

High-Level Design (High Level Design) - Securaa Custom Services¶

1. Executive Summary¶

1.1 System Purpose and Overview¶

Securaa Custom Services represents a sophisticated Go-based microservice architecture designed specifically for security automation platforms. This system serves as the backbone for integration and task management capabilities, enabling organizations to create, manage, and execute custom security applications and workflows. The service is architected to handle complex multi-tenant environments while maintaining high performance, security, and scalability standards.

The primary mission of Securaa Custom Services is to provide a unified platform where security teams can:

- Design and deploy custom security applications tailored to their specific needs
- Create and manage complex task workflows with interdependencies
- Import and export complete application configurations for portability and backup
- Process security events in real-time with automated response capabilities
- Maintain complete data isolation between different organizational tenants
- Scale seamlessly from small deployments to enterprise-level installations

1.2 Architecture Philosophy and Design Principles¶

The system is built on several key architectural principles that guide its design and implementation:

Microservices Architecture: Securaa Custom Services follows a microservices architectural pattern, operating as a standalone, independently deployable service. This approach provides several benefits including independent scaling, technology flexibility, and fault isolation. The service communicates with other components through well-defined REST APIs, ensuring loose coupling and high cohesion.

Layered Architecture Pattern: The internal structure follows a strict layered architecture with clear separation of concerns:

- **Presentation Layer** (Controllers): Handles HTTP requests, input validation, and response formatting
- **Business Logic Layer** (Services): Contains core business rules, workflows, and processing logic
- **Data Access Layer** (Models): Manages data structures, database interactions, and data validation
- **Infrastructure Layer:** Provides cross-cutting concerns like logging, security, and configuration management

Multi-Tenant by Design: The architecture is inherently designed for multi-tenancy, supporting both shared database models for smaller deployments and completely isolated database instances for enterprise MSSP (Managed Security Service Provider) environments. This flexibility allows the system to adapt to different organizational requirements and compliance needs.

Security-First Approach: Security is not an afterthought but a core design principle. Every component includes built-in security measures including field-level encryption, tenant isolation, comprehensive authentication and authorization mechanisms, and audit logging capabilities.

1.3 Technology Foundation¶

Programming Language and Runtime: Built with Go 1.17, leveraging Go's excellent concurrency model, memory efficiency, and strong standard library. Go's garbage collection and built-in HTTP server capabilities make it ideal for building high-performance web services.

Web Framework: Utilizes Gorilla Mux for HTTP routing, providing powerful URL routing capabilities, middleware support, and flexible request handling. Gorilla Mux offers better performance and flexibility compared to Go's standard HTTP router while maintaining simplicity.

Database Technology: MongoDB serves as the primary data store, chosen for its:

- Document-oriented structure that naturally fits complex security data models
- Built-in sharding capabilities for horizontal scaling
- Flexible schema design supporting evolving security requirements
- Excellent performance for read-heavy workloads common in security applications
- Native support for multi-tenant architectures

Caching Strategy: Redis provides high-performance caching capabilities for:

- Application configuration caching to reduce database load
- Session management and token storage
- Frequently accessed integration listings
- Real-time data that requires sub-second access times

1.4 Core Capabilities and Features¶

Custom Application Management: The system provides comprehensive capabilities for managing custom security applications. Users can create, configure, and deploy applications with complex parameter sets, file uploads (including logos), and integration-specific configurations. The system supports both generic applications that can be reused across tenants and highly customized applications tailored for specific use cases.

Generic Task Creation and Execution: One of the most powerful features is the ability to create and manage generic tasks that can be composed into complex workflows. These tasks support:

- Complex input parameter definitions with validation rules
- Dependency management between tasks
- Real-time execution monitoring and status tracking
- Output field mapping for downstream task consumption
- Error handling and retry mechanisms

Data Export/Import Functionality: The system includes sophisticated export/import capabilities that enable:

- Complete application configuration portability between environments
- Backup and disaster recovery scenarios
- Template-based application deployment across multiple tenants
- Version control and change management for application configurations

Integration Management: Comprehensive integration management features allow for:

- Connection management with external systems
- Credential storage with encryption and secure access
- API endpoint configuration and testing
- Health monitoring and status reporting
- Rate limiting and throttling capabilities

Event Processing and Case Ingestion: Real-time event processing capabilities include:

- High-volume security event ingestion from multiple sources
- Event correlation and analysis workflows
- Automated case creation and management
- Custom event processing rules and filters
- Integration with SIEM and other security platforms

Multi-Tenant Support: Enterprise-grade multi-tenancy includes:

- Complete data isolation between tenants
- Tenant-specific database instances (MSSP mode)
- Shared infrastructure with logical separation (single-tenant mode)
- Per-tenant configuration and customization
- Independent scaling and resource allocation

License Management: Sophisticated licensing capabilities support:

- Trial license limitations and usage tracking
- Enterprise license validation and feature enabling
- Usage monitoring and compliance reporting
- Automated license renewal and notification systems

2. System Architecture¶

2.1 Overall System Architecture¶

The Securaa Custom Services architecture is designed as a modern, cloud-native microservice that operates within a larger security automation ecosystem. The system is structured in multiple layers, each with specific responsibilities and clear interfaces between them. This architectural approach ensures maintainability, scalability, and testability while providing the flexibility needed for complex security workflows.

2.1.1 Architectural Layers Explanation¶

Client Layer: The topmost layer represents various client applications that interact with the Securaa Custom Services. This includes web-based dashboards for security analysts and programmatic API clients for automated integrations. Each client type may have different authentication and authorization requirements, but all interact through the same standardized REST API interface.

Load Balancing and Gateway Layer: This layer provides the entry point for all external traffic. The Application Load Balancer distributes incoming requests across multiple service instances, ensuring high availability and optimal resource utilization. The API Gateway, when present, provides additional services such as rate limiting, API versioning, request/response transformation, and centralized authentication policies.

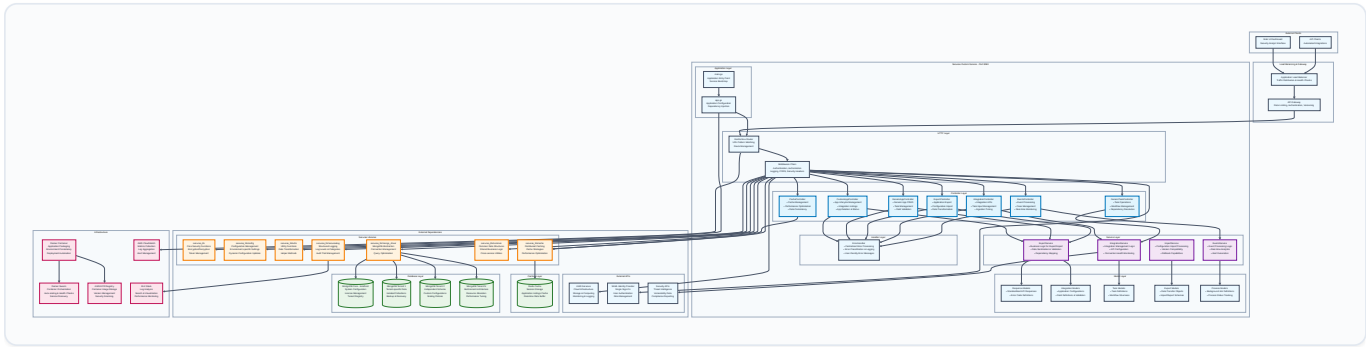
Application Core: The heart of the system consists of the Securaa Custom Service itself, which is structured in sublayers:

