tokens	Temporary service disruption during key replacement
Unauthorized access patterns	Failed authentication spikes, unusual geographic access	Rate limiting, IP blocking, enhanced audit logging	Legitimate users may experience delays
Storage integrity issues	Checksum failures, unexpected data modifications	Read-only mode, enhanced validation	Write operations suspended
Time synchronization loss	Clock skew detection, NTP failure	Conservative token validation, extended grace periods	Some authentication requests may fail
Administrative account compromise	Unusual admin activity, policy changes outside change windows	Require additional authentication, suspend admin operations	Administrative functions temporarily restricted

The most critical emergency procedure involves suspected master key compromise. The system immediately initiates emergency key rotation procedures, generates new master keys from fresh entropy sources, and begins re-encrypting all stored secrets. During this process, service availability may be completely suspended to prevent access using potentially compromised keys.

Suspected authentication system compromise triggers comprehensive token revocation and enhanced authentication requirements. All active tokens are invalidated, and new authentication requires additional verification steps such as out-of-band confirmation or multi-factor authentication.

Storage integrity emergencies place the system in a protective mode where all write operations are suspended and read operations include enhanced validation. The system maintains detailed forensic logs of all access attempts during integrity emergencies to support later investigation.

### Security Principle: Emergency Conservatism

During emergency security procedures, the system errs on the side of denying legitimate access rather than potentially allowing unauthorized access. Service availability is secondary to maintaining the confidentiality and integrity of stored secrets.

Emergency procedures include automatic notification mechanisms to alert security teams and system administrators. These notifications include detailed context about the triggering conditions and recommended manual intervention steps. The system maintains emergency contact lists and escalation procedures for different types of security events.

Recovery from emergency security procedures requires explicit administrative action after investigation confirms the system is safe to return to normal operation. This prevents the system from automatically resuming normal operations if the emergency conditions recur or if manual intervention is needed to address underlying security issues.

## Implementation Guidance

This section provides practical implementation guidance for building robust error handling and recovery capabilities into the secret management system. The focus is on creating production-ready error handling that maintains security guarantees even during failure conditions.

## Technology Recommendations

Component	Simple Option	Advanced Option
Error Handling Framework	Standard library error wrapping (Go: <code>fmt.Errorf</code> , Python: <code>raise from</code> )	Structured error library ( <a href="https://github.com/pkg/errors">github.com/pkg/errors</a> , <code>structlog</code> )
Health Check System	HTTP endpoint with basic service status	Comprehensive health monitoring ( <code>go-health</code> , <code>healthcheck</code> )
Circuit Breaker	Simple timeout and retry logic	Production circuit breaker ( <code>hystrix-go</code> , <code>circuit-breaker</code> )
Backup System	File-based backups with cron scheduling	Enterprise backup solution (Velero, Restic with encryption)
Monitoring and Alerting	Log file monitoring with grep	Full observability stack (Prometheus, Grafana, AlertManager)
Graceful Shutdown	Signal handling with context cancellation	Graceful shutdown library ( <code>graceful</code> , <code>shutdown-manager</code> )

## File Structure for Error Handling

```
internal/
  errors/
    errors.go          ← Custom error types and utilities
    codes.go           ← Error classification and HTTP status mapping
    recovery.go        ← Recovery procedure implementations
  health/
    checker.go         ← Health check implementations
    status.go          ← System status tracking
  backup/
    backup.go          ← Backup coordination and scheduling
    restore.go         ← Restoration procedures
    validation.go      ← Post-recovery validation
  graceful/
    shutdown.go        ← Graceful degradation implementation
    readonly.go         ← Read-only mode coordination
```

## Core Error Handling Infrastructure

```
package errors

import (
    "fmt"
    "net/http"
    "time"
)

// ErrorCode represents different categories of errors for appropriate handling

type ErrorCode string

const (
    ErrCodeStorageFailure     ErrorCode = "STORAGE_FAILURE"
    ErrCodeEncryptionFailure ErrorCode = "ENCRYPTION_FAILURE"
    ErrCodeNetworkFailure    ErrorCode = "NETWORK_FAILURE"
    ErrCodeKeyCorruption      ErrorCode = "KEY_CORRUPTION"
    ErrCodeAuthFailure        ErrorCode = "AUTH_FAILURE"
    ErrCodeReadOnlyMode       ErrorCode = "READ_ONLY_MODE"
)

// SecretManagementError provides structured error information for proper handling

type SecretManagementError struct {
    Code     ErrorCode
    Message  string
    Cause    error
    Timestamp time.Time
    Context  map[string]interface{}
    Recoverable bool
}
```

GO

```
    RetryAfter  time.Duration

}

func (e *SecretManagementError) Error() string {
    return fmt.Sprintf("[%s] %s: %v", e.Code, e.Message, e.Cause)
}

// NewStorageError creates an error for storage-related failures

func NewStorageError(message string, cause error, recoverable bool) *SecretManagementError {
    // TODO 1: Create SecretManagementError with ErrCodeStorageFailure

    // TODO 2: Set appropriate retry delay based on error type

    // TODO 3: Include storage-specific context information

    // TODO 4: Determine if error is recoverable based on cause
}

// NewEncryptionError creates an error for encryption-related failures

func NewEncryptionError(message string, cause error) *SecretManagementError {
    // TODO 1: Create SecretManagementError with ErrCodeEncryptionFailure

    // TODO 2: Mark as non-recoverable (encryption errors are usually permanent)

    // TODO 3: Include key version and algorithm context

    // TODO 4: Set timestamp for audit trail
}

// ToHTTPStatus maps error codes to appropriate HTTP status codes

func (e *SecretManagementError) ToHTTPStatus() int {
    // TODO 1: Map ErrCodeStorageFailure to 503 Service Unavailable

    // TODO 2: Map ErrCodeEncryptionFailure to 500 Internal Server Error

    // TODO 3: Map ErrCodeAuthFailure to 401 Unauthorized or 403 Forbidden

    // TODO 4: Map ErrCodeReadOnlyMode to 503 Service Unavailable
}
```

```
// TODO 5: Default to 500 for unmapped error codes  
}
```

## Health Check and Status Monitoring

```
package health

import (
    "context"
    "sync"
    "time"
)

// ComponentStatus represents the health status of a system component

type ComponentStatus string

const (
    StatusHealthy    ComponentStatus = "healthy"
    StatusDegraded   ComponentStatus = "degraded"
    StatusUnhealthy  ComponentStatus = "unhealthy"
    StatusUnknown    ComponentStatus = "unknown"
)

// HealthCheck defines the interface for component health checking

type HealthCheck interface {
    Name() string
    Check(ctx context.Context) (ComponentStatus, error)
    Timeout() time.Duration
}

// SystemHealth tracks the overall health of the secret management system

type SystemHealth struct {
    checks      map[string]HealthCheck
}
```

GO

```
lastResults map[string]ComponentStatus

mutex sync.RWMutex

checkInterval time.Duration

}

func NewSystemHealth(checkInterval time.Duration) *SystemHealth {

    return &SystemHealth{

        checks: make(map[string]HealthCheck),

        lastResults: make(map[string]ComponentStatus),

        checkInterval: checkInterval,

    }

}

// RegisterCheck adds a health check for a system component

func (h *SystemHealth) RegisterCheck(check HealthCheck) {

    // TODO 1: Add check to checks map using check.Name() as key

    // TODO 2: Initialize status as StatusUnknown

    // TODO 3: Consider thread safety for concurrent registration

}

// StartMonitoring begins continuous health checking for all components

func (h *SystemHealth) StartMonitoring(ctx context.Context) {

    // TODO 1: Create ticker for periodic health checks

    // TODO 2: Run health checks for all registered components

    // TODO 3: Update lastResults with current status

    // TODO 4: Log status changes and health degradation

    // TODO 5: Handle context cancellation for clean shutdown

}
```

```
// GetOverallStatus determines system-wide health based on component statuses

func (h *SystemHealth) GetOverallStatus() ComponentStatus {

    // TODO 1: Read all component statuses with appropriate locking

    // TODO 2: Return StatusUnhealthy if any critical component is unhealthy

    // TODO 3: Return StatusDegraded if any component is degraded

    // TODO 4: Return StatusHealthy only if all components are healthy

}

// StorageHealthCheck implements health checking for storage backends

type StorageHealthCheck struct {

    storage Backend

    timeout time.Duration

}

func (s *StorageHealthCheck) Check(ctx context.Context) (ComponentStatus, error) {

    // TODO 1: Create context with timeout for storage operation

    // TODO 2: Attempt simple read operation (like reading a health check key)

    // TODO 3: Measure response time and compare to thresholds

    // TODO 4: Return StatusHealthy for fast responses, StatusDegraded for slow responses

    // TODO 5: Return StatusUnhealthy for failed operations

}
```

## Backup and Recovery Implementation

```
package backup

import (
    "context"
    "crypto/sha256"
    "io"
    "time"
)

// BackupMetadata contains information about a backup for restoration

type BackupMetadata struct {
    Timestamp     time.Time
    BackupID      string
    Components    []string
    Checksum      []byte
    EncryptionKey string
    Consistency   ConsistencyLevel
}

type ConsistencyLevel int

const (
    ConsistencyEventual ConsistencyLevel = iota
    ConsistencyStrong
    ConsistencySnapshot
)

// BackupManager coordinates backup and restore operations across system components
```

GO

```
type BackupManager struct {

    storage          Backend
    encryptionKey   []byte
    backupLocation  string
    retentionDays   int
}

func NewBackupManager(storage Backend, encryptionKey []byte, location string) *BackupManager {
    return &BackupManager{
        storage:          storage,
        encryptionKey:   encryptionKey,
        backupLocation:  location,
        retentionDays:   90, // Default retention
    }
}

// CreateBackup performs a full system backup with consistency guarantees

func (b *BackupManager) CreateBackup(ctx context.Context, components []string)
(*BackupMetadata, error) {

    // TODO 1: Generate unique backup ID with timestamp

    // TODO 2: Create backup directory structure

    // TODO 3: For each component, call component-specific backup procedure

    // TODO 4: Calculate checksums for all backup files

    // TODO 5: Encrypt backup data using provided encryption key

    // TODO 6: Create and store backup metadata

    // TODO 7: Verify backup integrity before returning

    // Hint: Use consistent snapshot timestamp across all components
}
```

```
// RestoreFromBackup restores system state from a specific backup

func (b *BackupManager) RestoreFromBackup(ctx context.Context, backupID string) error {

    // TODO 1: Load and validate backup metadata

    // TODO 2: Verify backup integrity using stored checksums

    // TODO 3: Decrypt backup data using stored encryption key

    // TODO 4: Stop all write operations (enter read-only mode)

    // TODO 5: Restore components in dependency order

    // TODO 6: Validate restored data consistency

    // TODO 7: Restart services and exit read-only mode

    // Hint: Implement rollback capability if restoration fails

}

// ValidateBackupIntegrity checks that a backup can be successfully restored

func (b *BackupManager) ValidateBackupIntegrity(ctx context.Context, backupID string) error {
    // TODO 1: Load backup metadata and verify structure

    // TODO 2: Calculate checksums for all backup files

    // TODO 3: Compare calculated checksums with stored checksums

    // TODO 4: Attempt to decrypt a sample of backup data

    // TODO 5: Verify component-specific validation (e.g., secret decryption)

    // TODO 6: Return detailed error information for any validation failures

}

// CleanupExpiredBackups removes backups older than retention policy

func (b *BackupManager) CleanupExpiredBackups(ctx context.Context) error {
    // TODO 1: List all available backups with timestamps

    // TODO 2: Identify backups older than retention period

    // TODO 3: For each expired backup, verify it's safe to delete
```

```
// TODO 4: Remove backup files and metadata  
  
// TODO 5: Update backup indexes and cleanup empty directories  
  
// Hint: Keep minimum number of backups regardless of age  
}
```

## Graceful Degradation Implementation

```
package graceful

import (
    "context"
    "sync/atomic"
    "time"
)

// DegradationLevel represents the current operational mode of the system

type DegradationLevel int32

const (
    FullOperation DegradationLevel = iota
    ReadOnlyMode
    EmergencyMode
    MaintenanceMode
)

// GracefulDegradation manages system operation during failure conditions

type GracefulDegradation struct {

    currentLevel      int32 // atomic access to DegradationLevel

    emergencyTrigger chan struct{}`

    statusCallbacks   []func(DegradationLevel)

    healthChecker     *SystemHealth
}

func NewGracefulDegradation(healthChecker *SystemHealth) *GracefulDegradation {
    return &GracefulDegradation{`
```

