Final Project Report

Adv. Cloud Computing – CSCI 5409

Table of Contents

DocuShare: Secure File Sharing Application

Introduction

1. Project Overview

DocuShare is a secure file sharing application designed to enable users to upload, manage, and share files with enhanced security and control features. The application allows users to create shareable links with customizable security settings, track file access, and manage permissions for shared content.

2. Core Functionality

- **Secure File Storage:** Users can upload and store files securely in the cloud
- **Customizable Sharing Options:** Create shareable links with options for:
    - Password protection
    - Expiration dates
    - Download/view limits
    - Recipient identification requirements
- **Access Control:** Manage who can access files and revoke access when needed
- **Access Tracking:** Comprehensive logging of file access including views, downloads, IP addresses, and user agents
- **Analytics:** View statistics on file access and usage patterns

3. Target Users

- **Business Professionals:** Sharing sensitive documents with clients or colleagues
- **Creative Professionals:** Distributing portfolios or large media files
- **Educational Institutions:** Sharing course materials with controlled access
- **Individual Users:** Securely sharing personal files with friends and family

4. Performance Targets

- **Scalability:** Handle thousands of concurrent users and file operations
- **Reliability:** 99.9% uptime for file access and sharing functionality
- **Security:** End-to-end encryption and comprehensive access controls
- **Performance:** Fast upload and download speeds with minimal latency
- **Cost-Efficiency:** Optimize cloud resource usage to minimize operational costs

AWS Service Category Requirements

Selected AWS Services

Service	Category	Purpose
Amazon EC2	Compute	Hosting frontend and backend application components
Amazon S3	Storage	File storage and retrieval
Amazon DynamoDB	Database	NoSQL database for metadata, user info, and access logs
AWS Lambda	Compute	Serverless functions for file cleanup operations
AWS SQS	Application Integration	Message queue for triggering cleanup operations
AWS CloudWatch	Monitoring	Application and infrastructure monitoring
AWS IAM	Security	Identity and access management
Amazon VPC	Networking	Network isolation and security
AWS ECR	Container	Storage and management of Docker container images

Comparison with Alternatives

Compute: Amazon EC2 vs. Alternatives

Service	Pros	Cons	Why Selected/Rejected
Amazon EC2	- Full control over instances - No execution time limits - Customizable configurations - Predictable performance	- Requires server management - Continuous costs regardless of usage - Manual scaling configuration	Selected: Provides reliable, consistent performance for the API backend with full control over the runtime environment
AWS Lambda (fully serverless)	**- Serverless (no server management) - Pay-per-execution- Automatic scaling**	**- Cold start latency - 15-minute execution limit - Limited runtime environment control**	**Partially selected: Used only for scheduled cleanup tasks where event-driven architecture is beneficial**
Amazon ECS/EKS	- Container orchestration - Good for microservices	- Complex setup and management - Higher operational overhead	Rejected: Unnecessary complexity for this application

Storage: Amazon S3 vs. Alternatives

Service	Pros	Cons	Why Selected/Rejected
Amazon S3	- Highly durable (99.999999999%) - Scalable to any size -Built-in versioning - Fine-grained access control - Pre-signed URLs for secure access	- Higher cost for frequent access	Selected: Offers the best combination of durability, scalability, and security features essential for a file sharing application
Amazon EFS	- Shared file system - Good for applications needing file system access	- Higher cost - More complex to manage	Rejected: Overkill for simple file storage and sharing

Database: DynamoDB vs. Alternatives

Service	Pros	Cons	Why Selected/Rejected
Amazon DynamoDB	- Fully managed NoSQL - Millisecond latency - Automatic scaling - Pay-per-use pricing-Compatible with EC2 and Lambda	- Limited query flexibility - Potential for higher costs with improper design	Selected: Perfect fit for the application's need for fast, scalable access to metadata with simple key-value and document storage patterns
Amazon RDS	**- Relational database\n- SQL query capabilities\n- Strong consistency**	**- Requires capacity planning - Higher management overhead - Fixed costs regardless of usage**	**Rejected: Overkill for the simple data structures needed**
MongoDB Atlas	- Document database - Flexible schema - Rich query language	- Third-party service - Potentially higher costs - Additional integration work	Rejected: DynamoDB offers better integration with other AWS services

