- Design Decisions
  - Serverless vs Traditional Architecture
    - Decision
    - Rationale
    - Alternatives Considered
    - Trade-offs
  - Infrastructure as Code Approach
    - Decision
    - Rationale
    - Alternatives Considered
    - Trade-offs
  - Authentication Strategy
    - Decision
    - Rationale
    - Alternatives Considered
    - Trade-offs
  - Data Storage Strategy
    - Decision
    - Rationale
    - Alternatives Considered
    - Trade-offs
  - AI Integration Approach
    - Decision
    - Rationale
    - Alternatives Considered
    - Trade-offs
  - API Design
    - Decision
    - Rationale
    - Alternatives Considered
    - Trade-offs
  - Frontend Framework
    - Decision
    - Rationale
    - Alternatives Considered
    - Trade-offs
  - Deployment Automation

Design Decisions

This document outlines the key architectural decisions made in the Document Processing Accelerator, explaining the rationale, alternatives considered, and trade-offs.

Serverless vs Traditional Architecture

Decision

The system uses a fully serverless architecture built on AWS Lambda, API Gateway, S3, and DynamoDB.

Rationale

- **Cost Efficiency**: Pay-per-use model eliminates idle infrastructure costs
- **Operational Simplicity**: No server management or patching required
- **Scalability**: Automatic scaling based on actual usage patterns
- **Development Speed**: Faster to market with focus on business logic, not infrastructure

Alternatives Considered

- **Container-based Architecture (ECS/EKS)**:
  - Pro: More control over runtime environment
  - Con: Higher operational complexity and cost
- **EC2-based Architecture**:
  - Pro: Full control over server configuration

    - Con: Requires scaling design, higher maintenance overhead

Trade-offs

- Limited execution time (15-minute maximum for Lambda)
- Cold start latency in low-traffic scenarios
- Less control over underlying infrastructure

Infrastructure as Code Approach

Decision

Infrastructure is managed using a combination of Terraform (core infrastructure) and Serverless Framework (Lambda functions and API Gateway).

Rationale

- **Terraform**: Excellent for defining core AWS infrastructure with rich provider ecosystem
- **Serverless Framework**: Streamlines Lambda deployment with simplified configuration
- **Complementary Tools**: Each handles its domain exceptionally well

Alternatives Considered

- **CloudFormation Only**:
    - Pro: Native AWS integration
    - Con: More verbose syntax, steeper learning curve
- **Terraform Only**:
    - Pro: Single tool for all infrastructure
    - Con: Less streamlined Lambda deployment experience

Trade-offs

- Managing two IaC tools increases complexity
- Potential configuration synchronization challenges
- Learning curve for developers unfamiliar with both tools

Authentication Strategy

Decision

User authentication is handled by AWS Cognito with JWT token validation.

Rationale

- **Managed Service**: No need to build custom authentication systems
- **Security**: Industry standard OAuth2 and OIDC implementations
- **Integration**: Seamless integration with other AWS services
- **Extensibility**: Support for social logins and MFA

Alternatives Considered

- **Custom Auth System**:
  - Pro: Full control over authentication flow
  - Con: Security risks, development and maintenance overhead
- **Third-party Auth Providers (Auth0, Okta)**:
  - Pro: Rich feature set, potential enterprise integrations
  - Con: Additional cost, external dependency

Trade-offs

- Limited customization of login UI
- AWS-specific implementation

- Additional configuration complexity

Data Storage Strategy

Decision

Document metadata is stored in DynamoDB, while document files are stored in S3.

Rationale

- **Separation of Concerns**: Structured data vs unstructured data
- **Query Performance**: Fast metadata queries through DynamoDB indexes
- **Cost Efficiency**: S3 optimized for large object storage
- **Durability**: Both services offer 99.999999999% durability

Alternatives Considered

- **Relational Database (RDS)**:
  - Pro: Familiar SQL query language, ACID compliance
  - Con: Less scalable, higher cost for document metadata
- **Document Database (MongoDB Atlas)**:
  - Pro: Flexible schema, query capabilities
  - Con: External to AWS, additional integration effort

Trade-offs

- NoSQL design requires careful access pattern planning
- Eventual consistency model in DynamoDB
- Multiple data stores to manage

AI Integration Approach

Decision

AI document processing is implemented through direct OpenAI API integration.

Rationale

- **Cutting-edge AI**: Access to state-of-the-art language models
- **Flexibility**: Fine-tuned prompts for different document types
- **Rapid Development**: No need to build or train custom ML models
- **API-driven**: Consistent interface for all AI operations

Alternatives Considered

- **Amazon Textract/Comprehend**:
  - Pro: Native AWS integration, potentially lower cost
  - Con: Less advanced capabilities for complex document understanding
- **Custom ML Models**:
  - Pro: Fully customized for specific document types
  - Con: Significant development effort, ongoing training required

Trade-offs

- External API dependency
- Cost scales with usage
- Rate limiting considerations
- Less customization of model behavior

API Design

Decision

RESTful API with resource-based endpoints and standardized response patterns.

Rationale

- **Developer Familiarity**: Widely understood API design pattern
- **Cacheability**: Response caching for GET operations
- **Statelessness**: Simplifies scaling and fault tolerance
- **Swagger Documentation**: Easy to document and consume

Alternatives Considered

- **GraphQL**:
  - Pro: Flexible queries, reduced over-fetching
  - Con: Learning curve, more complex implementation
- **RPC-style API**:
  - Pro: Direct mapping to function calls
  - Con: Less standardized, harder to discover

Trade-offs

- Less flexible query capabilities compared to GraphQL
- Some operations may require multiple API calls
- More endpoints to maintain

Frontend Framework

Decision

React with TypeScript for the user interface.

Rationale

- **Component-based**: Reusable UI components
- **TypeScript**: Strong typing reduces runtime errors

- **Developer Ecosystem**: Large community, abundant libraries
- **Performance**: Virtual DOM for efficient rendering

Alternatives Considered

- **Angular**:
  - Pro: Comprehensive framework with included capabilities
  - Con: Steeper learning curve, more opinionated
- **Vue.js**:
  - Pro: Gentler learning curve, good performance
  - Con: Smaller ecosystem than React

Trade-offs

- Less structure compared to full frameworks like Angular
- More decisions to make about state management, routing
- Managing TypeScript type definitions

Deployment Automation

Decision

Separate CI/CD pipelines for frontend and backend components.

Rationale

- **Independent Deployment**: Frontend changes don't require backend deployment
- **Specialized Tooling**: Each component has optimized deployment process
- **Reduced Risk**: Issues in one component don't block others
- **Faster Iterations**: Quick UI changes without full-stack deployment

Alternatives Considered

- **Monolithic Deployment**:
    - Pro: Simpler synchronization of versions
    - Con: Slower deployment cycles, higher risk
- **Fully Automated GitOps**:
    - Pro: Declarative configuration, audit trail
    - Con: Complexity in setup and maintenance

Trade-offs

- Managing multiple pipelines
- Potential version mismatch between components
- Additional coordination required