- **HTTP Layer:** Manages all HTTP-related concerns including routing, middleware processing, and protocol handling
- **Controller Layer:** Contains the business logic controllers that handle specific functional areas
- **Service Layer:** Implements core business logic and orchestrates complex operations
- **Model Layer:** Defines data structures and handles data validation and transformation

External Dependencies: The system integrates with several external services and libraries:

- **Securaa Libraries:** A collection of specialized libraries providing security, configuration, and utility functions
- **Database Layer:** MongoDB instances configured for multi-tenant operations
- **Caching Layer:** Redis for high-performance data caching and session management
- **External APIs:** Integration with cloud services, identity providers, and third-party security tools

2.1.2 High-Level Architecture Diagram



2.2 Component Interaction Architecture¶

The component interaction architecture demonstrates how different parts of the system communicate and collaborate to fulfill business requirements. This architecture emphasizes loose coupling between components while maintaining high cohesion within each component's responsibilities.

2.2.1 Inter-Component Communication Patterns¶

Request-Response Pattern: The primary communication pattern between the presentation layer (controllers) and business logic layer (services) follows a synchronous request-response model. Controllers receive HTTP requests, validate input parameters, and delegate business logic to appropriate service classes. Services process the business logic and return structured responses that controllers format into HTTP responses.

Service Orchestration: Complex business operations often require coordination between multiple services. For example, the application export process involves the ExportService coordinating with integration data retrieval, task dependency mapping, and data sanitization operations. This orchestration is handled through direct service-to-service calls within the same application instance.

Event-Driven Patterns: For certain operations, especially those involving cache invalidation and background processing, the system uses event-driven communication patterns. When an application is created or modified, events are triggered to update caches, notify dependent systems, and trigger background processing tasks.

Database Access Patterns: All database access is abstracted through the model layer, which provides a consistent interface for data operations. The system uses the repository pattern implementation through the Securaa MongoDB driver library, ensuring consistent error handling, connection management, and query optimization across all database operations.

2.2.2 Component Architecture Diagram¶



This component architecture demonstrates the clear separation of concerns and the flow of data and control through the system. Each component has well-defined responsibilities and interfaces, making the system maintainable and extensible.

3. Data Architecture and Database Design¶

3.1 Database Architecture Philosophy¶

The data architecture of Securaa Custom Services is designed around the principles of multi-tenancy, data isolation, and scalability. The system employs MongoDB as its primary database technology due to its document-oriented nature, which naturally accommodates the complex and varied data structures required by security applications. The architecture supports two distinct operational modes: single-tenant deployment for smaller organizations and multi-tenant MSSP (Managed Security Service Provider) deployment for enterprise environments.

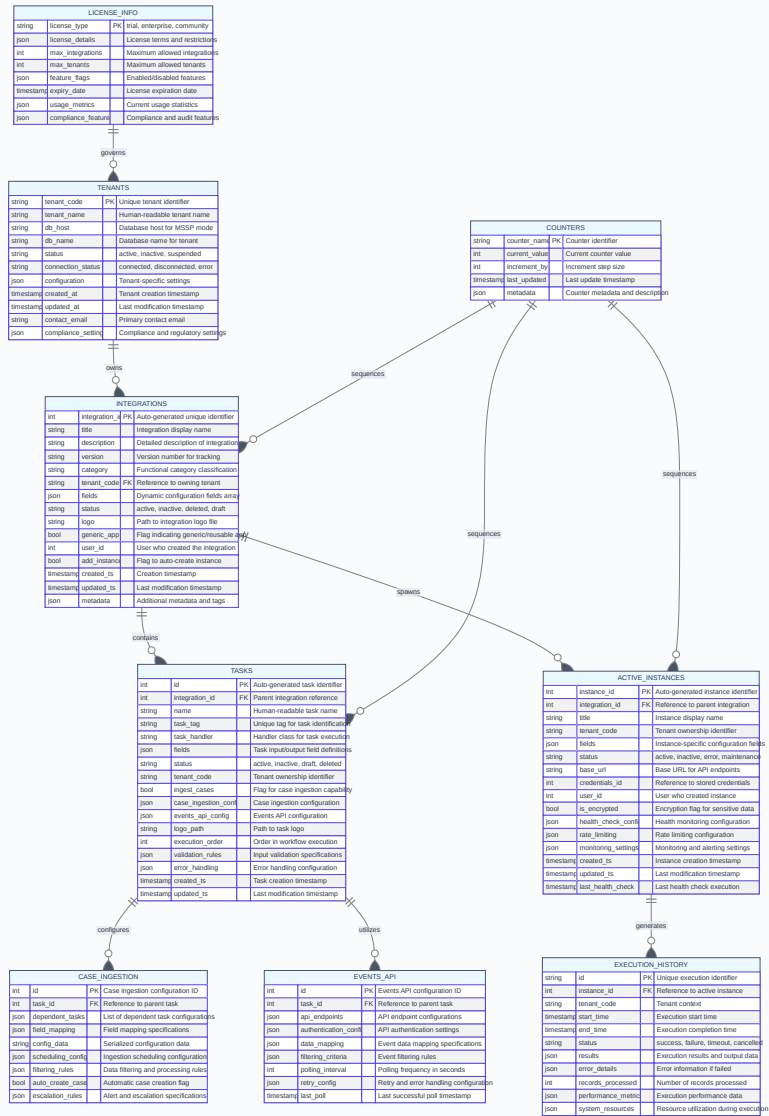
3.1.1 Multi-Tenant Data Strategy¶

Complete Data Isolation (MSSP Mode): In MSSP mode, each tenant receives a completely separate database instance. This approach provides the highest level of data isolation and security, ensuring that tenant data never coexists in the same database. Each tenant database maintains its own collections, indexes, and backup schedules. This model is ideal for organizations with strict compliance requirements or those handling highly sensitive data where data bleeding between tenants cannot be tolerated.

Logical Separation (Single-Tenant Mode): For smaller deployments or development environments, the system supports a shared database model with logical tenant separation. In this mode, all tenant data resides in the same database but is strictly separated through `tenant_code` fields in every document. Database queries automatically include tenant filters to ensure complete data isolation at the application level.

Core Database Functions: Regardless of the tenant mode, a core database (localhost) maintains system-wide information including tenant registry, license management, system configuration, and sequence counters. This central repository ensures consistency across all tenant instances while maintaining operational efficiency.

3.1.2 Database Design and Entity Relationships¶



3.1.3 Data Model Explanations

Tenants Collection: The foundation of the multi-tenant architecture, this collection maintains comprehensive information about each organizational tenant. It includes not only basic identification information but also database connection details for MSSP deployments, status tracking, and tenant-specific configuration settings. The collection supports both active tenant management and historical tracking for compliance purposes.