Messaging: Amazon SQS vs. Alternatives

Service	Pros	Cons	Why Selected/Rejected
Amazon SQS	- Fully managed message queue - Reliable delivery - Dead-letter queue support - Integrates well with Lambda	- Not real-time - No push capabilities	Selected: Perfect for reliable, asynchronous triggering of cleanup operations
Amazon SNS	- Push notifications - Multiple subscribers - Fan-out pattern	- No message persistence - No message ordering	Rejected: Message persistence needed for cleanup operations
Amazon EventBridge	- Event-driven architecture - Rich filtering - Multiple targets	- More complex setup - Overkill for simple triggers	Rejected: SQS provides simpler, more direct integration for this use case

Delivery Model

Selected Delivery Model: Hybrid Architecture with Multi-Tier Security

DocuShare is implemented using a hybrid architecture that combines traditional EC2-hosted services with serverless components in a multi-tier security model. The application is divided into public-facing and private backend components, with clear separation of concerns and network isolation. This model was chosen for the following reasons:

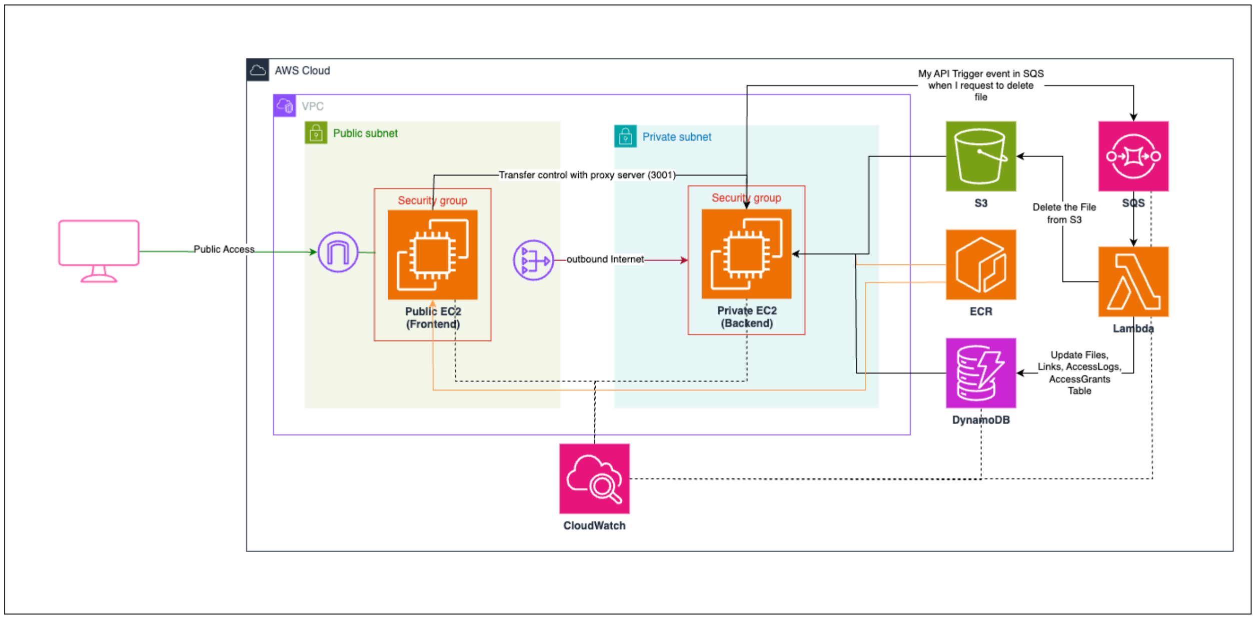
1. **Security Isolation:** The multi-tier approach with public and private subnets provides strong security boundaries, keeping sensitive backend operations isolated from direct internet access.
2. **Performance Consistency:** EC2 instances provide consistent performance without cold start issues, ensuring responsive API endpoints for all user interactions.
3. **Control and Customization:** Full control over the runtime environment allows for custom configurations, middleware, and optimizations that might be limited in a purely serverless environment.
4. **Selective Serverless Integration:** Using Lambda for specific tasks like cleanup operations leverages serverless benefits where they make the most sense, particularly for scheduled, infrequent tasks.
5. **Cost Optimization:** The hybrid approach balances performance needs with cost **efficiency, using** always-on resources where needed and event-driven components where appropriate.