GO

```
        currentLevel:      int32(FullOperation),  
  
        emergencyTrigger: make(chan struct{}, 1),  
  
        healthChecker:    healthChecker,  
  
    }  
  
}  
  
  
// GetCurrentLevel returns the current degradation level thread-safely  
  
func (g *GracefulDegradation) GetCurrentLevel() DegradationLevel {  
  
    return DegradationLevel(atomic.LoadInt32(&g.currentLevel))  
  
}  
  
  
// SetDegradationLevel changes the operational mode and notifies all listeners  
  
func (g *GracefulDegradation) SetDegradationLevel(level DegradationLevel, reason string) {  
  
    // TODO 1: Use atomic operation to set new degradation level  
  
    // TODO 2: Log degradation level change with timestamp and reason  
  
    // TODO 3: Call all registered status change callbacks  
  
    // TODO 4: Update system metrics and monitoring  
  
    // TODO 5: Send notifications to operations team for significant changes  
  
}  
  
  
// MonitorSystemHealth continuously monitors health and adjusts degradation level  
  
func (g *GracefulDegradation) MonitorSystemHealth(ctx context.Context) {  
  
    ticker := time.NewTicker(30 * time.Second)  
  
    defer ticker.Stop()  
  
  
for {  
  
    select {  
  
        case <-ctx.Done():  
  
            return  
    }  
}
```

```
        case <-ticker.C:

            // TODO 1: Get overall system health from health checker

            // TODO 2: Evaluate if current degradation level is appropriate

            // TODO 3: If storage is unhealthy, move to read-only mode

            // TODO 4: If critical components fail, move to emergency mode

            // TODO 5: If health improves, consider restoring higher service level

        case <-g.emergencyTrigger:

            // TODO 1: Immediately set EmergencyMode degradation level

            // TODO 2: Log emergency trigger with full context

            // TODO 3: Send high-priority alerts to operations team

        }

    }

}

// IsOperationAllowed checks if a specific operation is permitted at current degradation
level

func (g *GracefulDegradation) IsOperationAllowed(operation string) bool {

    currentLevel := g.GetCurrentLevel()

    switch currentLevel {

    case FullOperation:

        return true

    case ReadOnlyMode:

        // TODO 1: Allow secret retrieval operations

        // TODO 2: Allow policy evaluation operations

        // TODO 3: Block secret creation/update operations

        // TODO 4: Block dynamic secret generation

        // TODO 5: Allow lease renewals but not new lease creation
    }
}
```

```

        case EmergencyMode:

            // TODO 1: Allow only essential read operations

            // TODO 2: Block all write operations

            // TODO 3: Block all administrative operations

            // TODO 4: Allow health check operations

        case MaintenanceMode:

            // TODO 1: Block all client operations

            // TODO 2: Allow administrative and maintenance operations

            // TODO 3: Allow backup and restore operations

        default:

            return false
    }

}

// TriggerEmergency immediately places system in emergency mode

func (g *GracefulDegradation) TriggerEmergency(reason string) {

    // TODO 1: Log emergency trigger with full context and reason

    // TODO 2: Send signal to emergency trigger channel (non-blocking)

    // TODO 3: Generate audit event for emergency mode activation

    // TODO 4: Update monitoring metrics for emergency state

}

```

## Milestone Checkpoints

### After Milestone 1 (Encryption Engine):

- Verify error handling for key corruption by corrupting a DEK file and confirming graceful failure
- Test backup and restore of encrypted secrets with key rotation
- Confirm system fails safely when master key is unavailable

### After Milestone 2 (Authentication Engine):

- Test graceful degradation when auth database becomes unavailable

- Verify token validation continues using cached policies in read-only mode
- Confirm audit logging continues during authentication failures

#### After Milestone 3 (Dynamic Secrets):

- Test behavior when backend databases become unavailable
- Verify lease extension during degraded mode
- Confirm revocation queue persistence across system restarts

#### After Milestone 4 (High Availability):

- Test cluster behavior during network partitions
- Verify automatic failover and read-only mode for minority partitions
- Test backup and restore procedures across clustered deployment

### Common Debugging Scenarios

Symptom	Likely Cause	Diagnostic Steps	Resolution
System stuck in read-only mode	Storage backend connectivity issues	Check storage health endpoints, verify network connectivity	Restore storage connectivity, manually exit read-only mode
Backup restoration fails	Encryption key mismatch or corruption	Verify key shares, test key reconstruction	Restore keys from separate backup, regenerate corrupted keys
Health checks failing intermittently	Network timeouts or resource exhaustion	Monitor resource usage, check network latency	Adjust health check timeouts, scale resources
Emergency mode triggered unexpectedly	Security monitoring false positives	Review audit logs for trigger events	Tune security monitoring thresholds, add exception rules
Graceful shutdown hangs	Background processes not responding to context cancellation	Check goroutine dumps, identify blocking operations	Add timeout enforcement, improve context handling

### Testing Strategy

**Milestone(s):** This section spans all four milestones, with specific testing approaches for each: security testing for encrypted storage (Milestone 1), access control validation (Milestone 2), dynamic secret lifecycle testing (Milestone 3), and cluster consensus testing (Milestone 4).

Testing a secret management system requires a fundamentally different approach than testing typical applications. Think of it like testing a bank vault system — you're not just verifying that the safe opens when given the correct combination, but also that it absolutely refuses to open for any incorrect combination, that the steel walls resist drilling attempts, that the alarm system triggers under all threat scenarios, and that the backup power systems work during emergencies. In secret management, a single security failure can compromise an entire organization's infrastructure, making comprehensive testing not just important but absolutely critical.

The testing strategy for our secret management system operates on three complementary levels. **Security testing** forms the foundation, continuously verifying that our cryptographic implementations, access controls, and audit mechanisms work correctly under both normal and adversarial conditions. **Milestone checkpoints** provide structured validation points as we build each component, ensuring that fundamental behaviors work correctly before adding complexity. **Integration testing** validates the complete system workflows, testing how all components interact to deliver secure secret management capabilities to real applications.

This multi-layered approach reflects the **defense in depth** principle — if one testing layer misses a vulnerability, the other layers should catch it. We test not just the happy path where everything works correctly, but also the adversarial scenarios where attackers attempt to exploit weaknesses, the chaos scenarios where infrastructure fails unpredictably, and the operational scenarios where human operators make mistakes under pressure.

## Security Testing

Security testing for a secret management system goes far beyond checking that basic encryption works. Think of it like testing a corporate security system — you need to verify not just that authorized employees can enter the building, but that unauthorized individuals cannot gain access even with sophisticated attack techniques, that security cameras capture all relevant events, and that alarm systems function correctly during emergencies.

**Encryption verification** forms the cornerstone of our security testing approach. We must validate that our envelope encryption implementation provides the security properties we claim, that key rotation works without exposing plaintext secrets, and that our cryptographic implementations resist known attacks.

The encryption testing matrix covers multiple dimensions of verification:

Test Category	Security Property	Validation Method	Failure Impact
Ciphertext Analysis	Semantic Security	Statistical randomness tests on encrypted output	Reveals information about plaintext secrets
Key Isolation	Key Independence	Decrypt with wrong DEK version, verify failure	Cross-version secret exposure
Nonce Uniqueness	Replay Resistance	Detect nonce reuse across encryptions	Authentication bypass via replay
Authentication Tag	Tamper Detection	Modify ciphertext, verify decryption failure	Undetected data corruption
Key Derivation	Brute Force Resistance	Time password cracking attempts	Weak master key compromise
Memory Protection	Information Leakage	Scan memory dumps for plaintext keys	Key extraction from memory

**Cryptographic validation** requires testing our encryption engine against known attack vectors and edge cases. We implement test cases that attempt to exploit common vulnerabilities in envelope encryption systems, such as key confusion attacks where attackers trick the system into using the wrong decryption key, or padding oracle attacks that reveal information through error messages.

The key rotation security tests deserve particular attention. During key rotation, we temporarily have both old and new encryption keys in memory, creating a window where compromise of either key could affect system security. Our tests simulate various failure scenarios during rotation — such as power loss, network partitions, or process crashes — and verify that the system maintains security guarantees in all cases.

### Decision: Continuous Cryptographic Testing

- **Context:** Cryptographic bugs can be subtle and may not manifest in functional testing, but can completely compromise security
- **Options Considered:** Manual periodic reviews, automated test suite with crypto vectors, formal verification
- **Decision:** Comprehensive automated test suite with known crypto test vectors plus property-based testing
- **Rationale:** Automated testing catches regressions immediately, known vectors validate against published standards, property-based testing finds edge cases
- **Consequences:** Higher confidence in crypto implementation, but requires significant test infrastructure investment

**Access control validation** tests our authentication and authorization mechanisms under both normal and adversarial conditions. This includes testing for **constant-time comparison** vulnerabilities, where timing

differences in token validation could allow attackers to guess valid tokens, and policy bypass attempts where malformed requests might circumvent access controls.

Our access control test scenarios include:

Attack Vector	Test Method	Expected Outcome	Security Concern
Token Guessing	Brute force token validation with timing analysis	Constant response time regardless of token validity	Timing side-channel reveals valid token patterns
Path Traversal	Request secrets with manipulated paths (./, //)	Access denied for all unauthorized paths	Directory traversal bypasses path-based ACLs
Policy Confusion	Submit requests with conflicting policy claims	Deny access when policies conflict	Ambiguous policies might default to allow
Token Replay	Reuse expired or revoked tokens	Reject all invalid tokens consistently	Stale tokens continue to grant access
Privilege Escalation	Attempt to access admin endpoints with user tokens	Enforce role separation strictly	Users gain administrative privileges
Session Fixation	Predict or control token generation	Cryptographically random, unpredictable tokens	Attackers can guess or control user tokens

The **audit logging security tests** verify that our tamper-evident logging correctly captures all security-relevant events and resists attempts to modify or delete audit records. We test scenarios where attackers gain partial system access and attempt to cover their tracks by modifying log files.

**Penetration testing** simulates real-world attack scenarios against our deployed secret management system. These tests assume that attackers have gained some level of system access — perhaps through compromised application credentials or network position — and attempt to escalate their access to retrieve secrets they shouldn't have.

Common penetration testing scenarios include:

1. **Compromise Recovery Testing:** Simulate discovery of a leaked token and verify that token revocation prevents further access
2. **Insider Threat Simulation:** Test system behavior when legitimate users attempt to access secrets outside their authorized scope
3. **Network Attack Simulation:** Test TLS configuration and certificate validation under man-in-the-middle attacks
4. **Side-Channel Analysis:** Monitor system resource usage patterns to detect information leakage through CPU, memory, or network timing
5. **Social Engineering Simulation:** Test operational procedures for responding to requests for emergency access or system recovery

**Chaos security testing** introduces failures during security-critical operations to verify that our system maintains security properties even under adverse conditions. For example, we might simulate power loss during key rotation, network partitions during authentication, or storage corruption during secret updates.

The key insight from chaos security testing is that security vulnerabilities often emerge not from normal operation, but from the complex interactions between security mechanisms and failure recovery procedures. A system might correctly encrypt secrets during normal operation, but accidentally write plaintext to logs during error recovery.

## Milestone Checkpoints

Each milestone introduces new security-critical functionality that must be thoroughly validated before proceeding to the next phase. Think of these checkpoints like security clearance levels — you can't proceed to more sensitive information until you've proven that the current level is completely secure.

### Milestone 1 Checkpoint: Encrypted Secret Storage

The first milestone establishes the foundation of our security model through envelope encryption. The checkpoint validation must verify that secrets are never stored in plaintext and that our key management procedures work correctly.

Critical validation points for Milestone 1:

Validation Category	Test Description	Success Criteria	Failure Symptoms
Envelope Encryption	Store secret, examine storage backend directly	No plaintext visible in stored data	Secrets readable without decryption
Key Rotation	Rotate DEK, verify old and new versions decrypt correctly	Both versions accessible, no downtime	Secret access failures or plaintext exposure
Secret Versioning	Create multiple versions, verify independent retrieval	Each version returns correct value	Version confusion or data corruption
Master Key Protection	Examine memory dumps and storage for key material	Keys only in memory when actively used	Persistent key storage or memory leaks
Recovery Testing	Restart system, verify access to existing secrets	All secrets accessible after restart	Data loss or corruption after restart
Storage Backend Independence	Switch storage backends, verify secret accessibility	Seamless backend migration	Backend-specific secret corruption

The envelope encryption validation requires careful testing of the key hierarchy. We must verify that the master key never appears in persistent storage except when encrypted by external systems (for auto-unseal scenarios), that data encryption keys are properly versioned and rotated independently, and that secret values remain accessible across key rotation events.

**Performance baseline testing** establishes acceptable latency and throughput characteristics for the encryption engine. Secret management systems often become bottlenecks in application deployment pipelines, so understanding performance characteristics helps with capacity planning.

Key performance metrics for Milestone 1:

- **Encryption latency:** Time from secret submission to storage confirmation
- **Decryption latency:** Time from secret request to plaintext delivery
- **Key rotation duration:** End-to-end time for DEK rotation with large secret volumes
- **Memory usage patterns:** Peak memory consumption during encryption operations
- **Storage amplification:** Ratio of encrypted storage size to plaintext size

## Milestone 2 Checkpoint: Access Policies & Authentication

The second milestone introduces authentication and authorization capabilities that protect the encrypted secrets created in Milestone 1. The validation must verify that only properly authenticated and authorized requests can access secrets.

Authentication validation focuses on ensuring that our token-based and mTLS authentication mechanisms correctly identify clients and resist common attacks:

Authentication Test	Attack Simulation	Validation Method	Security Property
Token Validation	Submit requests with forged tokens	Reject all invalid tokens with constant timing	Cryptographic token integrity
Token Expiration	Use expired tokens after TTL	Access denied for expired credentials	Temporal access control
mTLS Certificate	Present invalid or expired client certificates	Reject connections with invalid certificates	Certificate-based identity
Replay Attack	Reuse intercepted authentication tokens	Detect and reject replayed tokens	Replay resistance
Timing Analysis	Measure token validation response times	Constant time regardless of token validity	Side-channel resistance
Concurrent Sessions	Multiple simultaneous sessions with same token	Proper session isolation and tracking	Session security

Authorization validation ensures that our path-based policy engine correctly enforces access controls even with complex policy rules and edge cases:

Authorization Test	Policy Scenario	Request Pattern	Expected Outcome
Exact Path Match	Policy allows <code>secret/app/db</code>	Request <code>secret/app/db</code>	Access granted
Wildcard Matching	Policy allows <code>secret/app/*</code>	Request <code>secret/app/api-key</code>	Access granted
Path Traversal	Policy allows <code>secret/app/*</code>	Request <code>secret/app/.../admin/key</code>	Access denied
Nested Wildcards	Policy allows <code>secret/**</code>	Request <code>secret/deep/nested/path</code>	Access granted according to recursive pattern
Policy Conflicts	Multiple policies with different permissions	Requests matching overlapping policies	Secure default (deny) when ambiguous
Missing Policies	No policy covers requested path	Any request to uncovered path	Default deny behavior

**Audit logging validation** ensures that all security-relevant events are properly captured and that the audit trail provides sufficient information for security investigations:

- **Event completeness:** All authentication attempts, authorization decisions, and secret access events logged
- **Event integrity:** Audit records resist tampering and provide cryptographic integrity guarantees
- **Event correlation:** Related events (authentication → authorization → secret access) can be traced through request IDs
- **Performance impact:** Audit logging doesn't significantly degrade system performance
- **Storage management:** Audit logs rotate properly and don't exhaust available storage

### Milestone 3 Checkpoint: Dynamic Secrets

The third milestone adds dynamic secret generation capabilities, introducing time-based security properties through lease management. Validation must verify that credentials are properly generated, tracked, and revoked.