Integrations Collection: This collection stores the core application definitions that users create within the system. Each integration represents a complete security application with its configuration, field definitions, and metadata. The flexible JSON fields structure accommodates varying integration requirements while maintaining data consistency through validation rules.

Tasks Collection: Tasks represent the building blocks of security workflows. This collection stores task definitions, their input/output specifications, and execution configurations. The relationship between tasks and integrations allows for complex workflow composition, while the case ingestion and events API configurations enable real-time security operations.

Active Instances Collection: When integrations are deployed for actual use, they create active instances that contain runtime configuration, credentials, and operational status. This separation between integration definitions (templates) and active instances (runtime) provides flexibility for testing, staging, and production deployments.

Execution History Collection: This collection maintains a comprehensive audit trail of all task executions, including performance metrics, results, and error information. This data is crucial for debugging, performance optimization, and compliance reporting.

3.2 Multi-Tenant Database Architecture¶

The multi-tenant architecture is designed to provide complete flexibility in deployment models while maintaining consistent application behavior regardless of the chosen approach.

Generic Task Endpoints: Task management endpoints provide comprehensive CRUD operations for task definitions, validation, and execution management. These endpoints support complex task relationships, dependency management, and workflow composition.

Export/Import Endpoints: Data portability endpoints enable complete application and configuration export/import operations. These endpoints handle complex dependency resolution, version compatibility, and data transformation requirements.

Integration Management Endpoints: These endpoints manage the operational aspects of integrations, including input parameter management, execution timing, and health monitoring.

Event Processing Endpoints: Real-time event processing endpoints handle security event ingestion, case management, and real-time monitoring capabilities.

4.1.2 Comprehensive API Endpoints Overview

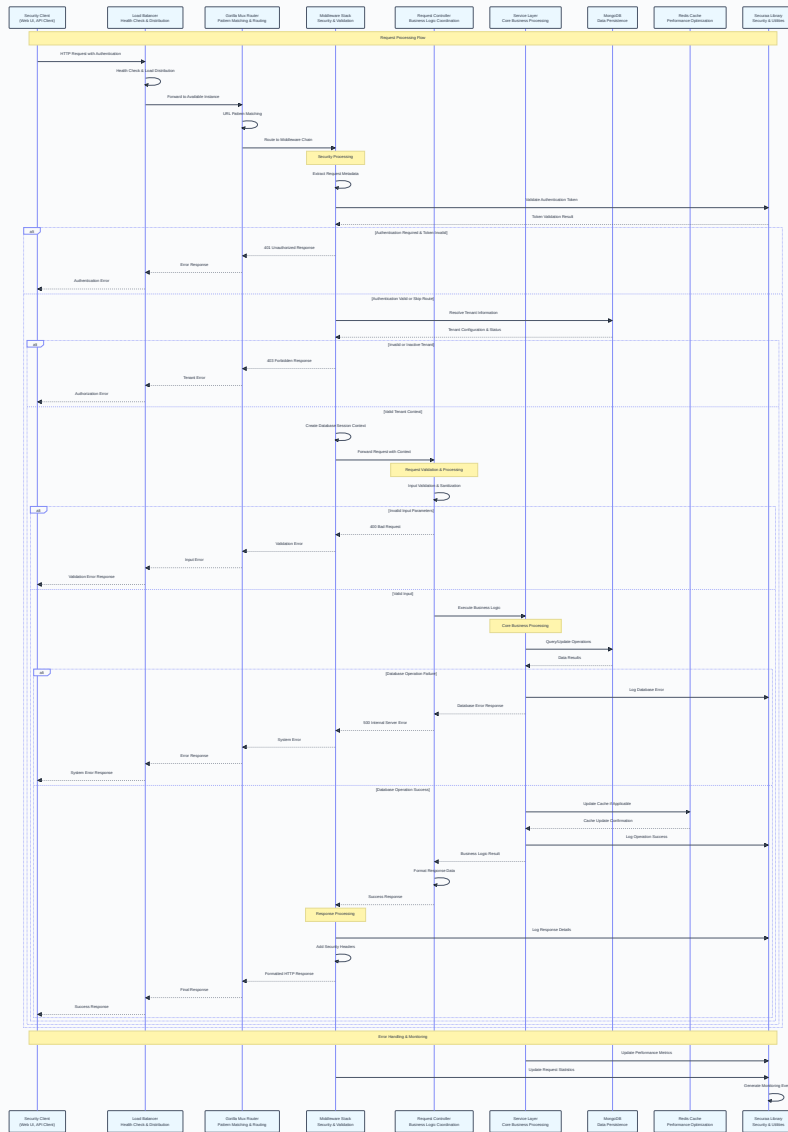
ENDPOINT PATTERN	HTTP METHOD	CONTROLLER	FUNCTIONAL PURPOSE	REQUEST/RESPONSE CHARACTERISTICS
/apps/v1/app	DELETE	CustomAppController	Application Deletion Handles complete application removal including task status updates and cache invalidation	Request: JSON body with integration_id and tenant_code Response: Success confirmation with cleanup status
/apps/v1/integration-name/{tenant-code}	GET	CustomAppController	Integration List Retrieval Returns filtered list of active integrations for specified tenant	Request: Path parameter for tenant context Response: Array of integration names and metadata
/apps/v1/app/generic	POST	GenericAppController	Generic Application Upsert Creates new or updates existing generic applications with multipart form support	Request: Multipart form with JSON fields, file uploads Response: Application ID and creation/update status
/apps/v1/task/generic	POST	GenericAppController	Generic Task Management Handles task creation, modification, and configuration	Request: JSON task definition with validation rules Response: Task ID and configuration status
/apps/v1/task/generic/{tenant-code}/{integration-id}/{task-id}	GET	GenericAppController	Task Information Retrieval Returns detailed task configuration and metadata	Request: Path parameters for context Response: Complete task definition and runtime info
/apps/v1/task/verify-generic	POST	GenericAppController	Task Validation Validates task configuration and execution prerequisites	Request: Task definition for validation Response: Validation results with error details
/apps/v1/task/fields/{tenant-code}/{task-id}	GET	GenericAppController	Task Output Fields Returns available output fields	Request: Task identification parameters Response: Field definitions and data types

ENDPOINT PATTERN	HTTP METHOD	CONTROLLER	FUNCTIONAL PURPOSE	REQUEST/RESPONSE CHARACTERISTICS
			for task result consumption	
/apps/v1/task/output-fields/{tenant-code}/{task-id}	GET	GenericAppController	Task Array Output Fields Returns array-structured output fields for complex results	Request: Task context parameters Response: Array field specifications
/apps/v1/app/export	POST	ExportController	Application Export Exports complete application configurations with dependency resolution	Request: List of integration IDs and export options Response: Serialized application data
/apps/v1/app/import	POST	ExportController	Application Import Imports application configurations with validation and conflict resolution	Request: Serialized application data and import options Response: Import status and conflict resolution results
/apps/v1/task/input/{tenant-code}/{task-id}	GET	IntegrationController	Task Input Schema Returns input parameter specifications for task execution	Request: Task identification Response: Input schema with validation rules
/casepanel/v1/events/{tenant-code}/{instance-id}/{case-id}/{off-set}/{limit}	GET	EventsController	Event Retrieval Returns paginated security events with filtering capabilities	Request: Context parameters and pagination options Response: Paginated event list with metadata
/apps/v1/case-ingestion-start-time/{tenant-code}/{instance-id}	GET	IntegrationController	Ingestion Timing Returns optimal start time for case ingestion processes	Request: Instance context Response: Calculated start time with optimization data
/apps/v1/task/generic-validate	POST	GenericAppController	Task Deletion Validation Validates if task can be safely deleted without	Request: Task identification and validation criteria Response: Validation status with dependency analysis