Alternative Delivery Models Considered

1. **Fully Serverless Architecture:**
   - Pros: Pay-per-use pricing, automatic scaling, no server management
   - Cons: Cold start latency, execution time limits, less control over environment
   - Rejection Reason: The need for consistent performance and control over the runtime environment for the core API functionality outweighed the benefits of a fully serverless approach
2. **Single-Tier EC2 Architecture:**
   - Pros: Simpler setup, fewer components to manage
   - Cons: Weaker security posture, less separation of concerns
   - Rejection Reason: Security requirements demanded proper isolation of backend services and data access
3. **Container-Based Microservices:**
   - Pros: Isolation between services, easier scaling of individual components
   - Cons: Higher operational complexity, steeper learning curve
   - Rejection Reason: The additional complexity of container orchestration wasn't justified for the application's requirements

Final Architecture

Architecture Diagram



Component Integration

1. **Client Layer**:
   - Web clients connect to the public EC2 instance in the public subnet
   - The public EC2 serves as both a frontend server and a proxy to the backend
2. **Application Tier**:
   - **Public EC2 (Frontend)**:
   - Handles initial client requests
   - Serves static content
   - Proxies API requests to the private backend
   - Located in a public subnet with internet access
3. **Private EC2 (Backend)**:
   - Hosts the core application logic
   - Processes file operations and business rules
   - Communicates with data stores (S3, DynamoDB)
   - Located in a private subnet with no direct internet access
   - Communicates with the frontend through port 3001
4. **Data Tier**:
   - **S3**: Stores the actual file data
   - **DynamoDB**: Stores metadata, user information, and access logs
   - **ECR**: Stores Docker images for the application components

5. **Event Processing**:
    - **SQS**: Message queue for triggering cleanup operations
    - **Lambda**: Processes cleanup events from SQS
    - Handles deletion of files from S3 and related metadata from DynamoDB
6. **Monitoring**:
    - **CloudWatch**: Monitors all components and collects logs

Data Storage

1. **File Storage (S3)**:
    - Files are stored in S3 buckets organized by user ID and file ID
    - Each file has a unique path: {userId}/{fileId}/{fileName}
    - S3 bucket named "docushare-storage" as defined in the Terraform configuration
2. **Metadata Storage (DynamoDB)**:
    - **Users Table**: User accounts, authentication, and subscription information
    - **Files Table**: File metadata (name, size, type, owner, etc.)
    - **Links Table**: Shareable link configurations and security settings
    - **Access Logs Table**: Detailed logs of all file access attempts
    - **Access Grants Table**: Specific access permissions for identified recipients
    - **Subscription Plans Table**: Available subscription tiers and features

Programming Languages and Technologies

1. **TypeScript/Node.js**:
    - Used for all backend services and Lambda functions
    - Chosen for:
        1. Strong typing to reduce runtime errors
        2. Large ecosystem of libraries
        3. Excellent AWS SDK support
        4. Shared code between frontend and backend
        5. Efficient execution in both EC2 and Lambda environments
2. **Express.js**:
    - Used for API routing and middleware
    - Chosen for its simplicity, flexibility, and widespread adoption
    - Well-suited for RESTful API development
3. **React/Next.js** (Frontend):
    - Used for the web client interface
    - Chosen for:
        1. Server-side rendering capabilities
        2. Optimized performance
        3. Component-based architecture

Cloud Deployment

The application is deployed to AWS using the following approach:

1. **Infrastructure as Code (IaC)**:
    - Terraform used to define and provision all infrastructure
    - Enables consistent, repeatable deployments
    - Facilitates environment replication (dev, staging, production)
    - Includes VPC, subnets, security groups, IAM roles, and other resources
2. **CI/CD Pipeline**:
    - GitHub Actions for continuous integration and deployment
    - Automated testing before deployment
    - Staged deployments with approval gates
3. **Network Architecture**:
    - VPC with public and private subnets across multiple availability zones
    - Internet Gateway for public subnet access
    - NAT Gateway for private subnet outbound access
    - Security groups with specific ingress/egress rules
4. **EC2 Deployment**:
    - Application code is packaged and deployed to EC2 instances
    - Instance profiles attach appropriate IAM roles
    - Security groups control network access
5. **Lambda Deployment**:
    - Cleanup Lambda function is packaged and deployed via Terraform
    - SQS event source mapping triggers Lambda execution