Dynamic secret generation testing focuses on the credential lifecycle and backend integration:

Backend Type	Generation Test	Revocation Test	Renewal Test
Database	Create unique user with limited privileges	Remove user and verify access denied	Extend user TTL and verify continued access
AWS IAM	Generate temporary access keys with scoped permissions	Delete IAM user and verify API calls fail	Renew access keys before expiration
SSH	Generate signed certificate with time limits	Verify certificate invalid after TTL	Issue renewed certificate with extended validity
PKI	Issue client certificate with usage constraints	Revoke certificate via CRL	Renew certificate with same subject but new validity period

**Lease management validation** ensures that the time-based security model works correctly:

- **TTL enforcement:** Credentials become invalid exactly when leases expire
- **Renewal mechanics:** Renewable leases extend correctly, non-renewable leases reject renewal attempts
- **Revocation timeliness:** Explicit revocation requests disable credentials within acceptable time bounds
- **Cleanup completeness:** Revocation removes all traces of credentials from target systems
- **Failure recovery:** Failed revocation attempts retry with exponential backoff until successful

The **lease reaper testing** validates the background processes that maintain system security by cleaning up expired credentials:

1. Create multiple dynamic secrets with different TTLs
2. Wait for natural expiration of some credentials
3. Verify that expired credentials no longer grant access to target systems
4. Confirm that lease records are properly cleaned from internal storage
5. Test reaper behavior during system restart and recovery scenarios

#### Milestone 4 Checkpoint: Unsealing & High Availability

The final milestone introduces distributed operation and master key protection through Shamir's secret sharing. Validation must verify that the cluster maintains security and availability guarantees under various failure scenarios.

**Shamir's secret sharing validation** ensures that our threshold scheme correctly protects the master key:

Share Configuration	Test Scenario	Validation Method	Security Property
3-of-5 threshold	Reconstruct with exactly 3 shares	Successfully unseal system	Minimum threshold suffices
2-of-5 attempt	Attempt reconstruction with only 2 shares	Reconstruction fails completely	Insufficient shares provide no information
Share corruption	Modify one share before reconstruction	Detect corruption and reject invalid share	Share integrity validation
Share ordering	Present shares in different orders	Reconstruction produces identical results	Order independence
Share storage	Examine individual share contents	Shares reveal no information about master key	Information-theoretic security
Recovery procedure	Simulate loss of shares, regenerate master key	Complete system recovery with new shares	Disaster recovery capabilities

**High availability validation** tests the distributed consensus mechanisms and leader election procedures:

- **Leader election:** Single leader emerges after cluster startup, leadership transfers correctly during failures
- **Split-brain prevention:** Network partitions don't result in multiple active leaders accepting conflicting writes
- **Data consistency:** All nodes converge to identical state after network partition recovery
- **Client failover:** Clients automatically redirect to new leader after failover events
- **Performance impact:** Consensus overhead doesn't significantly degrade operation latency

**Auto-unseal integration testing** validates the cloud KMS integration for automated system startup:

1. Configure auto-unseal with mock KMS provider
2. Seal the system and verify it requires external key to operate
3. Test auto-unseal during normal startup procedures
4. Simulate KMS failures during unseal attempts
5. Verify fallback to manual unseal when auto-unseal unavailable
6. Test key rotation scenarios with external KMS keys

## Integration Testing

Integration testing validates that all components work together correctly to provide secure secret management capabilities to real applications. Think of this as testing the entire bank vault system — not just the individual components like the safe, alarm system, and access controls, but how they all work together to protect valuable assets under realistic conditions.

**End-to-end workflow testing** simulates complete application interactions with our secret management system, from initial authentication through secret retrieval and dynamic credential generation. These tests use realistic application patterns and data volumes to surface integration issues that might not appear in unit tests.

Primary integration test workflows include:

Workflow	Description	Components Tested	Failure Modes
Application Bootstrap	App authenticates and retrieves configuration secrets	Auth engine, encryption engine, policy evaluation	Auth failures prevent app startup
Database Connection	App requests dynamic DB credentials for data access	Dynamic secret engine, lease management, DB backend	Connection failures or credential conflicts
Secret Rotation	Background process rotates static secrets used by apps	Encryption engine, versioning, client notification	App service disruption during rotation
Emergency Access	Operator manually accesses secrets during incident response	Unsealing, emergency policies, audit logging	Security versus availability trade-offs
Cluster Failover	Primary node fails, backup takes over without service disruption	HA cluster, leader election, data replication	Client connection disruption or data loss
Backup and Recovery	Complete system restore from encrypted backups	All components, unsealing, data verification	Incomplete recovery or data corruption

**Backend plugin testing** validates the integration between our dynamic secret engine and external systems that provide credentials. Each backend type has different integration patterns and failure modes that require specific testing approaches.

Database backend integration testing covers multiple database platforms and credential patterns:

- **PostgreSQL integration:** Test user creation, privilege assignment, connection validation, and cleanup procedures
- **MySQL integration:** Validate account creation with proper grants, password policies, and revocation procedures
- **MongoDB integration:** Test role-based access control, database-specific permissions, and user removal
- **Connection pooling:** Verify that credential generation doesn't overwhelm database connection limits
- **Schema permissions:** Test that generated users receive only necessary privileges for their intended use case

Cloud provider backend testing validates integration with major cloud platforms:

- **AWS IAM integration:** Test temporary access key generation, policy attachment, and key revocation
- **Azure AD integration:** Validate service principal creation, role assignment, and cleanup procedures
- **Google Cloud IAM integration:** Test service account key generation, project-level permissions, and key deletion

- **Resource isolation:** Verify that generated credentials can't access resources outside intended scope

**Load testing** evaluates system behavior under realistic traffic patterns and helps identify performance bottlenecks or resource exhaustion issues. Secret management systems often experience bursty traffic patterns — for example, during application deployment windows when many services simultaneously request credentials.

Load testing scenarios include:

Scenario	Traffic Pattern	Duration	Success Criteria
Steady State	Constant rate of secret requests	8 hours	< 100ms p99 latency, no errors
Deployment Burst	10x normal traffic for short periods	30 minutes	Graceful degradation, no failures
Key Rotation	Background key rotation during normal traffic	2 hours	No client impact, consistent latency
Dynamic Secret Storm	Many simultaneous dynamic credential requests	15 minutes	Backend rate limiting, queue management
Cluster Recovery	Traffic resumes after planned failover	1 hour	Automatic client reconnection, no data loss
Memory Pressure	Sustained high memory usage scenarios	4 hours	No memory leaks, graceful resource management

**Security integration testing** validates that our security mechanisms work correctly when multiple components interact under realistic conditions. This includes testing for emergent security vulnerabilities that might arise from component interactions.

Cross-component security scenarios:

- **Token lifecycle across components:** Verify that token expiration in auth engine immediately affects access in all other components
- **Audit trail completeness:** Confirm that operations involving multiple components generate complete audit trails
- **Error message information leakage:** Ensure that error responses don't reveal sensitive information about system internal state
- **Resource exhaustion attacks:** Test system behavior when attackers attempt to exhaust resources across multiple components
- **Privilege boundary enforcement:** Verify that compromise of one component doesn't automatically compromise others

**Monitoring and observability testing** validates that our system provides sufficient visibility for operations teams to maintain security and availability. This includes testing that metrics, logs, and health checks correctly

reflect system state and can support effective incident response.

Observability validation points:

- **Health check accuracy:** Health endpoints correctly reflect component status and dependency health
- **Metric completeness:** Prometheus metrics cover all critical operations and performance characteristics
- **Log correlation:** Related events across components can be traced through correlation IDs
- **Alert threshold accuracy:** Alert conditions trigger at appropriate times without excessive false positives
- **Dashboard effectiveness:** Operational dashboards provide actionable information during incidents

The integration testing phase also validates our **graceful degradation** capabilities — how the system behaves when some components fail but others remain operational. For example, if the dynamic secret engine fails, the system should continue serving static secrets while clearly indicating the reduced functionality.

## Implementation Guidance

The testing strategy implementation provides comprehensive validation for all aspects of our secret management system. This implementation focuses on Go testing frameworks and tools that provide robust security testing capabilities.

### Technology Recommendations:

Testing Category	Simple Option	Advanced Option
Unit Testing	Go standard testing package	Testify framework with assertions
Security Testing	Manual crypto test vectors	Rapid property-based testing
Integration Testing	HTTP client tests against local server	Testcontainers with real databases
Load Testing	Basic goroutine-based load generation	Vegeta or k6 load testing tools
Penetration Testing	Manual security reviews	OWASP ZAP automated scanning
Chaos Testing	Manual failure injection	Chaos Monkey or Litmus chaos engineering

### Recommended Testing File Structure:

```
project-root/
  cmd/
    server/
      main_test.go           ← integration tests
  internal/
    encryption/
      engine.go
      engine_test.go        ← unit tests
      engine_security_test.go ← security-specific tests
    auth/
      engine.go
      engine_test.go
      engine_integration_test.go ← cross-component tests
  dynamic/
    engine.go
    engine_test.go
    backends/
      database_test.go       ← backend-specific tests
  test/
    security/
      crypto_vectors_test.go ← known answer tests
      penetration_test.go    ← security scanning tests
    integration/
      workflows_test.go      ← end-to-end scenarios
      load_test.go           ← performance testing
  testdata/
    policies/               ← test policy files
    certificates/          ← test TLS certificates
    vectors/                ← crypto test vectors
  docker-compose.test.yml   ← test environment setup
```

## Security Testing Infrastructure:

GO

```
// Package security provides comprehensive security testing utilities

// for the secret management system.

package security

import (
    "crypto/rand"
    "crypto/subtle"
    "testing"
    "time"
)

// CryptoTestVector represents a known-answer test for cryptographic operations.

type CryptoTestVector struct {

    Name      string
    Plaintext []byte
    Key       []byte
    Nonce     []byte
    Expected  []byte
    ShouldFail bool
}

// LoadKnownVectors loads standardized crypto test vectors from NIST or other sources.

func LoadKnownVectors(algorithm string) ([]CryptoTestVector, error) {
    // TODO 1: Read test vectors from testdata/vectors/ directory
    // TODO 2: Parse vector format (usually hex-encoded values)
    // TODO 3: Return structured test cases for the specified algorithm
    // TODO 4: Include both positive (should succeed) and negative (should fail) cases
}
```

```
// TimingAttackTest validates constant-time operations by measuring execution times.

type TimingAttackTest struct {

    ValidInput    string
    InvalidInput  string
    Operation     func(string) bool
    Iterations    int
    Tolerance     time.Duration
}

// RunTimingAnalysis executes timing attack tests to detect side-channel vulnerabilities.

func (t *TimingAttackTest) RunTimingAnalysis(test *testing.T) {
    // TODO 1: Execute operation with valid input multiple times, measure duration
    // TODO 2: Execute operation with invalid input multiple times, measure duration
    // TODO 3: Calculate statistical significance of timing differences
    // TODO 4: Fail test if timing difference exceeds tolerance threshold
    // Hint: Use runtime.GC() before timing measurements for consistency
}

// PropertyTest defines properties that should hold for cryptographic operations.

type PropertyTest func(input []byte) bool

// RunPropertyBasedTests executes randomized property testing for crypto operations.

func RunPropertyBasedTests(t *testing.T, property PropertyTest, iterations int) {
    for i := 0; i < iterations; i++ {
        // TODO 1: Generate random input of varying lengths
        // TODO 2: Execute property test function
        // TODO 3: Report failure with specific input that violated property
        // TODO 4: Use crypto/rand for cryptographically secure randomness
    }
}
```

```
}

}

// PenetrationTestSuite coordinates security testing against running system.

type PenetrationTestSuite struct {

    BaseURL     string

    TestToken   string

    AdminToken string

    Client      *http.Client

}

// TestAuthenticationBypass attempts various authentication bypass techniques.

func (p *PenetrationTestSuite) TestAuthenticationBypass(t *testing.T) {

    testCases := []struct {

        name          string

        headers       map[string]string

        expectError  bool

    }{

        // TODO: Add test cases for:

        // - Missing authentication headers

        // - Malformed tokens

        // - Expired tokens

        // - Tokens with invalid signatures

        // - SQL injection in token fields

        // - Directory traversal in token claims

    }

    // TODO 1: Iterate through test cases
```

```
// TODO 2: Make HTTP requests with crafted headers  
  
// TODO 3: Verify that unauthorized requests are rejected  
  
// TODO 4: Check that error responses don't leak sensitive information  
  
}
```