ENDPOINT PATTERN	HTTP METHOD	CONTROLLER	FUNCTIONAL PURPOSE	REQUEST/RESPONSE CHARACTERISTICS
			breaking dependencies	
<code>/updateUserSessionExpiretime/{accesstoken}/{exiprytime}/</code>	GET	UtilityController	Session Management Updates user session expiration times for active tokens	Request: Token and expiry time parameters Response: Session update confirmation

4.2 Request/Response Processing Flow

The request processing architecture is designed to handle complex security workflows while maintaining performance and reliability. Each request passes through multiple layers of processing, validation, and transformation before generating a response.



4.2.1 Request Processing Layer Details

Load Balancer Layer: The load balancer performs intelligent traffic distribution based on instance health, current load, and response times. It continuously monitors backend instances and automatically removes unhealthy instances from the rotation. The load balancer also implements connection pooling and keep-alive mechanisms to optimize network performance.

Router Layer: The Gorilla Mux router provides sophisticated URL pattern matching capabilities, supporting complex route parameters, regular expressions, and conditional routing. The router maintains route performance metrics and supports dynamic route registration for plugin-based extensibility.

Middleware Layer: The middleware stack implements cross-cutting concerns in a layered approach:

- **Security Middleware:** Handles authentication, authorization, and security header injection
- **Logging Middleware:** Provides comprehensive request/response logging with correlation IDs
- **Tenant Middleware:** Resolves tenant context and database session management
- **Validation Middleware:** Implements input sanitization and basic validation rules
- **Monitoring Middleware:** Collects performance metrics and health indicators

Controller Layer: Controllers act as orchestrators, coordinating between the HTTP protocol and business logic. They handle input validation, format conversion, and response generation while delegating actual business processing to service classes.

Service Layer: The service layer contains the core business logic and domain-specific processing. Services are stateless and designed for easy testing and reuse. They coordinate complex operations, handle business rule validation, and manage transactional boundaries.

4.3 Authentication and Authorization Architecture¶

The system implements a multi-layered security model that provides comprehensive authentication and authorization capabilities while maintaining performance and usability.



4.3.1 Multi-Factor Authentication Flow

The authentication system supports multiple authentication methods that can be used independently or in combination for enhanced security:

JWT Token Authentication: Stateless authentication using JSON Web Tokens with digital signatures. Tokens include user identity, tenant context, role information, and expiration data. The system supports both access tokens for API calls and refresh tokens for session management.

SAML Integration: Enterprise-grade Single Sign-On integration supporting multiple identity providers. The SAML implementation includes attribute mapping, role synchronization, and session federation capabilities.

Session Management: Comprehensive session lifecycle management including creation, validation, renewal, and termination. The system maintains session state in Redis for performance while providing fallback mechanisms for high availability.

API Key Authentication: Service-to-service authentication using API keys with scope-based access control. API keys support rate limiting, usage analytics, and automatic rotation capabilities.

5. Scalability and Performance Architecture¶

5.1 Horizontal Scalability Strategy¶

The Securaa Custom Services architecture is designed for horizontal scalability from the ground up, supporting growth from small single-tenant deployments to large-scale multi-tenant enterprise environments. The scalability strategy encompasses multiple dimensions including application instances, database capacity, caching performance, and infrastructure resources.

5.1.1 Application Layer Scaling¶

Stateless Application Design: The core application is designed to be completely stateless, with all session and state information stored in external systems (Redis for sessions, MongoDB for persistent data). This stateless design enables seamless horizontal scaling by allowing any application instance to handle any request without dependency on local state.

Load Balancer Distribution: Multiple application instances operate behind sophisticated load balancers that distribute traffic based on various algorithms including round-robin, least connections, and weighted routing. The load balancers continuously monitor instance health and automatically remove unhealthy instances from rotation.

Auto-scaling Capabilities: The system supports automatic scaling based on various metrics including CPU utilization, memory usage, request queue depth, and custom business metrics. Auto-scaling policies can be configured to handle predictable load patterns (time-based scaling) and unpredictable traffic spikes (metric-based scaling).

Container Orchestration: Deployment using container orchestration platforms (Docker Swarm) provides advanced scaling capabilities including:

- Service auto-scaling based on resource metrics
- Horizontal service scaling with replicas
- Resource optimization
- Cluster management for infrastructure expansion

5.1.2 Scalability Architecture Diagram¶



Syntax error in text

internal version 10.0.0

5.2 Database Performance and Scaling¶

5.2.1 MongoDB Scaling Strategies¶

Replica Set Configuration: The primary database scaling approach uses MongoDB replica sets with optimized read preferences. The replica set configuration includes a primary node for write operations and multiple secondary nodes for read operations, with automatic failover capabilities.

Sharding Implementation: For large-scale deployments, the system supports MongoDB sharding based on tenant identifiers. This approach provides horizontal database scaling while maintaining tenant data isolation. Each shard contains a complete replica set for high availability.

Index Optimization: Comprehensive indexing strategies are implemented based on query patterns:

- Compound indexes for complex queries
- Text indexes for search functionality
- TTL indexes for automatic data expiration
- Sparse indexes for optional fields
- Partial indexes for conditional data

Connection Pool Optimization: Database connection pools are tuned for optimal performance with configurations adjusted for:

- Maximum connection limits based on instance capacity
- Minimum connection pools for consistent performance
- Connection timeout settings for reliability
- Health check intervals for connection validity

5.2.2 Caching Performance Strategy¶

Multi-Level Caching: The system implements a sophisticated multi-level caching strategy:

L1 Cache (Application Level): In-memory caching within each application instance for frequently accessed configuration data, reducing database queries for common operations.

L2 Cache (Distributed Redis): Shared caching across all application instances using Redis cluster for session data, integration listings, and computed results.

L3 Cache (CDN/Edge): Content delivery network caching for static assets, API responses, and geographically distributed content.

Cache Invalidation Strategy: Intelligent cache invalidation ensures data consistency while maximizing cache hit rates:

- Event-driven invalidation for data changes
- Time-based expiration for ephemeral data
- Tag-based invalidation for related data groups
- Conditional invalidation based on business rules

5.3 Performance Optimization Techniques¶

5.3.1 Application Performance Optimization¶

Goroutine Management: Optimal use of Go's concurrency model with goroutine pools for handling concurrent requests while avoiding goroutine leaks and excessive memory usage.