Comparison with Course Architectures

The DocuShare architecture represents a hybrid approach that differs from both the purely serverless and traditional three-tier architectures taught in the course:

1. **Differences from Serverless Architecture**:
    - Core API functionality runs on EC2 rather than Lambda
    - More direct control over the runtime environment
    - No cold start issues or execution time limits for API operations
    - Potentially higher baseline costs but more predictable performance
2. **Differences from Traditional Three-Tier Architecture**:
    - Still leverages serverless components (Lambda) for specific tasks
    - Uses managed services (DynamoDB, S3) rather than self-managed databases
    - Implements a more sophisticated network security model with public/private subnets
    - More cloud-native approach to storage and scheduled tasks

This hybrid approach is justified by the specific requirements of DocuShare:

1. Need for consistent, low-latency API responses for file operations
2. Complex business logic that benefits from a traditional application structure
3. Strong security requirements necessitating proper network isolation
4. Scheduled cleanup operations that are well-suited to serverless execution
5. Desire to leverage the best aspects of both architectural patterns

Security Consideration

Data Security at All Layers

Network Layer Security

1. **VPC Isolation**: Complete network isolation with public and private subnets
2. **Security Groups**: Tightly controlled access with specific ingress/egress rules
3. Public EC2: Only allows HTTP (80), HTTPS (443), SSH (from restricted CIDR blocks), and backend traffic (3001)
4. Private EC2: Only allows traffic from the public EC2 security group on specific ports (3001, 22)
5. **NAT Gateway**: Controlled outbound access for private subnet
6. **Internet Gateway**: Restricted to public subnet only

Application Layer Security

1. **Public/Private Separation**: Frontend in public subnet, backend in private subnet
2. **Proxy Pattern**: Frontend EC2 acts as a gateway to backend services
3. **HTTPS Enforcement**: All communications use TLS 1.2+
4. **JWT Authentication**: Secure token-based authentication
5. **Input Validation**: Sanitizes all incoming requests
6. **Error Handling**: Prevents information leakage through errors

Data Layer Security

1. **S3 Bucket Policies**: Restrict access to authorized principals only
2. **S3 Server-Side Encryption**: All files encrypted at rest with AES-256
3. **Pre-signed URLs**: Time-limited, scoped access to files
4. **DynamoDB Encryption**: All table data encrypted at rest
5. **Access Logging**: Comprehensive logging of all data access

IAM Implementation

Based on the Terraform configuration, a comprehensive IAM strategy with the principle of least privilege has been implemented:

1. **Lambda Role Configuration**:
    - The Lambda function has a dedicated IAM role with specific permissions
    - Permissions are limited to exactly what's needed: DynamoDB operations (Scan, DeleteItem, GetItem), S3 DeleteObject operations, and SQS message processing
    - The S3 permissions are scoped to a specific bucket path (docushare-storage/*)
    - The SQS permissions are limited to the specific queue that triggers the Lambda
2. **EC2 Instance Roles**:
    - **Public EC2 (Frontend) Role**:
        1. Limited to CloudWatch monitoring and ECR access
        2. Includes SSM permissions for secure instance management without SSH keys
        3. No permissions to directly access data stores (S3, DynamoDB)
    - **Private EC2 (Backend) Role**:
        1. Has permissions to access S3, DynamoDB, and SQS
        2. Includes CloudWatch monitoring and ECR access
        3. Includes SSM permissions for secure management
        4. Properly isolated in a private subnet with no direct internet access
3. **Network Security**:
    - VPC with public and private subnets
    - NAT Gateway for controlled outbound access from private subnet
    - Security groups with specific ingress rules

This implementation follows security best practices by:

1. Isolating backend services in private subnets
2. Using specific IAM roles for each component
3. Limiting permissions to only what's necessary
4. Implementing network-level security with security groups
5. Using a proxy pattern where the frontend EC2 acts as a gateway to backend services

Vulnerabilities and Mitigation Strategies

1. **Potential Vulnerability**: Brute force password attacks on protected links
    - **Mitigation**: Rate limiting on password verification endpoints
    - **Mitigation**: Account lockout after multiple failed attempts
    - **Mitigation**: Password hashing with bcrypt and appropriate salt rounds