**Milestone Checkpoint Implementation:**

GO

```
// Package checkpoint provides milestone validation utilities.

package checkpoint

import (
    "context"
    "testing"
    "time"

    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

// Milestone1Validator validates encrypted secret storage functionality.

type Milestone1Validator struct {
    server    *SecretServer
    storage   Backend
    testData map[string]string
}

// ValidateEnvelopeEncryption verifies that secrets are properly encrypted at rest.

func (m *Milestone1Validator) ValidateEnvelopeEncryption(t *testing.T) {
    ctx := context.Background()

    testSecret := "super-secret-password"

    // TODO 1: Store a secret through the API
    // TODO 2: Examine the storage backend directly for the encrypted data
    // TODO 3: Verify that plaintext secret does not appear in storage
    // TODO 4: Verify that encrypted data includes proper authentication tags
```

```
// TODO 5: Attempt to decrypt with wrong key and verify it fails

// Expected behavior: No plaintext visible in storage, proper encryption metadata
}

// ValidateKeyRotation tests key rotation without affecting secret accessibility.

func (m *Milestone1Validator) ValidateKeyRotation(t *testing.T) {
    ctx := context.Background()

    // TODO 1: Create several test secrets

    // TODO 2: Trigger key rotation operation

    // TODO 3: Verify all existing secrets remain accessible

    // TODO 4: Create new secret and verify it uses new key version

    // TODO 5: Verify that both old and new key versions work correctly

    // Expected behavior: Zero downtime rotation with version compatibility
}

// Milestone2Validator validates authentication and authorization functionality.

type Milestone2Validator struct {
    server      *SecretServer
    testPolicies map[string]*Policy
    testTokens   map[string]*Token
}

// ValidateAccessControl tests policy enforcement and authentication.

func (m *Milestone2Validator) ValidateAccessControl(t *testing.T) {
    // TODO 1: Create test policies with different path permissions
```

```

// TODO 2: Create test tokens with different policy assignments

// TODO 3: Attempt to access secrets with each token

// TODO 4: Verify that access is granted/denied according to policies

// TODO 5: Test edge cases like path traversal attempts

// Expected behavior: Only authorized requests succeed, proper audit logging

}

// LoadTestValidator provides performance and load testing validation.

type LoadTestValidator struct {

    baseURL      string
    concurrency int
    duration     time.Duration
}

// ValidatePerformance executes load test and validates performance characteristics.

func (l *LoadTestValidator) ValidatePerformance(t *testing.T) {

    // TODO 1: Create multiple goroutines to simulate concurrent clients

    // TODO 2: Execute secret read/write operations for specified duration

    // TODO 3: Measure latency percentiles (p50, p95, p99)

    // TODO 4: Track error rates and timeout conditions

    // TODO 5: Verify that performance meets acceptance criteria

    // Expected behavior: <100ms p99 latency, <1% error rate under normal load

}

```

## Integration Testing Framework:

GO

```
// Package integration provides end-to-end testing capabilities.

package integration

import (
    "context"
    "database/sql"
    "testing"

    "github.com/testcontainers/testcontainers-go"
    "github.com/testcontainers/testcontainers-go/wait"
)

// TestEnvironment manages the complete testing environment including dependencies.

type TestEnvironment struct {

    VaultContainer testcontainers.Container

    DBContainer    testcontainers.Container

    VaultURL       string

    DBURL          string

    AdminToken     string

}

// SetupIntegrationTest creates a complete test environment with all dependencies.

func SetupIntegrationTest(t *testing.T) *TestEnvironment {
    ctx := context.Background()

    // TODO 1: Start PostgreSQL container for database backend testing

    // TODO 2: Start secret management server container

    // TODO 3: Wait for both services to be ready (health checks)
```

```
// TODO 4: Initialize server with test configuration

// TODO 5: Create admin token for test operations

// TODO 6: Return configured test environment

}

// Use testcontainers to manage Docker containers for integration tests

}

// TestCompleteWorkflow validates end-to-end secret management operations.

func TestCompleteWorkflow(t *testing.T) {

    env := SetupIntegrationTest(t)

    defer env.Cleanup()

    // TODO 1: Authenticate with admin token

    // TODO 2: Create policies for different access levels

    // TODO 3: Create application tokens with specific policies

    // TODO 4: Store static secrets using application token

    // TODO 5: Configure dynamic secret backend (database)

    // TODO 6: Generate dynamic credentials

    // TODO 7: Use dynamic credentials to connect to database

    // TODO 8: Verify credential cleanup after lease expiration

    // Expected behavior: Complete workflow succeeds without errors

}

// TestFailoverScenario validates high availability and disaster recovery.

func TestFailoverScenario(t *testing.T) {

    env := SetupClusterEnvironment(t) // 3-node cluster

    defer env.Cleanup()
```

```

    // TODO 1: Store secrets on primary node

    // TODO 2: Kill primary node container

    // TODO 3: Verify that backup node becomes leader

    // TODO 4: Verify that secrets remain accessible through new leader

    // TODO 5: Restart failed node and verify it rejoins cluster

    // Expected behavior: Transparent failover with minimal downtime
}

```

### Debugging and Troubleshooting Guide:

Test Failure Symptom	Likely Cause	Diagnostic Steps	Resolution
Crypto tests fail with "invalid authentication tag"	Wrong key used for decryption or corrupted ciphertext	Check key versioning logic, verify nonce uniqueness	Fix key lookup or nonce generation
Timing tests show significant differences	Non-constant-time comparison in token validation	Profile token validation code, check for early returns	Use <code>subtle.ConstantTimeCompare</code> for all secret comparisons
Integration tests timeout	Service not ready or network connectivity issues	Check container logs, verify port binding	Fix service startup or network configuration
Load tests show degraded performance	Resource exhaustion or inefficient algorithms	Monitor CPU/memory usage, profile hot code paths	Optimize algorithms or increase resource limits
Dynamic secret tests fail	Backend service not configured or credentials invalid	Check backend service logs, verify connection strings	Fix backend configuration or credentials

The comprehensive testing strategy ensures that our secret management system maintains security and reliability guarantees across all operational scenarios. The combination of security-focused testing, milestone

validation, and integration testing provides confidence that the system will protect secrets effectively in production environments.

## Debugging Guide

**Milestone(s):** This section applies to all four milestones, providing diagnostic techniques and troubleshooting procedures essential for developing and maintaining a secret management system. Common implementation bugs span encrypted storage (Milestone 1), access control (Milestone 2), dynamic secrets (Milestone 3), and high availability (Milestone 4).

Think of debugging a secret management system like diagnosing a security breach in a bank vault system. Just as bank security personnel have standardized procedures for investigating access failures, suspicious activities, and system malfunctions, secret management systems require systematic approaches to identify and resolve issues. The critical difference is that debugging must preserve security properties - you cannot simply "dump all state" to diagnose an issue if that would expose sensitive data.

The challenge of debugging secret management systems lies in the intersection of security and observability. Traditional debugging techniques like memory dumps, detailed stack traces, and verbose logging can inadvertently expose the very secrets the system is designed to protect. This creates a unique constraint where diagnostic information must be comprehensive enough to identify problems but sanitized enough to maintain security boundaries.

### Common Implementation Bugs

Secret management systems exhibit predictable failure patterns due to the complexity of cryptographic operations, distributed consensus, and strict security requirements. Understanding these common bugs helps developers recognize symptoms quickly and apply targeted fixes rather than spending hours debugging cryptic failures.

### Encryption and Key Management Bugs

The most frequent bugs in secret management systems involve improper handling of cryptographic keys and encryption operations. These bugs are particularly dangerous because they often appear to work correctly during testing but fail catastrophically in production or compromise security without obvious symptoms.

**⚠ Pitfall: Master Key in Memory Exposure** Developers often leave the master key in memory after encryption operations or fail to zero memory after use. This creates a window where process memory dumps or swap files can expose the master key, compromising all encrypted secrets. The master key should be cleared from memory immediately after use and never stored in garbage-collected languages without explicit zeroing.

Bug Pattern	Symptoms	Root Cause	Detection Method	Fix
Uncleared master key	No immediate symptoms; discovered during security audit	Master key remains in process memory after operations	Memory scanning tools; static analysis	Implement <code>SecureMemory</code> zeroing; use defer statements
Wrong key version	<code>DecryptSecret</code> fails with "invalid key version"	Using incorrect <code>DataEncryptionKey</code> version for decryption	Audit logs show version mismatches	Verify <code>GetDEKForVersion</code> maps versions correctly
Missing key rotation	All new secrets fail encryption after rotation	Old <code>DataEncryptionKey</code> marked inactive but no new key generated	Check active DEK count in storage	Implement proper <code>RotateDataEncryptionKey</code> workflow
GCM nonce reuse	Silent corruption; authentication failures	Same nonce used with same key multiple times	Cryptographic analysis; random nonce collisions	Use counter-based nonces or larger random space
Weak entropy source	Predictable encryption keys	Using <code>math/rand</code> instead of <code>crypto/rand</code>	Statistical analysis of generated keys	Replace with cryptographically secure random source

**⚠ Pitfall: Improper Secret Versioning** Many implementations fail to handle secret versioning correctly during key rotation. When `RotateDataEncryptionKey` is called, existing secrets must remain accessible with their original keys while new secrets use the updated key. Developers often assume all secrets use the current key, causing decryption failures for older versions.

## Authentication and Authorization Bugs

Authentication bugs in secret management systems often stem from improper token validation, timing vulnerabilities, or policy evaluation errors. These bugs are particularly critical because they can bypass the entire security model.

**⚠ Pitfall: Timing Attack Vulnerabilities** Token comparison operations that exit early on the first character mismatch create timing side-channels that allow attackers to brute-force valid tokens character by character. The `ValidateTokenConstantTime` function must compare the entire token regardless of where differences occur.

Bug Pattern	Symptoms	Root Cause	Detection Method	Fix
Variable-time comparison	Successful token brute-force attacks	Early exit in token comparison	Timing analysis; statistical measurement	Implement constant-time comparison
Token cleanup failure	Expired tokens still authenticate	Token expiration not enforced	Check token count growth over time	Add background cleanup process
Policy evaluation bypass	Unauthorized access succeeds	Incorrect path matching logic	Audit failed access attempts	Fix <code>matchesPath</code> wildcard handling
Session fixation	Single token works across different identities	Token not bound to specific identity	Cross-identity token usage detected	Bind tokens to identity in creation
Privilege escalation	Users access restricted paths	Policy inheritance bug	Monitor policy evaluation results	Review <code>AuthorizeRequest</code> logic

**⚠ Pitfall: Overpermissive Path Matching** Path patterns like `secret/*` often match more than intended due to improper wildcard handling. Developers assume `*` matches only single path segments, but implementation bugs can cause it to match multiple segments, effectively granting broader access than intended.

## Dynamic Secret Lifecycle Bugs

Dynamic secret systems are particularly prone to bugs related to lease management, credential cleanup, and race conditions between generation and revocation processes.

**⚠ Pitfall: Revocation Race Conditions** When a lease expires, there's often a race condition between the lease reaper marking the lease as expired and the revocation worker actually cleaning up the credentials. During this window, the credentials may still be valid in the target system even though the vault considers them revoked.

Bug Pattern	Symptoms	Root Cause	Detection Method	Fix
Credential leak after expiry	Database connections persist after lease expiry	Revocation worker failing silently	Monitor active connections vs issued leases	Add revocation retry with alerts
Lease reaper deadlock	No leases ever expire; memory grows unbounded	Lock ordering issue in lease processing	Thread dumps; deadlock detection	Consistent lock ordering; timeouts
TTL calculation overflow	Leases never expire or expire immediately	Integer overflow in time calculations	Check extreme TTL values	Use <code>time.Duration</code> with bounds checking
Backend connection exhaustion	New credential generation fails	Each request creates new database connection	Monitor connection pool metrics	Implement connection reuse
Renewal window bug	Clients cannot renew near-expiry leases	Renewal rejected if remaining TTL too low	Audit renewal rejection logs	Adjust renewal time window logic

**⚠ Pitfall: Failed Revocation Handling** Many implementations assume credential revocation always succeeds and fail to handle cases where the target system is unavailable or returns errors. Failed revocations should be retried with exponential backoff and eventually moved to a dead letter queue for manual intervention.

## High Availability and Consensus Bugs

Distributed secret management systems introduce additional complexity around leader election, data replication, and split-brain prevention. These bugs often manifest as data inconsistencies or service unavailability.

**⚠ Pitfall: Split-Brain Scenarios** When network partitions occur, multiple nodes may believe they are the leader and accept write operations. This creates conflicting state that can be difficult to reconcile and may compromise security if different nodes grant different access to the same secrets.

Bug Pattern	Symptoms	Root Cause	Detection Method	Fix
Multiple active leaders	Conflicting secret versions	Network partition with inadequate quorum	Monitor leadership claims across nodes	Implement strict quorum requirements
Sealed state bypass	Operations succeed on sealed vault	Seal state not checked consistently	Audit operations on sealed nodes	Add seal checks to all operations
Share reconstruction failure	Cannot unseal with valid shares	Corrupted shares or incorrect reconstruction	Verify share checksums	Implement share validation
Auto-unseal infinite loop	Repeated unseal attempts without success	KMS provider returning inconsistent results	Monitor unseal attempt frequency	Add circuit breaker pattern
Consensus timeout issues	Operations hang during cluster changes	Raft timeouts too aggressive for network	Monitor consensus operation latency	Tune timeout values for environment

## Diagnostic Techniques

Effective debugging of secret management systems requires specialized techniques that maintain security boundaries while providing sufficient observability. Traditional debugging approaches must be adapted to avoid exposing sensitive data while still providing actionable information.

### Structured Logging and Audit Trails

The foundation of secret management debugging is comprehensive audit logging that captures security-relevant events without exposing secret values. Every operation should generate audit events that can be analyzed to reconstruct the sequence of events leading to a problem.