Memory Management: Efficient memory usage through:

- Object pooling for frequently created objects
- Memory-mapped files for large data sets
- Garbage collection tuning for low latency
- Memory profiling and optimization

Database Query Optimization:

- Prepared statement usage for repeated queries
- Batch operations for bulk data processing
- Query result caching for expensive operations
- Connection pooling for database efficiency

Network Optimization:

- HTTP/2 support for multiplexed connections
- Compression for large responses
- Keep-alive connections for persistent clients
- Request/response optimization

5.3.2 Performance Monitoring and Tuning¶

Real-time Performance Metrics: Comprehensive performance monitoring including:

- Response time percentiles (P50, P95, P99)
- Throughput measurements (requests per second)
- Error rate tracking and alerting
- Resource utilization monitoring (CPU, memory, network)

Capacity Planning: Data-driven capacity planning using historical metrics and predictive analytics to anticipate scaling needs before performance degradation occurs.

Performance Testing: Regular performance testing including:

- Load testing for normal usage patterns
- Stress testing for peak capacity determination
- Endurance testing for long-running stability
- Spike testing for sudden load increases

6. Deployment and Infrastructure Architecture¶

6.1 Container and Orchestration Strategy¶

The deployment architecture leverages containerization and orchestration technologies to provide scalable, reliable, and maintainable infrastructure. The system is designed to run efficiently in various environments from development laptops to large-scale production clusters.

6.1.1 Containerization Approach¶

Docker Container Design: The application is packaged as a lightweight Docker container based on minimal base images for security and performance. The container includes only essential runtime dependencies, reducing attack surface and improving startup times.

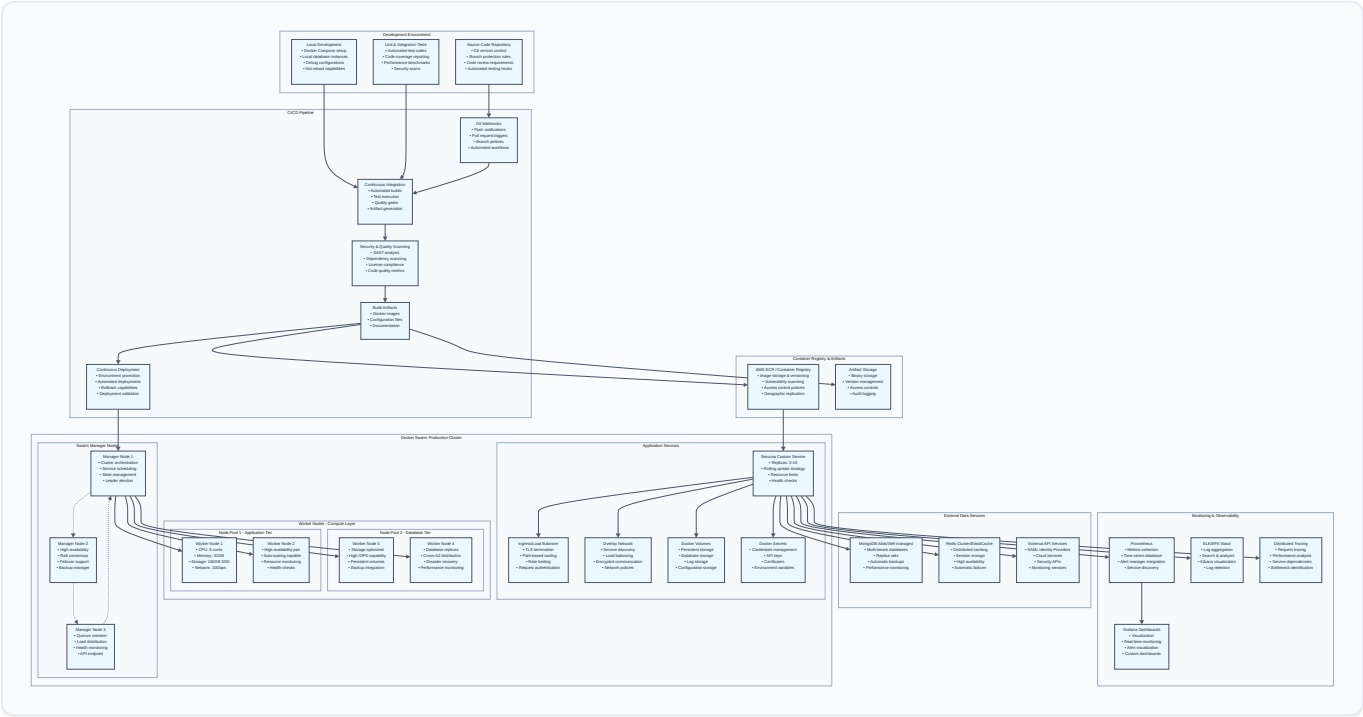
Multi-stage Build Process: The Docker build process uses multi-stage builds to optimize container size:

- Build stage: Includes development tools and dependencies for compilation
- Runtime stage: Contains only production runtime and compiled artifacts
- Security scanning: Automated vulnerability scanning during build process

Container Security: Security-hardened containers following best practices:

- Non-root user execution
- Read-only filesystem where possible
- Minimal package installation
- Regular base image updates
- Security scanning integration

6.1.2 Docker Swarm Deployment Architecture¶



6.2 Environment Management and Deployment Pipeline

6.2.1 Multi-Environment Strategy

Development Environment: Local development environments using Docker Compose for rapid development and testing. Includes hot-reload capabilities, debug configurations, and local database instances.

Staging Environment: Production-like environment for integration testing, performance validation, and user acceptance testing. Mirrors production configuration with reduced scale.

Production Environment: High-availability production environment with full monitoring, backup, and disaster recovery capabilities. Includes blue-green deployment support for zero-downtime updates.

6.2.2 Deployment Strategies

Rolling Deployments: Default deployment strategy using Docker Swarm rolling updates to ensure zero downtime during application updates. Includes configurable update strategies and rollback capabilities.

Blue-Green Deployments: For critical updates, blue-green deployments provide instant rollback capabilities and complete environment validation before traffic switching.

Canary Deployments: Gradual traffic shifting for new versions, allowing real-world validation with minimal risk exposure.

7. Monitoring, Observability, and Operational Excellence

7.1 Comprehensive Monitoring Strategy

The monitoring and observability architecture provides comprehensive visibility into system performance, security events, and business operations. This multi-layered approach ensures rapid issue detection, root cause analysis capabilities, and proactive performance optimization.

7.1.1 Monitoring Architecture

Application Performance Monitoring (APM): Comprehensive application-level monitoring including request tracing, error tracking, performance profiling, and user experience monitoring.

Infrastructure Monitoring: System-level monitoring covering resource utilization, network performance, storage usage, and infrastructure health.

Business Metrics Monitoring: Domain-specific monitoring for security operations including integration success rates, event processing performance, and tenant utilization patterns.

Security Event Monitoring: Specialized monitoring for security-related events including authentication failures, authorization violations, and suspicious activity patterns.

7.1.2 Observability Stack Architecture



System error in test

internal version 10.0.5

7.2 Operational Excellence Practices¶

7.2.1 Site Reliability Engineering (SRE) Practices¶

Service Level Objectives (SLOs): Well-defined SLOs for critical system metrics:

- **Availability:** 99.9% uptime for production services
- **Latency:** 95th percentile response time under 500ms
- **Error Rate:** Less than 0.1% error rate for API calls
- **Throughput:** Sustained handling of peak load with auto-scaling

Error Budget Management: Systematic tracking of error budgets to balance feature velocity with system reliability. Error budget depletion triggers engineering focus on reliability improvements.