2. **Potential Vulnerability**: Public EC2 instance compromise

- **Mitigation**: Regular security patches and updates
- **Mitigation**: Principle of least privilege for instance roles
- **Mitigation**: Network security groups limiting access
- **Mitigation**: Isolation of sensitive operations to private subnet

3. **Potential Vulnerability**: Metadata exposure in DynamoDB
    - **Mitigation**: Strict IAM policies limiting table access
    - **Mitigation**: Data filtering before returning to clients
    - **Mitigation**: Encryption of sensitive fields

Additional Security Mechanisms

1. **Authentication**:
    - **JWT Tokens**: Used for API authentication with short expiration times
    - **Password Hashing**: bcrypt with 10 salt rounds for secure password storage
    - **Token Refresh**: Secure token refresh mechanism to maintain sessions
2. **Authorization**:
    - **Resource-Based Access Control**: Each resource (file, link) has explicit owner
    - **Fine-Grained Permissions**: Customizable sharing settings per file
    - **Access Grants**: Explicit permission records for identified recipients
3. **Encryption**:
    - **TLS**: All data in transit encrypted with TLS 1.2+
    - **S3 SSE**: Server-side encryption for all stored files
    - **DynamoDB Encryption**: All table data encrypted at rest
4. **Access Control**:
    - **Pre-signed URLs**: Time-limited access to files
    - **Password Protection**: Optional password layer for shared links
    - **Expiration Dates**: Automatic link expiration
    - **Access Limits**: Configurable view/download limits
5. **Monitoring and Auditing**:
    - **Access Logs**: Comprehensive logging of all file access
    - **CloudWatch Alarms**: Alerts for suspicious activity
    - **CloudTrail**: Tracking of all API calls and administrative actions

Cost Analysis

Up-front Cloud Costs

Service	Usage Assumptions	Estimated Cost
EC2 Reserved Instances	Upfront payment for 1-year RI commitment (2 t3.medium instances)	$420
S3 Storage Setup	Initial bucket creation and lifecycle policy configuration	$0 (no upfront cost)
DynamoDB Tables	Creation of 6 tables with on-demand capacity	$0 (no upfront cost)
VPC Infrastructure	VPC, subnets, NAT Gateway, Internet Gateway setup	$0 (no upfront cost)
Route 53 Domain	Domain registration	$12/year
SSL Certificate	AWS Certificate Manager SSL certificate	$0 (free with ACM)
CloudFront Distribution	CDN setup for content delivery	$0 (no upfront cost)
Initial Data Transfer	Data upload for application deployment	$0-$10 (minimal for initial setup)
Total Up-front Cloud Costs		**$432-$442**

Ongoing Costs (Monthly Estimates)

Service	Usage Assumptions	Estimated Cost
EC2 Instances	1 t3.medium (public) + 1 t3.medium (private), 24/7 operation	$70.08
NAT Gateway	1 NAT Gateway, 24/7 operation	$32.85
Elastic IP	1 Elastic IP (for NAT Gateway)	$0.00 (free when attached)
Lambda	30 invocations/day for cleanup, 512MB memory, avg. 1s duration	$0.02
SQS	30 messages/day, standard queue	$0.00 (within free tier)
S3 Storage	500GB storage	$11.50
S3 Requests	5M PUT/COPY/POST/LIST requests, 20M GET requests	$25.00
DynamoDB	10GB storage, 5M read units, 2M write units	$25.00
Data Transfer	1TB outbound	$90.00

CloudWatch	Logs and monitoring	$10.00
Route 53	DNS queries	$0.50
Total Monthly Cost		**$264.95**