**Audit Event Analysis** provides the primary diagnostic capability for secret management systems. The `AuditLogger` captures detailed information about every request, including authentication attempts, policy evaluations, and secret access patterns. Unlike traditional application logs, audit logs must be tamper-evident and cannot be disabled even during debugging.

Event Type	Required Fields	Diagnostic Value	Privacy Considerations
Authentication	Identity, AuthMethod, SourceIP, Result	Identify failed login patterns	Hash sensitive identifiers
Authorization	Path, Operation, Policies, Result	Debug policy evaluation failures	Log pattern matches not actual paths
Secret Access	Path, Version, Operation, Result	Track secret usage patterns	Never log secret values
Dynamic Secret Generation	BackendPath, RoleName, TTL, LeaseID	Debug credential lifecycle issues	Log metadata not credentials
System Events	EventType, NodeID, Duration, Result	Monitor system health	Include performance metrics

The key insight for audit-based debugging is that security events form patterns that reveal the root cause without exposing sensitive data. For example, a series of authorization failures for the same path pattern might indicate a policy misconfiguration, while authentication failures followed by successful access suggest credential compromise.

**Correlation Analysis** across audit events helps identify complex issues that span multiple system components. The RequestID field allows tracing a single request through authentication, authorization, encryption, and storage operations. Time-based correlation can identify patterns like credential stuffing attacks or systematic enumeration attempts.

## Cryptographic Verification and Validation

Secret management systems require specialized diagnostic techniques for verifying cryptographic operations without exposing key material. These techniques focus on validating algorithm properties and detecting implementation errors.

**Test Vector Validation** uses known plaintext/ciphertext pairs to verify that encryption implementations produce expected results. The CryptoTestVector structure captures test cases from cryptographic standards that can be used to validate envelope encryption operations.

### Test Vector Validation Process:

1. Load standardized test vectors for AES-256-GCM operations
2. Execute EncryptSecret with known plaintext and test vector key
3. Compare produced ciphertext with expected test vector output
4. Verify authentication tag matches expected value
5. Test DecryptSecret with test vector ciphertext produces original plaintext
6. Validate that modified ciphertext fails authentication

**Key Hierarchy Verification** ensures that the envelope encryption key derivation process maintains security properties. This involves verifying that data encryption keys are properly derived from the master key and that

key versions are tracked correctly.

Verification Check	Method	Expected Result	Failure Indication
DEK Uniqueness	Generate multiple DEKs; compare	All different	Weak entropy or reused nonces
Master Key Derivation	Derive same key with same parameters	Identical results	Implementation inconsistency
Key Version Monotonicity	Check version sequence in storage	Strictly increasing	Version management bug
Encryption Determinism	Encrypt same data twice with different DEKs	Different ciphertexts	Proper randomization
Authentication Integrity	Modify ciphertext; attempt decryption	Decryption failure	Authentication working

## Performance and Resource Monitoring

Secret management systems exhibit specific performance characteristics that can indicate both functional and security issues. Monitoring these patterns helps identify problems before they impact service availability.

**Operation Latency Analysis** reveals performance bottlenecks and potential security issues. For example, unusually long authentication times might indicate brute-force attempts, while encryption operations taking variable time could suggest timing side-channels.

**Resource Utilization Patterns** help identify memory leaks, connection exhaustion, and other resource-related issues that are common in secret management systems due to the complexity of credential lifecycle management.

Metric	Normal Range	Warning Threshold	Critical Issues
Authentication Latency	10-50ms	>100ms	Timing attacks; database issues
Encryption Operations/sec	100-1000	<10	Key management problems
Active Lease Count	Proportional to load	Unbounded growth	Revocation failures
Memory Usage	Stable with spikes	Continuous growth	Key material leaks
Database Connections	<connection pool size	Pool exhausted	Dynamic secret backend issues

## Symptom-Cause-Fix Reference

This reference provides quick diagnosis for common symptoms observed in secret management systems. Each entry follows the pattern of observable symptom, most likely root cause, diagnostic steps, and recommended fix.

## Authentication and Access Issues

**Symptom:** "Access denied" for valid users with correct tokens

Aspect	Details
<b>Most Likely Cause</b>	Policy path matching failure or token expiration
<b>Diagnostic Steps</b>	1. Check audit logs for authorization events 2. Verify token validity with <code>GetToken</code> 3. Test path pattern matching manually 4. Confirm policy assignment to token
<b>Quick Fix</b>	Update policy rules to include correct path patterns
<b>Long-term Fix</b>	Implement policy testing framework and path validation

**Symptom:** Authentication succeeds but operations immediately fail

Aspect	Details
<b>Most Likely Cause</b>	Vault is in sealed state or cluster leadership issues
<b>Diagnostic Steps</b>	1. Check <code>IsSealed()</code> status 2. Verify cluster leader election 3. Review unseal process logs 4. Test with administrative token
<b>Quick Fix</b>	Unseal vault with sufficient shares or resolve leadership
<b>Long-term Fix</b>	Implement auto-unseal and monitoring for seal state

## Encryption and Storage Failures

**Symptom:** "Encryption failed" errors for new secret creation

Aspect	Details
<b>Most Likely Cause</b>	No active data encryption key or master key issues
<b>Diagnostic Steps</b>	1. Check active DEK count in storage 2. Verify master key accessibility 3. Test key generation manually 4. Review encryption engine logs
<b>Quick Fix</b>	Generate new DEK with <code>GenerateDataEncryptionKey</code>
<b>Long-term Fix</b>	Implement automatic key rotation and monitoring

**Symptom:** Existing secrets become unreadable after system restart

Aspect	Details
<b>Most Likely Cause</b>	Master key lost or DEK version mapping corrupted
<b>Diagnostic Steps</b>	1. Verify master key reconstruction from shares 2. Check DEK version consistency 3. Test decryption with known good secret 4. Review backup integrity
<b>Quick Fix</b>	Restore master key from backup or re-unseal properly
<b>Long-term Fix</b>	Implement master key backup verification and DEK integrity checks

## Dynamic Secret Lifecycle Problems

**Symptom:** Generated credentials don't work in target system

Aspect	Details
<b>Most Likely Cause</b>	Backend role configuration error or credential template issue
<b>Diagnostic Steps</b>	1. Test credential generation manually 2. Verify backend connectivity 3. Check role creation statements 4. Validate credential format
<b>Quick Fix</b>	Update role configuration with correct statements
<b>Long-term Fix</b>	Implement backend connection testing and credential validation

**Symptom:** Credentials work but never get revoked after expiry

Aspect	Details
<b>Most Likely Cause</b>	Revocation worker failure or backend connection issues
<b>Diagnostic Steps</b>	1. Check revocation queue size 2. Verify backend connectivity 3. Review revocation worker logs 4. Test manual revocation
<b>Quick Fix</b>	Restart revocation workers and clear queue backlog
<b>Long-term Fix</b>	Implement revocation monitoring and retry mechanisms

## High Availability and Cluster Issues

**Symptom:** Operations succeed on one node but fail on others

Aspect	Details
<b>Most Likely Cause</b>	Split-brain condition or replication lag
<b>Diagnostic Steps</b>	1. Check leadership status on all nodes 2. Verify Raft log consistency 3. Monitor replication lag 4. Test quorum requirements
<b>Quick Fix</b>	Force leader re-election and wait for convergence
<b>Long-term Fix</b>	Tune consensus timeouts and improve network reliability

#### Symptom: Cannot unseal vault despite providing correct shares

Aspect	Details
<b>Most Likely Cause</b>	Share corruption, wrong threshold, or reconstruction bug
<b>Diagnostic Steps</b>	1. Verify share checksums 2. Check threshold configuration 3. Test Shamir reconstruction manually 4. Review share generation logs
<b>Quick Fix</b>	Regenerate shares from master key backup
<b>Long-term Fix</b>	Implement share validation and backup verification

## Implementation Guidance

The debugging infrastructure for a secret management system requires careful implementation to maintain security while providing comprehensive diagnostic capabilities. This guidance provides the essential components for building effective debugging tools.

## Technology Recommendations

Component	Simple Option	Advanced Option
Logging	Standard log/slog with file rotation	Structured logging with ELK stack
Metrics	Basic Prometheus metrics	Full observability with Grafana
Tracing	HTTP request IDs	Distributed tracing with Jaeger
Health Checks	Simple HTTP endpoints	Comprehensive health monitoring
Debugging Tools	Built-in diagnostic commands	Remote debugging with security boundaries

## File Structure

```
cmd/
  vault-debug/
    main.go          ← Debugging CLI tool
internal/
  debug/
    audit.go        ← Debugging infrastructure
    crypto.go       ← Audit log analysis
    health.go       ← Cryptographic verification
    profiling.go    ← System health monitoring
  monitoring/
    metrics.go      ← Performance analysis
    health.go       ← Metrics and observability
  logging/
    audit.go        ← Prometheus metrics
    sanitizer.go    ← Health check endpoints
    sanitizer.go    ← Structured logging
    audit.go        ← Audit logger implementation
    sanitizer.go    ← Log sanitization
```

## Audit Logger Implementation

```
// NewAuditLogger creates tamper-evident audit logging with structured output GO

func NewAuditLogger(filename string) (*AuditLogger, error) {

    // TODO 1: Open log file with append mode and appropriate permissions (0600)

    // TODO 2: Initialize sequence number from last log entry + 1

    // TODO 3: Write startup marker with timestamp and process ID

    // TODO 4: Set up log rotation based on size and retention policy

    // TODO 5: Initialize mutex for thread-safe concurrent logging

    // Hint: Use O_APPEND|O_CREATE|O_WRONLY flags for file opening

}

// LogEvent writes audit event to tamper-evident log with security sanitization

func (a *AuditLogger) LogEvent(event AuditEvent) error {

    // TODO 1: Acquire mutex lock for thread-safe operation

    // TODO 2: Increment and assign sequence number to event

    // TODO 3: Sanitize event data to remove sensitive information

    // TODO 4: Marshal event to JSON with timestamp and checksum

    // TODO 5: Write to file with immediate flush to ensure durability

    // TODO 6: Release mutex lock

    // Hint: Never log actual secret values, only metadata and result status

}
```

## Cryptographic Test Vector Validation

```
// LoadKnownVectors loads standardized test vectors for encryption validation      GO
func LoadKnownVectors(algorithm string) ([]cryptoTestVector, error) {

    // TODO 1: Read test vectors from embedded resources or external file

    // TODO 2: Parse vectors based on algorithm type (AES-GCM, etc.)

    // TODO 3: Validate vector format and required fields

    // TODO 4: Return structured test cases for validation

}

// RunCryptographicValidation verifies encryption implementation against known vectors

func RunCryptographicValidation(engine *Engine, vectors []CryptoTestVector) error {

    for _, vector := range vectors {

        // TODO 1: Create test DEK with vector key material

        // TODO 2: Encrypt vector plaintext using engine

        // TODO 3: Compare result with expected ciphertext

        // TODO 4: Decrypt ciphertext and verify original plaintext

        // TODO 5: Test authentication by modifying ciphertext and expecting failure

        // Hint: Use constant-time comparison for cryptographic results

    }

}
```

## Performance Monitoring and Health Checks

```
// SystemHealth provides comprehensive health monitoring for all vault components
```

```
type SystemHealth struct {  
    checks      map[string]HealthCheck  
    lastResults map[string]ComponentStatus  
    mutex       sync.RWMutex  
    checkInterval time.Duration  
}  
  
// RegisterCheck adds component health monitoring with automatic status tracking  
  
func (h *SystemHealth) RegisterCheck(check HealthCheck) {  
    // TODO 1: Add check to registry with unique name  
    // TODO 2: Initialize last result status as unknown  
    // TODO 3: Start background goroutine for periodic checking  
    // TODO 4: Set up alerting thresholds for status changes  
}  
  
// GetOverallStatus aggregates individual component status into system-wide health  
  
func (h *SystemHealth) GetOverallStatus() ComponentStatus {  
    // TODO 1: Acquire read lock for thread-safe access  
    // TODO 2: Iterate through all component statuses  
    // TODO 3: Apply aggregation rules (any unhealthy = system unhealthy)  
    // TODO 4: Return overall status with degradation level  
    // Hint: Critical components (storage, encryption) have higher weight  
}
```

GO

## Diagnostic Command-Line Tools

```
// VaultDebugTool provides safe diagnostic capabilities for production systems GO

func main() {

    // TODO 1: Parse command-line arguments for diagnostic operation

    // TODO 2: Connect to vault instance using administrative credentials

    // TODO 3: Execute requested diagnostic with security boundaries

    // TODO 4: Output sanitized results without exposing sensitive data

    // Available commands: health, audit-analysis, key-status, lease-summary

}

// AnalyzeAuditLogs provides security-focused analysis of vault audit trails

func AnalyzeAuditLogs(logPath string, timeRange TimeRange) (*AnalysisReport, error) {

    // TODO 1: Read audit log entries within specified time range

    // TODO 2: Parse and validate log entry integrity

    // TODO 3: Identify patterns indicating security issues

    // TODO 4: Generate report with actionable recommendations

    // TODO 5: Flag potential security incidents for investigation

    // Hint: Look for authentication failures, unusual access patterns, policy violations

}
```

## Milestone Checkpoints

### After Milestone 1 (Encryption Engine):

- Run `go test ./internal/debug/crypto.go` to verify cryptographic test vectors pass
- Execute `vault-debug key-status` to confirm DEK rotation works correctly
- Check audit logs contain encryption operation events without secret values

### After Milestone 2 (Authentication):

- Test `vault-debug auth-analysis` to identify authentication patterns
- Verify constant-time token comparison with timing analysis tools
- Confirm policy evaluation audit events capture authorization decisions

## After Milestone 3 (Dynamic Secrets):

- Monitor lease metrics to ensure revocation processes work correctly
- Use `vault-debug lease-summary` to verify lease lifecycle tracking
- Test credential cleanup with backend connectivity failures

## After Milestone 4 (High Availability):

- Validate cluster health monitoring across all nodes
- Test split-brain detection and prevention mechanisms
- Verify auto-unseal monitoring and failure alerting works correctly

## Common Debugging Scenarios

### Scenario: Memory Usage Growing Unboundedly

1. Use `pprof` to capture heap profiles during operation
2. Check for unreleased cryptographic key material in memory
3. Verify lease cleanup processes are running correctly
4. Look for goroutine leaks in background workers

### Scenario: Authentication Latency Spikes

1. Enable detailed timing logs for authentication operations
2. Check database connection pool utilization
3. Monitor for brute-force attack patterns in audit logs
4. Verify constant-time operations aren't introducing delays

### Scenario: Cluster Consensus Failures

1. Check network connectivity between cluster nodes
2. Verify Raft log consistency across all nodes
3. Monitor leadership election frequency and causes
4. Test quorum requirements under various failure scenarios

The debugging infrastructure must balance comprehensive observability with strict security requirements, ensuring that diagnostic capabilities never compromise the secrets they're designed to protect.