Chaos Engineering: Regular chaos engineering exercises to validate system resilience, including:

- Network partition testing
- Database failover scenarios
- Container termination testing
- Load spike simulations

7.2.2 Incident Response and Management¶

Incident Response Procedures: Comprehensive incident response procedures including:

- Automated incident detection and alerting
- Clear escalation procedures and on-call rotations
- Incident command structure for major incidents
- Post-incident review processes and improvement tracking

Disaster Recovery Planning: Complete disaster recovery procedures including:

- Regular backup validation and restoration testing
- Cross-region failover capabilities
- Recovery time and recovery point objectives
- Communication plans for extended outages

Business Continuity: Business continuity planning covering:

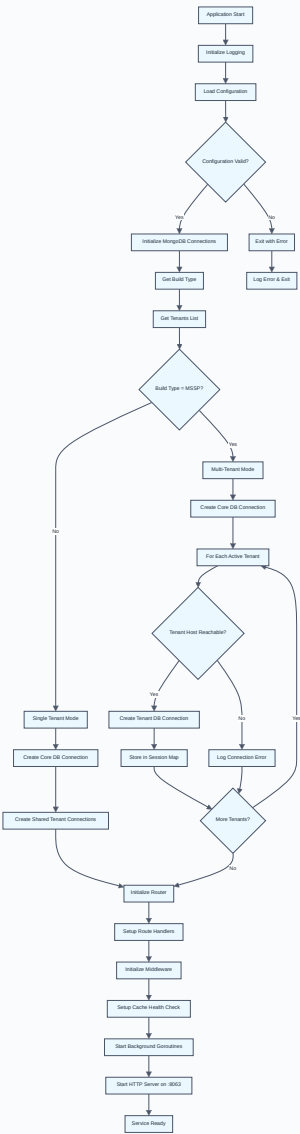
- Critical business function identification
- Alternative operating procedures
- Vendor and supplier contingency plans
- Communication strategies for stakeholders

8. Comprehensive Process Flows and Workflows¶

This section provides detailed process flowcharts that illustrate the key workflows and operational processes within the Securaa Custom Services system. These flowcharts complement the architectural diagrams by showing the step-by-step execution paths for critical system operations.

8.1 Application Startup and Initialization Process¶

The application startup process is critical for ensuring proper system initialization, database connectivity, and service readiness. This process handles both single-tenant and multi-tenant configurations while establishing all necessary connections and background services.



Startup Process Explanation: The application startup follows a carefully orchestrated sequence ensuring all dependencies are properly initialized. The process begins with logging setup and configuration validation, followed by database connection establishment. The system then determines the deployment mode (single-tenant vs. multi-tenant) and configures database sessions accordingly. Finally, HTTP routing, middleware, and background services are initialized before the server starts accepting requests.

8.2 HTTP Request Processing and Middleware Chain¶

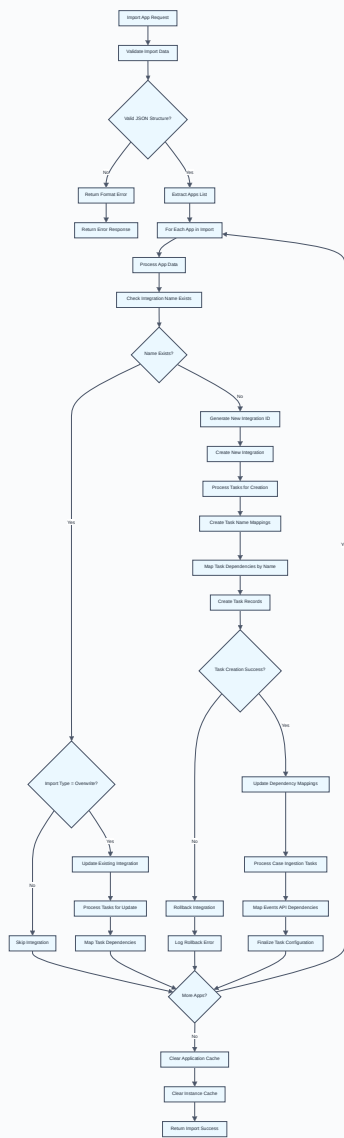
The HTTP request processing pipeline demonstrates how requests flow through various middleware layers before reaching business logic controllers. This process ensures proper authentication, authorization, and tenant isolation for every request.

8.4 Application Export Process

Confidential

tracting integration and task configurations
replacing internal IDs with portable name
t environments.

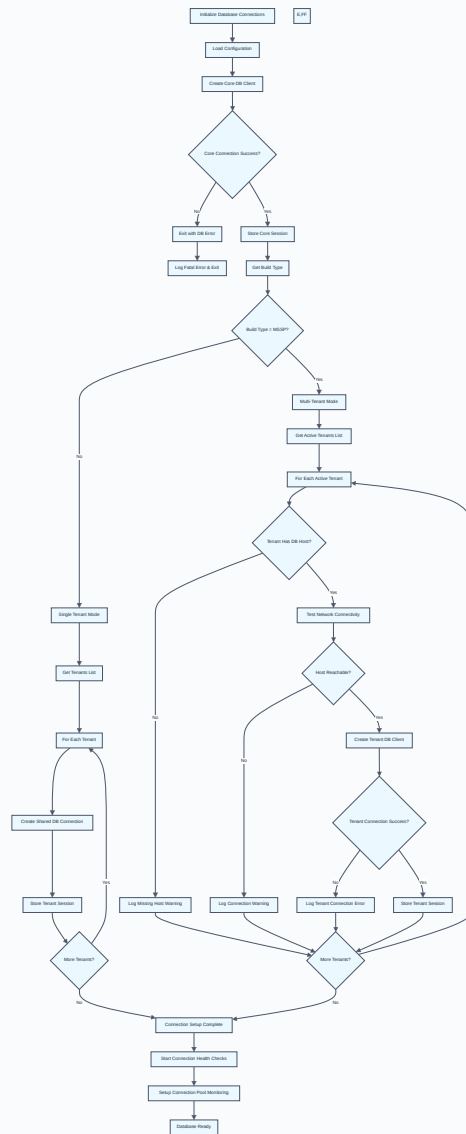
the application configurations. This workflow



Import Process Explanation: The import workflow processes exported application packages, handling both creation of new applications and updates to existing ones. The system carefully maps task dependencies by name, maintains referential integrity, and includes rollback capabilities for error scenarios. Cache invalidation ensures that imported changes are immediately visible across the system.