Cost Optimization Alternatives

1. **Serverless Architecture Alternative**
    - **Approach**: Replace EC2 instances with API Gateway + Lambda
    - **Potential Savings**: $70-$100/month on compute costs
    - **Trade-offs**:
        1. Cold start latency impacts user experience
        2. 15-minute execution limit for Lambda functions
        3. More complex development model
    - **Recommendation**: Not recommended for the core application due to performance requirements, but could be considered for low-traffic features
2. **Spot Instances for Development/Testing**
    - **Approach**: Use EC2 Spot instances for non-production environments
    - **Potential Savings**: 70-90% on EC2 costs for development/testing
    - **Trade-offs**: Potential interruptions, not suitable for production
    - **Recommendation**: Implement for development and testing environments only
3. **NAT Gateway Optimization**
    - **Approach**: Replace NAT Gateway with NAT Instance or schedule NAT Gateway usage
    - **Potential Savings**: $15-$30/month
    - **Trade-offs**:
        1. NAT Instance: Lower reliability, manual management
        2. Scheduled usage: Limited internet access during off-hours
    - **Recommendation**: Consider NAT Instance for development environments, keep NAT Gateway for production
4. **S3 Storage Optimization**
    - **Approach**: Implement lifecycle policies to move older files to cheaper storage classes
    - **Potential Savings**: 30-40% on S3 storage costs
    - **Trade-offs**: Higher retrieval costs, potential latency for archived files
    - **Recommendation**: Implement for files older than 30 days
5. **DynamoDB Capacity Mode Optimization**
    - **Approach**: Switch from on-demand to provisioned capacity with auto-scaling
    - **Potential Savings**: 20-40% on DynamoDB costs with predictable workloads
    - **Trade-offs**: Requires capacity planning, potential throttling if under-provisioned

- **Recommendation**: Implement after collecting usage patterns for 2-3 months

Justification for Current Architecture

The current hybrid architecture with EC2 instances in public and private subnets represents a balanced approach to cost, performance, and security:

1. **Performance Justification**:
    - EC2 instances provide consistent, predictable performance without cold start issues
    - Critical for file sharing application where user experience depends on responsive uploads/downloads
    - The cost premium over serverless is justified by the improved user experience
2. **Security Justification**:
    - Multi-tier architecture with public/private subnets provides strong security boundaries
    - The additional cost of NAT Gateway ($32.85/month) is justified by the security benefits
    - Proper isolation of backend services from direct internet access is essential for a file sharing application
3. **Operational Efficiency**:
    - Simplified troubleshooting and debugging compared to distributed serverless architecture
    - Reduced operational complexity justifies slightly higher infrastructure costs
    - More familiar architecture for most development teams, reducing training and support costs
4. **Cost-Effective Hybrid Approach**:
    - Selective use of serverless (Lambda/SQS) for appropriate workloads (cleanup operations)
    - EC2 for consistent workloads, serverless for variable/infrequent workloads
    - Balances performance needs with cost efficiency

Long-term Cost Considerations

1. **Scaling Economics**:
    - As usage grows, the relative cost efficiency of EC2 improves compared to serverless
    - Reserved Instances become more cost-effective with scale
    - At very large scale, considering dedicated hosts or Savings Plans could provide additional savings
2. **Multi-Region Expansion**:
    - Expanding to multiple regions would approximately double infrastructure costs

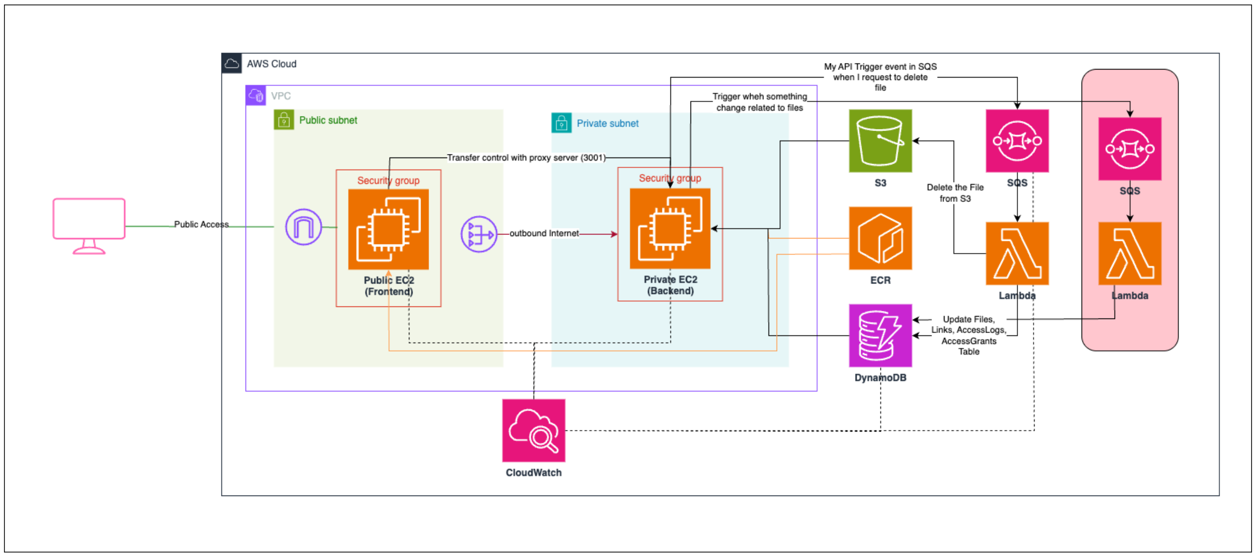
    - Justified by improved global performance and disaster recovery capabilities
    - Should be implemented only when user base requires global presence
3. **Containerization Evolution**:
    - Future migration to container orchestration (ECS/EKS) could improve resource utilization
    - Initial investment in containerization would be offset by improved scaling efficiency
    - Consider when application complexity increases or team size grows