## Future Extensions

**Milestone(s):** This section applies to the completed system after all four milestones, providing guidance for scaling and extending the secret management system beyond the core implementation.

Building a secret management system is like constructing a skyscraper foundation — the initial structure must be robust enough to support future expansion, but you can't anticipate every possible addition. After implementing the core secret management capabilities across all four milestones, organizations typically discover new requirements that push the system's boundaries. This section explores strategic extensions that maintain the security model while addressing enterprise scalability, operational complexity, and evolving infrastructure needs.

The extensions fall into three categories that address different growth pressures. **Scalability improvements** handle increased load and geographic distribution. **Additional secret backends** support new infrastructure components and security requirements. **Enterprise features** address organizational complexity with multi-tenancy, compliance reporting, and sophisticated authentication methods. Each extension builds on the solid foundation established in the core milestones while introducing new architectural challenges.

## Scalability Improvements

Think of scaling a secret management system like expanding a secure bank vault operation from a single location to a global network. The core security principles remain unchanged, but you need new mechanisms for coordination, caching, and performance optimization. Unlike typical web applications where eventual consistency is acceptable, secret management requires strong consistency for security policies while optimizing for read-heavy workloads.

### Horizontal Scaling Architecture

The current system uses a single-leader Raft cluster that handles both reads and writes on the leader node. This creates a performance bottleneck as organizations grow to thousands of applications requesting secrets. **Read replicas** provide the first scaling improvement by allowing read operations against follower nodes while maintaining consistency guarantees.

#### Decision: Read-Only Replica Architecture

- **Context:** The Raft leader becomes overwhelmed by read requests while writes remain low volume
- **Options Considered:** Read replicas, read-through caching, client-side caching
- **Decision:** Implement read replicas with linearizable read guarantees
- **Rationale:** Maintains strong consistency while distributing read load across multiple nodes
- **Consequences:** Enables horizontal read scaling but requires careful lease token validation

Read replicas must handle several consistency challenges that don't exist in typical databases. **Token validation** requires checking that tokens haven't been revoked, which means replicas need up-to-date token state. **Policy evaluation** must use current policies, not stale versions that might grant excessive access. The solution involves a **token validation cache** with short TTLs and a **policy version vector** that tracks policy updates across replicas.

Component	Purpose	Consistency Model	Cache TTL
Token Validation Cache	Verify token existence and expiration	Eventually consistent with invalidation	30 seconds
Policy Version Vector	Track policy updates across replicas	Strongly consistent	N/A
Secret Content Cache	Cache encrypted secret data	Read-your-writes consistent	5 minutes
Lease Index Cache	Track active leases for validation	Eventually consistent	1 minute

## Geographic Distribution

Organizations with global infrastructure need secret management systems that provide low latency across regions while maintaining security guarantees. **Regional clusters** create independent Raft clusters in each geographic region, connected through **cross-region replication** for disaster recovery and policy synchronization.

The challenge is determining what data requires global consistency versus regional autonomy. **Security policies** must be globally consistent to prevent privilege escalation through region-shopping. **Audit logs** require global aggregation for compliance reporting. However, **secret content** can be region-specific, and **dynamic secrets** can be generated locally to minimize latency.

Data Type	Replication Strategy	Consistency Level	Cross-Region Sync
Security Policies	Synchronous multi-region	Strong consistency	Real-time
Master Keys	Manual key ceremony	No replication	Offline process
Secret Content	Regional with backup	Regional consistency	Encrypted backup
Audit Logs	Asynchronous aggregation	Eventually consistent	Batched
Dynamic Leases	Regional generation	Regional consistency	Metadata only

**Cross-region authentication** requires special handling because tokens issued in one region must be validated globally. The solution involves **token attestation** where each region cryptographically signs token metadata, allowing other regions to validate tokens without direct communication.

## Performance Optimization

As secret request volume grows, several optimization techniques become necessary. **Connection pooling** reduces the overhead of TLS handshakes for client connections. **Bulk operations** allow applications to request multiple secrets in a single API call. **Compression** reduces network overhead for large policy documents and audit logs.

The most impactful optimization is **intelligent caching** at multiple layers. **Client-side caching** with short TTLs reduces server load while maintaining security. **Server-side caching** optimizes expensive operations like policy evaluation and encryption key retrieval. **Negative caching** prevents repeated requests for non-existent secrets.

**Critical Insight:** Unlike web application caching, secret management caching must be **security-first**. A cache miss is preferable to serving stale data that might grant unauthorized access. All caches must include cryptographic checksums and expiration enforcement.

Cache Layer	Cache Duration	Security Control	Invalidation Method
Client Secret Cache	5 minutes	Client-enforced TTL	TTL expiration only
Client Policy Cache	1 minute	Checksum validation	Server push invalidation
Server Query Cache	30 seconds	Encrypted cache keys	Write-through invalidation
Encryption Key Cache	1 hour	Hardware security module	Key rotation invalidation

**⚠ Pitfall: Cache Coherence Attacks** Attackers may attempt to exploit cache inconsistencies to access secrets using revoked tokens. Always validate tokens against authoritative sources for write operations, even if caches indicate the token is valid. Implement cache invalidation patterns that err on the side of denying access.

## Additional Secret Backends

The dynamic secret engine provides extensibility for new credential types beyond the basic database backend implemented in Milestone 3. Think of secret backends like specialized vending machines — each one understands how to generate, manage, and revoke a specific type of credential while following the same lease-based lifecycle.

### Cloud Provider Integration

Modern applications increasingly rely on cloud services that require API credentials with fine-grained permissions. **Cloud provider backends** generate time-limited IAM credentials that applications can use to access cloud resources without storing long-lived access keys.

**AWS IAM backend** integration requires several components. The secret management system needs an **IAM service account** with permissions to create and delete temporary credentials. **Role assumption** allows generating credentials for specific AWS roles based on the requesting application's identity. **Policy templates** define the permissions granted to generated credentials.

AWS Backend Component	Purpose	Configuration	Security Model
Service Account	Bootstrap credentials for AWS API	Static long-lived key	Rotated monthly
Role Templates	Define assumable roles per application	JSON policy documents	Version controlled
Session Manager	Track and revoke active sessions	In-memory session cache	Automatic cleanup
Permission Mapper	Map vault policies to AWS permissions	Path-based mapping rules	Least privilege

The **Azure Active Directory backend** follows similar patterns but uses **service principal** credentials and **OAuth 2.0 flows**. **Google Cloud IAM backend** uses **service account key generation** with **workload identity federation** for enhanced security.

**⚠ Pitfall: Cloud Credential Sprawl** Without proper lifecycle management, cloud backends can create thousands of orphaned credentials that never get revoked. Implement aggressive cleanup policies and monitor cloud provider APIs for credential usage patterns.

## PKI and Certificate Management

Applications increasingly require **X.509 certificates** for mutual TLS authentication, code signing, and service mesh communication. A **PKI backend** transforms the secret management system into a **certificate authority** that can issue, renew, and revoke certificates based on requesting application identity.

The PKI backend requires careful **root certificate management**. The **root CA private key** must be stored separately from the secret management system, potentially in **hardware security modules**. **Intermediate CA certificates** can be stored within the system and used for day-to-day certificate issuance.

Certificate Type	Issuing Authority	Validity Period	Use Case
Service Certificates	Intermediate CA	24 hours	Service-to-service mTLS
Client Certificates	Intermediate CA	7 days	Application authentication
Code Signing	Dedicated signing CA	1 hour	Container image signing
Web Certificates	Public CA integration	90 days	External web services

**Certificate revocation** requires maintaining **Certificate Revocation Lists (CRLs)** or implementing **Online Certificate Status Protocol (OCSP)** responders. The revocation system must coordinate with the lease management system to ensure certificates are revoked when their associated leases expire.

## SSH Certificate Authority

**SSH certificate authentication** provides an alternative to managing SSH keys across large server fleets. The **SSH CA backend** generates **user certificates** and **host certificates** that are trusted by SSH servers without requiring key distribution.

**User certificates** contain the requesting user's identity and authorized commands. **Host certificates** prove server identity and prevent man-in-the-middle attacks. Both certificate types include **principals** that define what users or hosts the certificate represents.

SSH Certificate Component	Purpose	Source	Validation
Certificate Public Key	Identity being certified	Generated per request	Cryptographic signature
Principals List	Authorized identities	Policy-driven mapping	Path-based rules
Critical Options	Restrictions on usage	Backend configuration	Force-commands, source IP
Extensions	Additional permissions	Default template	Agent forwarding, X11

## Enterprise Features

Enterprise environments introduce organizational complexity that requires additional security and operational capabilities. Think of enterprise features like upgrading from a personal safe to a bank vault system — you need compartmentalization, audit trails, and administrative controls that weren't necessary for simpler use cases.

### Multi-Tenancy Implementation

**Multi-tenancy** allows multiple organizations or business units to share the same secret management infrastructure while maintaining complete isolation. Each **tenant** operates as if they have a dedicated system, with separate encryption keys, policies, and audit logs.

The **tenant isolation model** uses **path-based segregation** where each tenant's secrets exist under a unique path prefix. **Encryption isolation** ensures tenants use separate data encryption keys, preventing cross-tenant data access even if path-based isolation fails. **Policy isolation** prevents tenants from accessing each other's policies or creating policies that affect other tenants.

Isolation Layer	Implementation	Security Guarantee	Failure Impact
Path Isolation	URL prefix enforcement	Prevents cross-tenant path access	Single tenant affected
Encryption Isolation	Tenant-specific DEKs	Cryptographic separation	No cross-tenant data leakage
Policy Isolation	Tenant namespace scoping	Administrative separation	No privilege escalation
Audit Isolation	Separate log streams	Compliance separation	No information disclosure

**Tenant management** requires administrative APIs for creating tenants, configuring tenant-specific policies, and managing tenant lifecycles. **Tenant authentication** can integrate with existing **identity providers** using **SAML** or **OpenID Connect** federation.

#### Decision: Tenant Isolation Model

- **Context:** Need to support multiple organizations with different security requirements
- **Options Considered:** Path-based isolation, separate clusters per tenant, virtualized instances
- **Decision:** Combine path-based isolation with encryption key separation
- **Rationale:** Provides strong security guarantees while maintaining operational efficiency
- **Consequences:** Requires careful policy engine design to prevent tenant boundary violations

#### Compliance and Reporting

Enterprise environments must demonstrate compliance with regulations like **SOX**, **PCI DSS**, **HIPAA**, and **GDPR**. **Compliance reporting** transforms raw audit logs into structured reports that demonstrate security controls and data handling practices.

**Audit log enrichment** adds contextual information to basic access logs. **User identity resolution** maps authentication tokens back to human users for accountability. **Data classification** tags secrets with sensitivity levels and tracks access patterns. **Retention policies** ensure audit logs are preserved for required periods while respecting data minimization requirements.

Compliance Requirement	Implementation	Data Collected	Report Format
Access Tracking	Enhanced audit logs	User, resource, time, result	JSON with schema validation
Data Retention	Automated log archival	All secret operations	Tamper-evident storage
Change Management	Policy versioning	All configuration changes	Diff-based reports
Incident Response	Security event correlation	Failed access patterns	Real-time alerting

**Compliance dashboards** provide real-time visibility into security posture. **Anomaly detection** identifies unusual access patterns that might indicate compromise. **Automated reporting** generates periodic compliance summaries without manual intervention.

## Advanced Authentication Methods

Enterprise environments often require **federation** with existing identity systems and **risk-based authentication** that adapts security requirements based on request context. **SAML integration** allows using enterprise identity providers for authentication. **OpenID Connect** provides more modern federation capabilities with **JSON Web Token** based identity assertion.

**Risk-based authentication** evaluates request context to determine required authentication strength. **Device fingerprinting** tracks client characteristics to detect unusual access patterns. **Geolocation analysis** flags access attempts from unexpected locations. **Behavioral analysis** learns normal access patterns and detects deviations.

Authentication Factor	Risk Level	Additional Requirements	Use Case
API Token	Low	None	Automated applications
mTLS Certificate	Medium	Certificate validation	Service authentication
SAML Assertion	Medium	Identity provider verification	Human users
MFA + Biometrics	High	Multi-factor verification	Administrative access

**Adaptive policies** adjust security requirements based on calculated risk scores. **Step-up authentication** prompts for additional verification when accessing high-value secrets. **Session management** tracks authentication sessions and enforces re-authentication for sensitive operations.