8.6 Database Connection Management and Health Monitoring

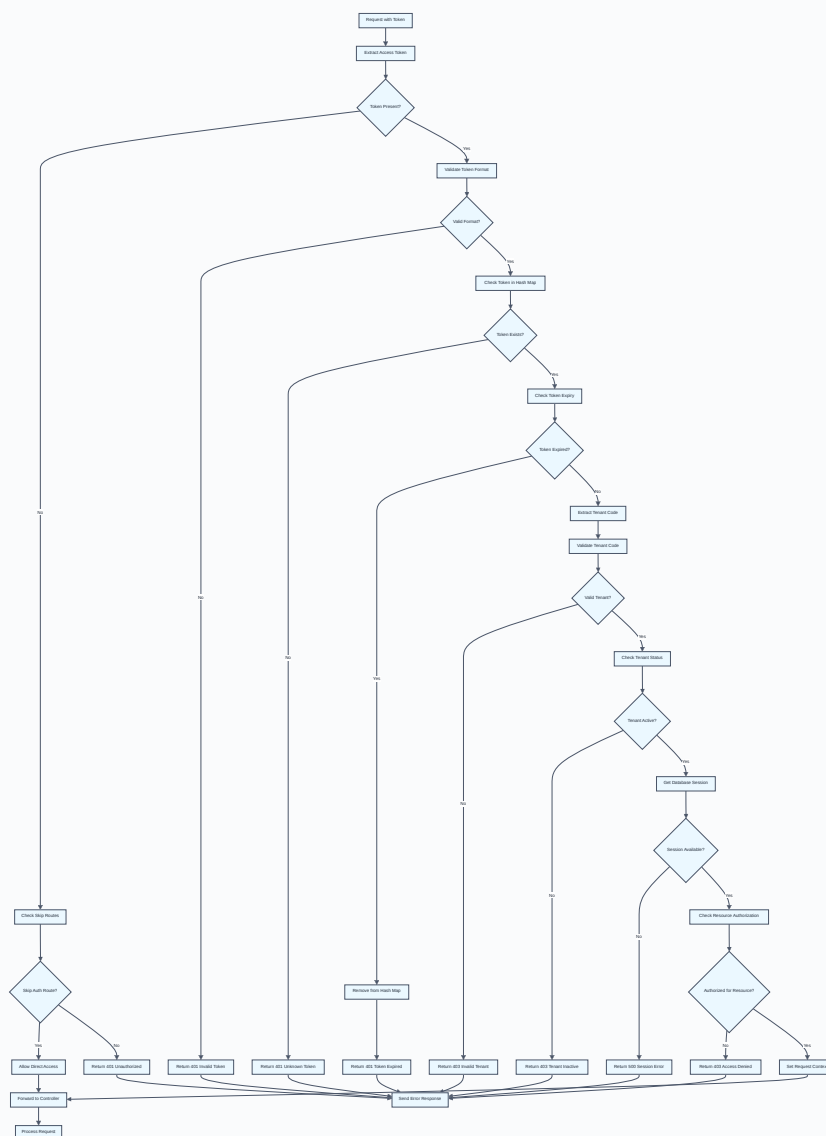
Database connection management is critical for system reliability, especially in multi-tenant environments where connections must be established and maintained for multiple tenant databases.



Database Connection Management Explanation: The connection management process handles both single-tenant and multi-tenant database configurations. For multi-tenant deployments, the system attempts to establish connections to all active tenant databases, gracefully handling connection failures while ensuring the system remains operational. Health monitoring and connection pooling ensure optimal database performance and reliability.

8.7 Authentication and Authorization Workflow

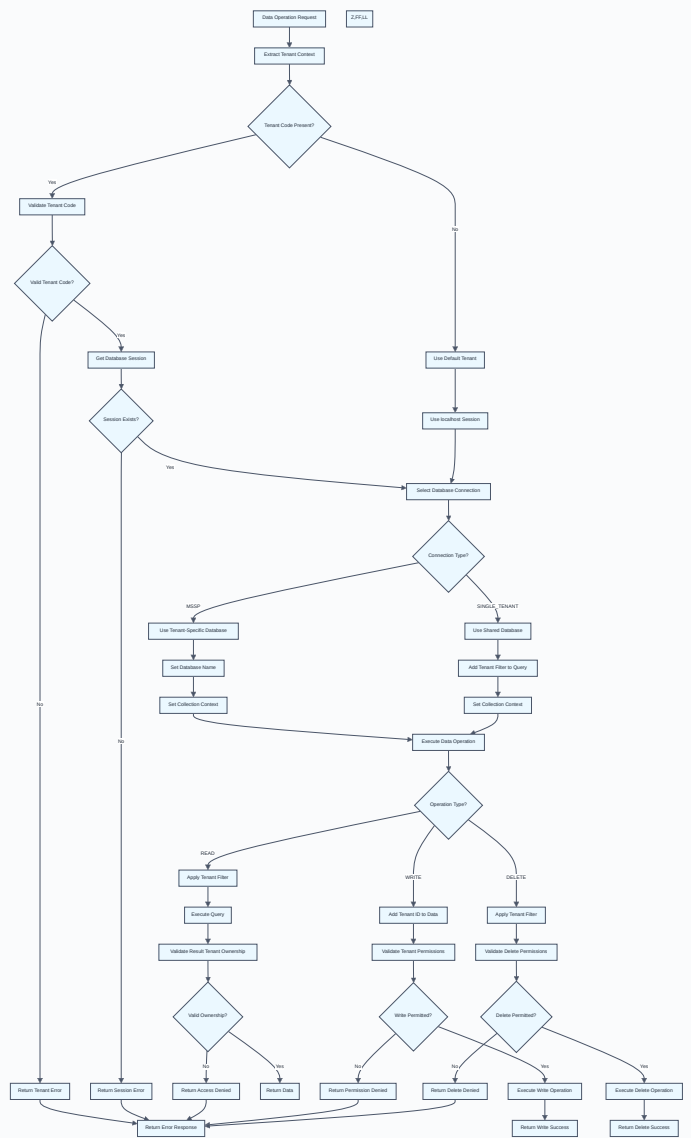
The authentication and authorization process ensures secure access to system resources while supporting both authenticated and public endpoints.



Authentication/Authorization Explanation: This workflow implements comprehensive security controls for all system access. The process validates access tokens, checks expiration, resolves tenant contexts, and ensures users have appropriate permissions for requested resources. The system supports both authenticated endpoints and public routes (such as health checks) while maintaining strict security boundaries.