The current architecture provides a cost-effective foundation that balances performance, security, and operational efficiency. While there are opportunities for cost optimization, the chosen approach represents a pragmatic solution that prioritizes reliability and security while maintaining reasonable operational costs.

Future Evolution

Potential Next Features

1. **Enhanced Security**:
    - Client-side encryption options
    - Multi-factor authentication for sensitive files
    - Watermarking for shared documents
    - **AWS Services**: KMS for key management, Cognito for MFA, EC2 with image processing libraries for watermarking
2. **AI-Powered Features**:
    - Automatic content categorization
    - Text extraction and search
    - Image recognition and tagging
    - **AWS Services**: Rekognition for image analysis, Comprehend for text analysis, Kendra for search
3. **Email Notification System**:
    - Automated email notifications for important events:
        1. Link expiration warnings (e.g., 3 days before expiry)
        2. File access notifications (when files are viewed/downloaded)
        3. Security alerts (unusual access patterns, multiple failed password attempts)
        4. Account activity summaries (weekly/monthly usage reports)
    - Customizable notification preferences for users
    - Branded email templates with actionable links
    - Scheduled digest options to reduce email volume
    - **AWS Services**: Amazon SES for email delivery, EventBridge for scheduling, Lambda for email generation, DynamoDB for notification preferences



Infrastructure Evolution

1. **Scaling Strategy**:
   - Implement Auto Scaling groups for both public and private EC2 instances
   - Add load balancers for improved distribution and availability
   - Implement read replicas or DAX for DynamoDB as usage grows
2. **Multi-Region Deployment**:
   - Replicate infrastructure across multiple AWS regions
   - Implement global tables for DynamoDB
   - Use S3 Cross-Region Replication
   - Configure Route 53 for global DNS routing
3. **Enhanced Monitoring and Observability**:
   - Implement AWS X-Ray for distributed tracing
   - Set up detailed CloudWatch dashboards
   - Implement automated alerting and incident response

Conclusion

DocuShare has been successfully implemented as a secure, scalable file sharing application using a hybrid architecture that combines EC2-hosted services with serverless components in a multi-tier security model. This approach provides the reliability and control of traditional server architecture with the flexibility and efficiency of serverless computing where appropriate.

The application meets all the initial requirements for secure file storage, customizable sharing options, and comprehensive access tracking. The multi-tier architecture with public and private subnets provides strong security boundaries, keeping sensitive backend operations isolated from direct internet access.

The security architecture ensures data protection at all layers, from network to application to data storage, with multiple mechanisms to prevent unauthorized access. The IAM implementation follows the principle of least privilege, with specific roles for each component and permissions limited to only what's necessary.

The cost analysis demonstrates that while the hybrid approach has a higher baseline cost than a fully serverless solution, it provides more predictable performance and can be optimized through reserved instances and auto-scaling strategies. With the proposed future enhancements, DocuShare has the potential to evolve into a comprehensive content collaboration platform while maintaining its core strengths in security and ease of use.

The use of Infrastructure as Code with Terraform ensures consistent, repeatable deployments and enables version control of the infrastructure configuration, which is a security best practice. The CI/CD pipeline with GitHub Actions provides automated testing and deployment, enabling rapid iteration and feature additions.

Overall, the DocuShare application represents a well-architected solution that balances security, performance, and cost considerations while providing a solid foundation for future growth and enhancement.