**⚠ Pitfall: Authentication Bypass** Complex authentication flows create opportunities for bypass attacks. Always validate that required authentication factors were actually verified, not just claimed. Implement defense in depth with multiple validation layers.

## Implementation Guidance

The scalability and enterprise features described in this section represent significant extensions to the core secret management system. Each extension should be approached incrementally, with careful attention to maintaining the security model established in the core milestones.

## Technology Recommendations

Component	Simple Option	Advanced Option
Read Replicas	HTTP proxy with Raft follower reads	Custom replication protocol with consistency guarantees
Cross-Region Sync	Database replication with encryption	Custom consensus protocol with conflict resolution
PKI Backend	Go crypto/x509 with file-based CA	Hardware Security Module integration
Multi-Tenancy	Path prefix enforcement	Container-based isolation with separate processes
Compliance Reporting	Structured JSON logs with external analysis	Real-time stream processing with compliance dashboard

## File Structure Extensions

The extension features require new modules that integrate with the existing codebase:

```
internal/
  scaling/
    replicas/           ← read replica management
      replica_manager.go
      consistency_checker.go
    caching/            ← multi-layer caching
      client_cache.go
      server_cache.go
  geographic/          ← cross-region support
    region_manager.go
    cross_region_sync.go

backends/
  cloud/              ← cloud provider backends
    aws_iam.go
    azure_ad.go
    gcp_iam.go
  pki/                ← certificate authority
    ca_manager.go
    cert_issuer.go
  ssh/                ← SSH certificate authority
    ssh_ca.go
    user_certs.go

enterprise/
  tenancy/             ← multi-tenant support
    tenant_manager.go
    isolation_enforcer.go
  compliance/          ← compliance reporting
    audit_enricher.go
    report_generator.go
  federation/          ← advanced authentication
    saml_handler.go
    risk_analyzer.go
```

## Scaling Infrastructure Starter Code

```
// ReplicaManager handles read-only replicas for horizontal scaling          GO

type ReplicaManager struct {

    replicas     map[string]*ReplicaNode

    healthCheck *ReplicaHealthChecker

    loadBalancer *ReadLoadBalancer

    consistency *ConsistencyChecker

}

// ReplicaNode represents a read-only follower

type ReplicaNode struct {

    NodeID      string

    Endpoint    string

    LastContact time.Time

    IsHealthy   bool

    ReadLoad    int64

}

// RouteReadRequest distributes read requests across healthy replicas

func (rm *ReplicaManager) RouteReadRequest(ctx context.Context, req *ReadRequest) (*ReadResponse, error) {

    // TODO 1: Check if request requires strong consistency (recent writes)

    // TODO 2: Select healthy replica with lowest load

    // TODO 3: Validate replica has required policy/token state

    // TODO 4: Execute read request with consistency checks

    // TODO 5: Update replica health and load metrics

}

// ConsistencyChecker ensures read replicas have current security state
```

```
type ConsistencyChecker struct {

    tokenVersions map[string]int64 // Track token state versions

    policyVersions map[string]int64 // Track policy state versions

    maxStaleness time.Duration // Maximum acceptable staleness

}

// ValidateReplicaConsistency checks if replica can safely serve read

func (cc *ConsistencyChecker) ValidateReplicaConsistency(ctx context.Context, replica
*ReplicaNode, req *ReadRequest) error {

    // TODO 1: Check token version on replica vs leader

    // TODO 2: Check policy version for requested path

    // TODO 3: Verify staleness within acceptable bounds

    // TODO 4: Return error if replica too stale for security

}
```

## Cloud Backend Skeleton

```
// AWSIAMBackend generates temporary AWS credentials          GO

type AWSIAMBackend struct {

    client      *iam.IAM

    sts         *sts.STS

    rolePrefix string

    accountID  string

    templates   map[string]*IAMRoleTemplate
}

// IAMRoleTemplate defines AWS role configuration for secret generation

type IAMRoleTemplate struct {

    RoleName      string

    PolicyDocument string

    SessionDuration time.Duration

    ExternalID     string

    TrustPolicy    string

}

// GenerateCredentials creates temporary AWS credentials for application

func (aws *AWSIAMBackend) GenerateCredentials(ctx context.Context, roleName string, ttl time.Duration) (map[string]interface{}, map[string]interface{}, error) {

    // TODO 1: Look up role template for requested role

    // TODO 2: Assume role with STS to get temporary credentials

    // TODO 3: Return credentials in standard format

    // TODO 4: Return revocation data (session info for cleanup)

}

// RevokeCredentials removes AWS session and credentials
```

```
func (aws *AWSIAMBBackend) RevokeCredentials(ctx context.Context, revocationData  
map[string]interface{}) error {  
  
    // TODO 1: Extract session info from revocation data  
  
    // TODO 2: Call STS to invalidate session  
  
    // TODO 3: Handle case where session already expired  
  
}
```

## Multi-Tenancy Core Logic

```
// TenantManager enforces isolation between organizational tenants
```

```
type TenantManager struct {

    tenants      map[string]*TenantConfig

    pathEnforcer *PathIsolationEnforcer

    keyManager   *TenantKeyManager

    auditSplitter *TenantAuditSplitter

}

// TenantConfig defines tenant isolation parameters
```

```
type TenantConfig struct {

    TenantID      string

    PathPrefix    string

    EncryptionKeyID string

    PolicyNamespace string

    AuditStreamID  string

    CreatedAt     time.Time

    AdminContacts  []string

}

// EnforceTenantIsolation validates request stays within tenant boundaries
```

```
func (tm *TenantManager) EnforceTenantIsolation(ctx context.Context, tenantID string,
requestPath string, operation string) error {

    // TODO 1: Look up tenant configuration

    // TODO 2: Verify request path starts with tenant prefix

    // TODO 3: Check operation is allowed for tenant

    // TODO 4: Log tenant boundary enforcement in audit

}
```

GO

```

// GetTenantEncryptionKey returns tenant-specific DEK

func (tm *TenantManager) GetTenantEncryptionKey(ctx context.Context, tenantID string,
keyVersion int) ([]byte, error) {

    // TODO 1: Validate tenant ID and key version

    // TODO 2: Retrieve tenant-specific master key

    // TODO 3: Decrypt and return tenant DEK

    // TODO 4: Ensure cross-tenant key isolation

}

```

## Milestone Checkpoints

After implementing scaling extensions:

- **Performance Test:** Verify read replicas handle 10x read load increase with < 100ms latency
- **Consistency Test:** Confirm security policy changes propagate to replicas within 1 second
- **Geographic Test:** Validate cross-region secret replication with encrypted transport

After implementing cloud backends:

- **AWS Integration:** Generate and revoke AWS IAM credentials successfully
- **Certificate Generation:** Issue and validate X.509 certificates for mTLS
- **SSH Access:** Authenticate to test servers using generated SSH certificates

After implementing enterprise features:

- **Tenant Isolation:** Verify tenant A cannot access tenant B's secrets under any conditions
- **Compliance Export:** Generate compliance report covering all audit requirements
- **Federation Test:** Authenticate users through SAML and validate policy enforcement

## Common Implementation Challenges

**⚠ Scaling Challenge: Cache Invalidation Races** When policies change, cache invalidation messages may arrive out of order, causing replicas to temporarily allow unauthorized access. Implement version vectors that ensure caches never regress to older policy versions.

**⚠ Backend Challenge: Credential Cleanup Failures** Cloud providers may be temporarily unavailable during credential revocation, leaving orphaned credentials. Implement exponential backoff retry with dead letter queues for failed revocations that require manual cleanup.

**⚠ Enterprise Challenge: Tenant Boundary Violations** Complex policy rules may inadvertently allow cross-tenant access. Implement mandatory tenant ID validation at the storage layer as a final enforcement mechanism, regardless of policy evaluation results.

The extensions described in this section transform the basic secret management system into an enterprise-grade platform capable of supporting large-scale, globally distributed infrastructure. Each extension maintains the core security principles while addressing real-world operational requirements that emerge as organizations scale their secret management practices.

## Glossary

**Milestone(s):** This section provides essential terminology and definitions that apply across all four milestones, serving as a reference for understanding the complex concepts, architectural patterns, and security mechanisms implemented throughout the secret management system.

The secret management domain combines cryptography, distributed systems, authentication protocols, and security engineering concepts that may be unfamiliar to developers new to this space. This glossary provides precise definitions for technical terms, explains domain-specific concepts with clear examples, and clarifies the specific meaning of terms within the context of our secret management system.

Think of this glossary as a technical dictionary that transforms abstract security concepts into concrete understanding. Just as a foreign language dictionary helps you understand individual words before reading literature, this glossary helps you understand individual concepts before diving into system implementation. Each definition includes not just what something is, but why it matters and how it fits into the larger security architecture.

## Core Security Concepts

The foundation of any secret management system rests on fundamental security principles that guide every design decision. These concepts form the mental framework for understanding why the system is built the way it is.

Term	Definition	Example/Context
<b>Zero-trust system</b>	Security model requiring authentication and authorization for every request, assuming no inherent trust based on network location or previous access	Our secret management system validates every API request with a token, even from internal services
<b>Defense in depth</b>	Multiple security layers protecting against various attack vectors, ensuring that compromise of one layer doesn't compromise the entire system	Combines envelope encryption, access policies, audit logging, and network TLS
<b>Assume breach</b>	Security principle of designing for compromised components, building systems that remain secure even when individual parts are attacked	Dynamic secrets expire automatically even if the credential database is compromised
<b>Blast radius containment</b>	Limiting impact of compromise to specific subset of credentials or operations	Each tenant's secrets encrypted with different keys; compromise affects only that tenant
<b>Secret sprawl</b>	Scattered secrets across infrastructure creating security vulnerabilities through inconsistent storage, access control, and rotation practices	Database passwords hardcoded in config files across hundreds of microservices
<b>System charter</b>	Contract defining system scope and limitations, establishing what the system will and will not do	Our system manages application secrets but explicitly excludes user passwords and PKI certificates

## Cryptographic Concepts

Secret management systems rely heavily on cryptographic primitives and patterns. Understanding these concepts is essential for implementing secure storage and key management.

Term	Definition	Example/Context
<b>Envelope encryption</b>	Multi-layer encryption with master key protecting data keys, enabling key rotation without re-encrypting all data	Master key encrypts DEKs, DEKs encrypt individual secrets; rotate DEK without touching master key
<b>Nested safe system</b>	Mental model for envelope encryption with multiple security layers, like safes within safes	Bank vault (master key) contains safety deposit boxes (DEKs) containing valuables (secrets)
<b>Data encryption keys</b>	DEKs that actually encrypt secrets, protected by master key, enabling efficient key rotation and access control	Each secret encrypted with unique DEK; compromise of one DEK affects only that secret
<b>Key rotation isolation</b>	Ability to rotate keys independently without affecting other keys or requiring system downtime	Rotate DEK for tenant A without affecting tenant B's secrets or requiring system restart
<b>Version immutability</b>	Principle that secret versions cannot be changed once created, ensuring audit trail and preventing tampering	Secret version 3 remains unchanged even when version 4 is created; enables rollback and compliance
<b>Lazy re-encryption</b>	Background process to migrate old versions to new keys without blocking operations	Old secret versions gradually re-encrypted with new DEK during normal access patterns
<b>Constant-time comparison</b>	Cryptographic comparison that takes same time regardless of input to prevent timing attacks	Token validation uses constant-time comparison to prevent attackers from guessing tokens

## Authentication and Authorization

Access control in secret management systems requires sophisticated identity and permission models that can scale across different authentication methods and organizational structures.

Term	Definition	Example/Context
<b>Bearer token authentication</b>	Authentication method using cryptographically signed tokens carried in HTTP headers	<code>X-Vault-Token: hvs.CAESIF...</code> header authenticates API requests
<b>Corporate badge system</b>	Mental model for token-based access control, where tokens work like employee badges granting building access	Token contains policies like badge contains access levels; both can be revoked centrally
<b>Path-based access control</b>	ACL system using hierarchical paths with wildcard patterns to grant fine-grained permissions	Policy allows <code>secret/app-*/database/*</code> but denies <code>secret/app-*/api-keys/*</code>
<b>Wildcard pattern matching</b>	Flexible path matching using * for single segments and ** for recursive matching	<code>secret/*/database</code> matches <code>secret/app1/database</code> but not <code>secret/app1/cache/database</code>
<b>Default deny</b>	Security model where access is denied unless explicitly allowed by policy	User can only access paths explicitly granted in their token's policies
<b>Policy inheritance</b>	Mechanism where identities inherit policies from groups and roles	User inherits developer group policies plus individual role-specific policies
<b>Tamper-evident logging</b>	Audit logging designed to detect modification or deletion of log entries	Each audit entry includes hash of previous entry; broken chain indicates tampering
<b>Mutual TLS authentication</b>	Bidirectional certificate-based authentication between client and server	Both client and server present certificates; validates identity cryptographically

## Dynamic Secret Management

Dynamic secrets represent a paradigm shift from static credentials to just-in-time credential generation with automatic lifecycle management.