8.8 Multi-Tenant Data Isolation Process

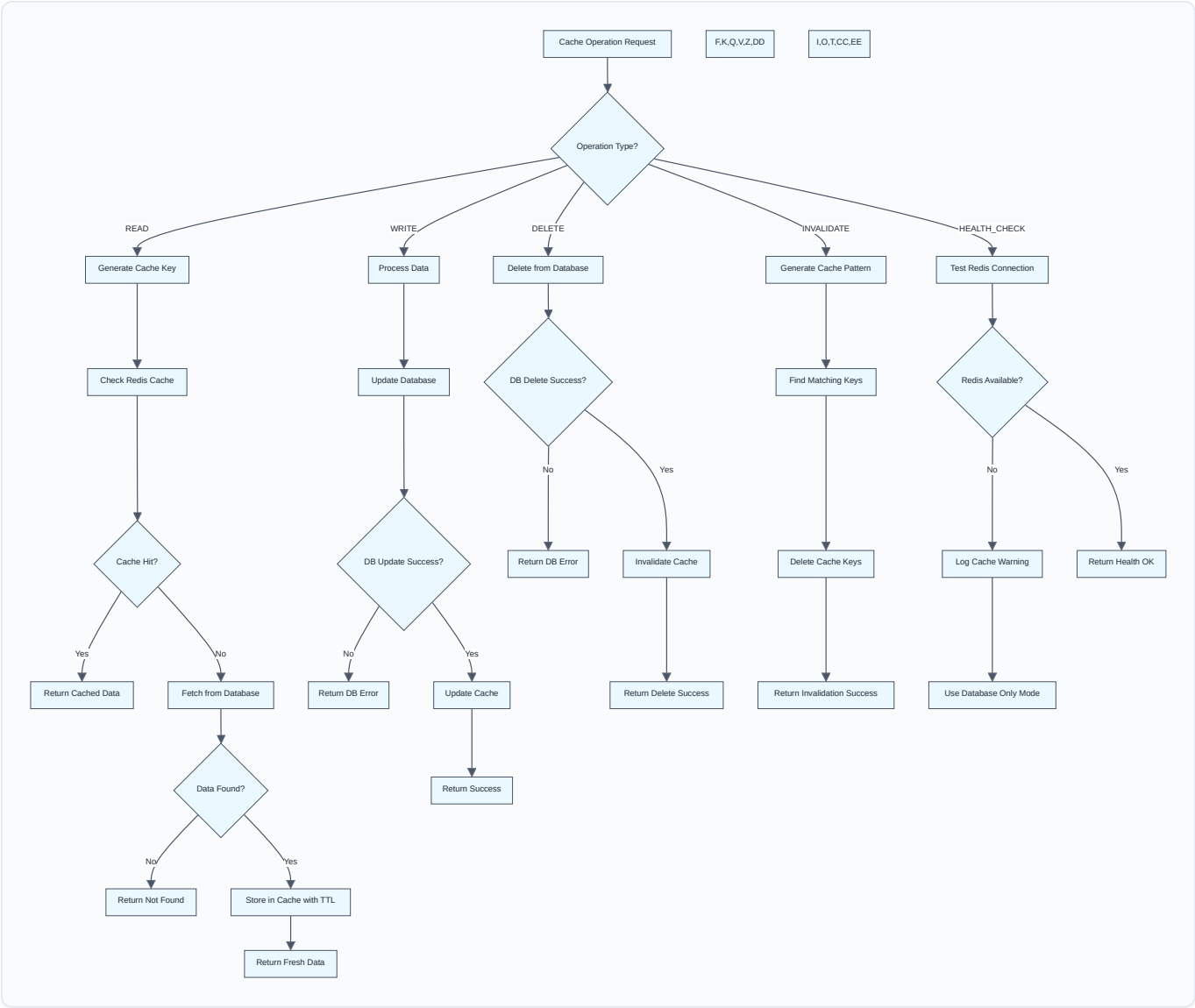
Data isolation is fundamental to multi-tenant security, ensuring that tenant data remains completely separated and secure from unauthorized access.



Multi-Tenant Data Isolation Explanation: This critical security workflow ensures complete data separation between tenants. The system supports two isolation models: separate databases (MSSP mode) and shared database with tenant filtering. All data operations include tenant validation to prevent cross-tenant data access, maintaining strict security boundaries while supporting efficient multi-tenant operations.

8.9 Cache Management and Performance Optimization¶

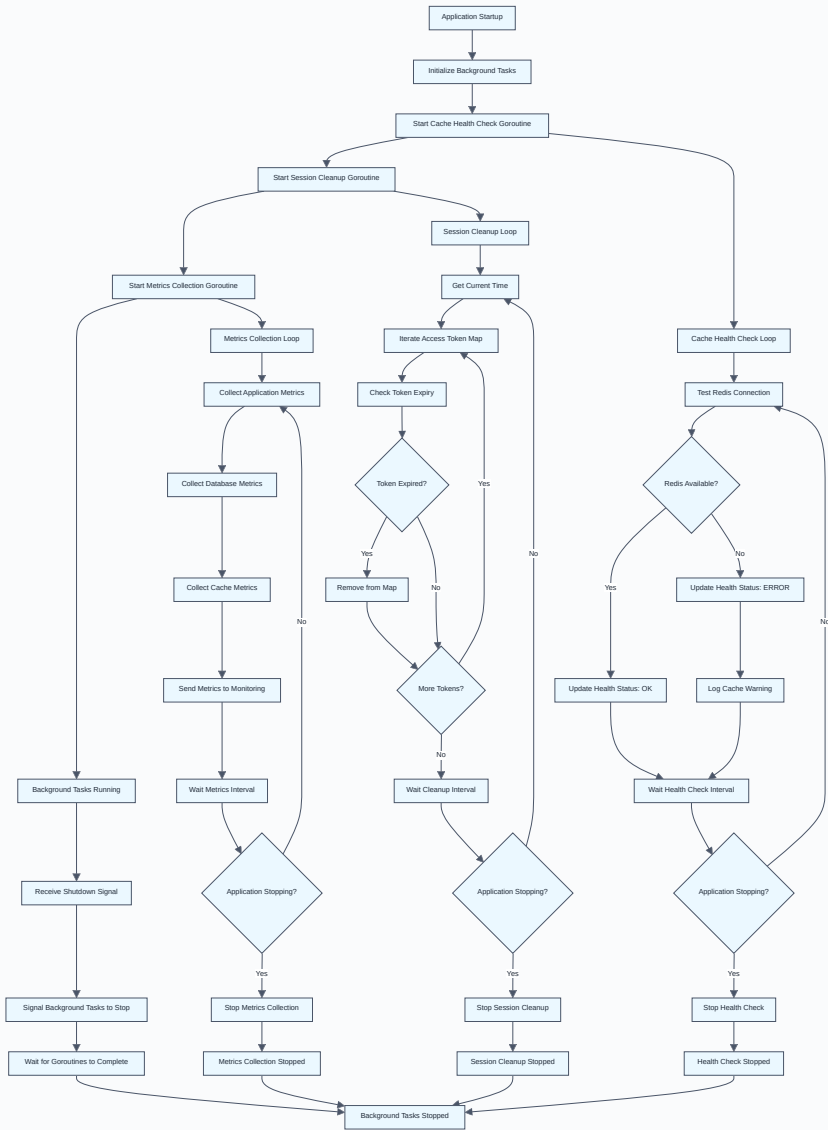
The caching system optimizes performance while maintaining data consistency through intelligent cache management strategies.



Cache Management Explanation: The caching workflow implements a write-through cache pattern with automatic fallback to database-only mode when cache services are unavailable. The system handles cache hits/misses, maintains data consistency through coordinated updates, and provides intelligent invalidation patterns for related data. Health monitoring ensures system resilience during cache outages.

8.10 Background Task Processing and System Maintenance¶

Background services handle essential system maintenance tasks including health monitoring, session cleanup, and metrics collection.



Background Task Processing Explanation: The system runs essential background services using Go goroutines for health monitoring, session management, and metrics collection. These services operate independently with proper graceful shutdown handling, ensuring system cleanliness and reliable monitoring data. The background tasks provide the foundation for operational excellence and system observability.

This comprehensive High-Level Design document provides detailed architectural guidance for the Securaa Custom Services system, covering all aspects from system overview through operational excellence and detailed process workflows. The architecture is designed to be scalable, secure, and maintainable while meeting the complex requirements of modern security automation platforms.