Term	Definition	Example/Context
<b>Credential vending machine</b>	System that generates fresh credentials on demand rather than storing pre-created credentials	Request database access, receive unique username/password valid for 1 hour
<b>Dynamic secrets</b>	Short-lived credentials generated on demand with automatic expiration and cleanup	Database credentials created when requested, automatically deleted after TTL expires
<b>Just-in-time generation</b>	Creating credentials only when requested not stored in advance	No pre-created database users; new user created for each credential request
<b>Lease-based tracking</b>	Time-limited credential lifecycle management with explicit expiration and renewal capabilities	Each credential has lease ID, expiration time, and renewal count
<b>Automatic cleanup</b>	System responsibility for revoking credentials when leases expire without requiring manual intervention	Expired database users automatically dropped from database by revocation engine
<b>Backend plugin</b>	Service-specific implementation of credential generation and management for different systems	PostgreSQL backend knows how to CREATE USER, AWS backend knows IAM API calls
<b>Revocation engine</b>	Cleanup system that removes expired credentials from target systems using service-specific protocols	Executes DROP USER for database credentials, DeleteAccessKey for AWS credentials
<b>Lease reaper</b>	Background process that finds and processes expired leases on configurable intervals	Runs every 30 seconds, finds leases expired in last batch, queues for revocation
<b>Exponential backoff</b>	Retry strategy with increasing delays between attempts to handle temporary failures gracefully	First retry after 1s, then 2s, 4s, 8s, up to maximum delay
<b>Dead letter queue</b>	Storage for revocation requests that failed maximum retry attempts requiring manual intervention	Failed AWS credential deletions stored for operator review and manual cleanup
<b>Priority revocation</b>	Expedited credential cleanup for security incidents bypassing normal revocation queues	Compromised service account credentials revoked immediately, not queued

## Distributed Systems and High Availability

Secret management systems must remain available and consistent across multiple nodes while handling network partitions and node failures gracefully.

Term	Definition	Example/Context
<b>Shamir's secret sharing</b>	Cryptographic threshold scheme splitting secrets into shares requiring minimum number to reconstruct	Master key split into 5 shares, requiring any 3 to reconstruct and unseal system
<b>Threshold scheme</b>	Requires minimum number of shares to reconstruct secret, providing security against partial compromise	3-of-5 threshold means compromise of 2 shares doesn't compromise master key
<b>Polynomial interpolation</b>	Mathematical technique for reconstructing polynomial from points used in Shamir's secret sharing	Secret encoded as y-intercept of polynomial; shares are points used for reconstruction
<b>Sealed state</b>	Secure mode where master key absent and operations blocked until sufficient shares provided	Server starts sealed; all secret operations return 503 until unsealing completes
<b>Unsealing</b>	Process of reconstructing master key and enabling operations by collecting threshold shares	Operators provide 3 of 5 key shares; system reconstructs master key and enables operations
<b>Leader election</b>	Distributed consensus process selecting single coordinator node to prevent split-brain scenarios	Raft algorithm ensures exactly one node accepts writes; others forward to leader
<b>Raft consensus</b>	Algorithm ensuring strong consistency across cluster nodes using leader-follower replication	Leader accepts writes, replicates to majority before committing; handles leader failure
<b>Auto-unseal</b>	Automated unsealing using external key management service instead of manual share collection	AWS KMS encrypts master key; system auto-unseals using KMS on startup
<b>Key encryption key</b>	External key used to encrypt/decrypt the master key in auto-unseal configurations	AWS KMS key encrypts vault master key; stored encrypted master key in configuration
<b>Split-brain scenarios</b>	Dangerous state where multiple nodes accept conflicting writes due to network partition	Two data centers lose connectivity; both elect leaders and accept conflicting secret updates

## Request Processing and Error Handling

Processing secret management requests requires sophisticated pipeline architecture with comprehensive error handling and graceful degradation capabilities.

Term	Definition	Example/Context
<b>Request processing pipeline</b>	Multi-stage request validation, authentication, and authorization flow with audit logging	Parse → Authenticate → Authorize → Execute → Audit → Response
<b>Circuit breaker</b>	Pattern preventing cascading failures by stopping requests to failed services	Stop sending lease revocations to failed database after 5 consecutive failures
<b>Audit event generation</b>	Comprehensive logging of security-relevant operations for compliance and forensics	Every secret access logged with identity, timestamp, result, and request details
<b>Graceful degradation</b>	Reducing functionality while maintaining core security guarantees during failures	Database backend failure disables dynamic secrets but static secrets remain available
<b>Read-only mode</b>	Degraded operational state allowing secret retrieval but blocking write operations	Storage failure triggers read-only mode; secrets accessible but new secrets rejected
<b>Disaster recovery</b>	Procedures for restoring service after catastrophic failures using backups and redundancy	Restore from encrypted backup when primary cluster completely destroyed
<b>Emergency security procedures</b>	Final defense mechanisms when normal security controls may be insufficient	Emergency seal command immediately blocks all operations and clears memory
<b>Fail-safe approach</b>	Prioritizing data integrity over availability during ambiguous failure conditions	Unknown error during secret write results in failure rather than potential corruption
<b>Backup separation</b>	Storing encrypted data and decryption keys using different systems and procedures	Encrypted secrets backed up to S3; key shares stored in different security deposit boxes

## Testing and Security Validation

Secret management systems require specialized testing approaches to validate both functional correctness and security properties under various conditions.

Term	Definition	Example/Context
<b>Timing side-channel</b>	Information leakage through execution time differences revealing secret information	Token validation time varies by input length, allowing attackers to guess valid tokens
<b>Property-based testing</b>	Testing approach using randomized inputs to verify algorithmic properties hold universally	Test that $\text{encrypt}(\text{decrypt}(x)) == x$ for thousands of random inputs
<b>Penetration testing</b>	Security testing simulating real-world attack scenarios against deployed systems	Attempt privilege escalation, token theft, timing attacks against running system
<b>Chaos testing</b>	Testing system behavior under random failure conditions to validate resilience	Randomly kill nodes, corrupt network packets, fill disk space during operations
<b>Load testing</b>	Performance testing under realistic traffic patterns to validate scalability	1000 concurrent clients requesting dynamic secrets every second for 10 minutes
<b>Integration testing</b>	Testing complete workflows across all system components end-to-end	Create secret, generate policy, authenticate user, retrieve secret, verify audit log

## Scalability and Enterprise Features

Production secret management systems must handle enterprise-scale requirements including multi-tenancy, geographic distribution, and regulatory compliance.

Term	Definition	Example/Context
<b>Read replicas</b>	Follower nodes handling read-only requests for scaling without compromising consistency	Secret reads distributed across 3 replicas; writes processed by single leader
<b>Geographic distribution</b>	Regional clusters with cross-region replication for disaster recovery and latency	Primary cluster in us-east-1; replica cluster in eu-west-1 for European customers
<b>Multi-tenancy</b>	Organizational isolation within shared infrastructure using cryptographic and path separation	Each customer's secrets encrypted with tenant-specific keys and isolated paths
<b>Compliance reporting</b>	Structured audit reports for regulatory requirements like SOX, GDPR, HIPAA	Monthly access reports showing who accessed which secrets when for audit review
<b>Federation</b>	Integration with external identity providers for authentication and authorization	Authenticate users against corporate Active Directory; map AD groups to vault policies
<b>Risk-based authentication</b>	Adaptive security based on request context like location, time, device, behavior	Unusual access patterns trigger additional authentication challenges
<b>Tenant isolation</b>	Cryptographic and administrative separation between organizations sharing infrastructure	Customer A cannot access customer B's secrets even with system administrator privileges
<b>Path-based segregation</b>	Tenant separation using URL prefix enforcement with cryptographic boundaries	Tenant A limited to <code>/secret/tenant-a/*</code> paths with separate encryption keys

## System Operations and Maintenance

Operating a secret management system requires understanding maintenance procedures, monitoring approaches, and operational best practices.

Term	Definition	Example/Context
<b>Lease renewal</b>	Extending credential lifetime before expiration for ongoing operations	Application renews database credentials every hour to maintain continuous access
<b>Key rotation</b>	Periodic replacement of encryption keys to limit blast radius of compromise	Master key rotated annually; data encryption keys rotated monthly
<b>Seal recovery</b>	Emergency procedures for reconstructing master key when normal unsealing fails	Use backup key shares stored in different locations to recover sealed system
<b>Health monitoring</b>	Continuous validation of system components and dependencies	Monitor encryption engine, storage backend, cluster consensus, lease reaper
<b>Performance tuning</b>	Optimizing system configuration for throughput, latency, and resource utilization	Adjust lease reaper batch size, revocation worker count, token cache TTL
<b>Backup validation</b>	Regular testing of backup and restore procedures to ensure disaster recovery capability	Monthly restore test using previous week's backup to validate recovery procedures

## Implementation Guidance

Secret management systems involve complex terminology that can be overwhelming for developers new to the security domain. This implementation guidance provides practical approaches for managing this complexity during development.

## Technology Recommendations for Documentation

Component	Simple Option	Advanced Option
Code Documentation	Inline comments with examples	Comprehensive API documentation with OpenAPI/Swagger
Architecture Documentation	Markdown files with diagrams	Interactive documentation with PlantUML diagrams
Troubleshooting Guides	Text-based symptom/solution tables	Searchable knowledge base with categorized issues
Glossary Management	Static markdown glossary	Searchable glossary with cross-references and examples

## Recommended Documentation Structure

Organize terminology and documentation to support both development and operations teams:

---

```
docs/
  glossary.md           ← this comprehensive glossary
  concepts/
    envelope-encryption.md   ← detailed concept explanations
    dynamic-secrets.md      ← deep dive into encryption patterns
    access-control.md       ← credential lifecycle management
    high-availability.md   ← authentication and authorization
    troubleshooting/
      common-errors.md     ← clustering and consensus
      diagnostic-procedures.md   ← operational guidance
      performance-tuning.md   ← frequent issues and solutions
      troubleshooting/
        common-errors.md   ← step-by-step problem diagnosis
        diagnostic-procedures.md   ← optimization recommendations
        troubleshooting/
          common-errors.md   ← API documentation
          diagnostic-procedures.md   ← auth method documentation
          troubleshooting/
            common-errors.md   ← backend-specific APIs
  api/
    authentication.md
    secret-engines.md
```

## Terminology Management Code

```
// TerminologyValidator helps ensure consistent terminology usage across the codebase
// GO

type TerminologyValidator struct {

    preferredTerms map[string]string

    deprecatedTerms map[string]string

}

// NewTerminologyValidator creates a validator with standard secret management terms

func NewTerminologyValidator() *TerminologyValidator {

    return &TerminologyValidator{



        preferredTerms: map[string]string{

            "envelope encryption": "Multi-layer encryption with master key protecting data
keys",

            "dynamic secrets":      "Short-lived credentials generated on demand",

            "zero-trust":           "Security model requiring authentication for every
request",

            "secret sprawl":        "Scattered secrets creating security vulnerabilities",

        },


        deprecatedTerms: map[string]string{

            "secret management": "Use 'secret storage' or 'credential management' for
clarity",

            "encryption":         "Specify 'envelope encryption' or 'field-level
encryption'",

            "authentication":     "Specify 'bearer token' or 'mutual TLS authentication'",

        },


    }
}

// ValidateDocumentation checks documentation for consistent terminology usage

func (tv *TerminologyValidator) ValidateDocumentation(content string) []TerminologyIssue {
```

```
// TODO: Implement documentation parsing and terminology validation  
  
// TODO: Check for deprecated terms and suggest preferred alternatives  
  
// TODO: Verify technical terms are defined before first usage  
  
// TODO: Flag inconsistent usage of the same concept across documents  
  
return nil  
  
}
```

## Glossary Integration Utilities

```
// GlossaryLookup provides runtime access to terminology definitions
```

```
type GlossaryLookup struct {

    definitions map[string]Definition

    examples     map[string][]string

    crossRefs   map[string][]string

}

type Definition struct {

    Term          string

    Definition   string

    Context      string

    Examples     []string

    SeeAlso      []string

    Category     string

}

// LoadGlossary parses the markdown glossary into structured data

func LoadGlossary(glossaryPath string) (*GlossaryLookup, error) {

    // TODO: Parse markdown glossary tables into structured definitions

    // TODO: Extract cross-references and examples

    // TODO: Build category indexes for related terms

    // TODO: Validate all cross-references resolve to valid terms

    return nil, nil

}

// Define returns the definition for a technical term

func (gl *GlossaryLookup) Define(term string) (Definition, bool) {
```

GO

```
// TODO: Return definition with examples and cross-references  
  
// TODO: Handle case-insensitive lookup and common variations  
  
// TODO: Track usage statistics for documentation improvement  
  
return Definition{}, false  
  
}
```

## Milestone Checkpoint: Terminology Consistency

After implementing each milestone, validate terminology consistency:

### Milestone 1 - Encryption Terms:

- Run: `grep -r "encryption" internal/ | validate-terminology.sh`
- Expected: All references use "envelope encryption" when describing the key hierarchy
- Signs of issues: Generic "encryption" terms without specifying the pattern

### Milestone 2 - Authentication Terms:

- Run: `grep -r "auth" internal/ | validate-terminology.sh`
- Expected: Specific terms like "bearer token authentication" and "mutual TLS"
- Signs of issues: Vague "authentication" without specifying the method

### Milestone 3 - Dynamic Secret Terms:

- Run: `grep -r "credential" internal/ | validate-terminology.sh`
- Expected: "Dynamic secrets" and "lease-based tracking" terminology
- Signs of issues: "Temporary credentials" or "short-lived tokens" instead

### Milestone 4 - Consensus Terms:

- Run: `grep -r "cluster" internal/ | validate-terminology.sh`
- Expected: "Raft consensus" and "leader election" terminology
- Signs of issues: Generic "replication" without specifying consensus algorithm

## Debugging Terminology Issues

Symptom	Likely Cause	How to Diagnose	Fix
Inconsistent API documentation	Mixed terminology across endpoints	Search codebase for term variations	Establish style guide with preferred terms
Confusing error messages	Technical jargon without context	Review error messages with non-experts	Add glossary references to error docs
Unclear troubleshooting guides	Assumed knowledge of domain terms	Test guides with junior developers	Define terms inline or reference glossary
Complex architecture docs	Too many undefined acronyms	Count undefined terms per page	Maintain acronym list and spell out on first use