GitPushForce - Technical Architecture Document

Table of Contents

1. Database Architecture

1.1 Core Schema Design

The GitPushForce platform utilizes a **PostgreSQL relational database** designed to support complex group expense sharing scenarios with complete audit trails and flexible expense categorization.

Primary Tables

USER Table

- Purpose: Stores user accounts and authentication credentials
- Key Fields: id (PK), email (UNIQUE, indexed), hashed_password, first_name, last_name, phone_number, budget (nullable), created_at, updated_at
- Relationships: One-to-Many with EXPENSE, CATEGORY, USERGROUP, GROUPLOG
- Constraints: Email format validation (regex), phone numeric validation

GROUP Table

- Purpose: Represents expense sharing groups
- Key Fields: id (PK), name, description (nullable), invitation_code (UNIQUE, indexed), created_at
- Relationships: One-to-Many with EXPENSE, USERGROUP, GROUPLOG
- Features: Unique invitation codes for joining, audit trail of all membership changes

EXPENSE Table

- Purpose: Individual expense records within system
- Key Fields: id (PK), user_id (FK), group_id (FK, nullable), title, amount (> 0), description, category_id (FK), created_at
- Relationships: Many-to-One with USER, GROUP, CATEGORY; One-to-Many with EXPENSEPAYMENT
- Design Feature: group_id nullable supports both personal and shared expenses
- Validation: Amount must be positive, category required

CATEGORY Table

- Purpose: Organize expenses into categories with AI-powered keyword tagging
- Key Fields: id (PK), user_id (FK, nullable), title (max 30 chars), keywords (PostgreSQL array)
- Relationships: One-to-Many with EXPENSE
- Design Feature: User-specific and system-wide categories supported
- Technology: Keywords stored as PostgreSQL arrays for efficient tagging and searching

EXPENSEPAYMENT Table

- Purpose: Tracks payment status for shared expenses
- Key Fields: expense_id (FK, PK), user_id (FK, PK), paid_at (nullable)
- Relationships: Many-to-One with EXPENSE and USER
- Design: Composite primary key prevents duplicate payment records

- Use Case: When expense split among group, tracks who has paid their portion

USERGROUP Table (Join Table)

- Purpose: Implements many-to-many relationship between users and groups
- Key Fields: user_id (FK, PK), group_id (FK, PK)
- Design: Composite primary key ensures no duplicate memberships
- Relationships: Links USER and GROUP tables for group membership

GROUPLOG Table

- Purpose: Complete audit trail of group membership changes
- Key Fields: id (PK), group_id (FK), user_id (FK), action (enum: 'JOIN', 'LEAVE'), created_at
- Relationships: Many-to-One with GROUP and USER
- Purpose: Provides complete activity history for compliance and transparency

1.2 Relationship Architecture

The database implements three relationship types:

One-to-Many (1:M)

- USER creates many EXPENSES, CATEGORIES, logs
- GROUP has many EXPENSES and activity logs
- EXPENSE has many PAYMENTS
- CATEGORY has many EXPENSES

Many-to-Many (M:N)

- USER and GROUP via USERGROUP join table
- Prevents redundancy while supporting flexible group membership

Cascading Deletes

- USER deletion cascades to: EXPENSE, CATEGORY, USERGROUP, GROUPLOG
- GROUP deletion cascades to: EXPENSE, USERGROUP, GROUPLOG
- EXPENSE deletion cascades to: EXPENSEPAYMENT
- CATEGORY deletion cascades to: EXPENSE

1.3 Indexing Strategy

Implemented Indexes

- `idx_user_email` - Authentication and user lookup performance
- `idx_expense_user_id` - User expense filtering queries
- `idx_expense_group_id` - Group expense retrieval
- `idx_expense_created_at` - Sorting and date range queries
- `idx_usergroup_user_id` - Membership queries
- `idx_usergroup_group_id` - Group member listing
- `idx_expensepayment_expense_id` - Payment status lookups

- `idx_grouplog_group_id` - Activity history retrieval
- `idx_group_invitation_code` - Group joining via invitation codes

---

2. Application Architecture Overview

2.1 Layered Architecture Pattern

The GitPushForce platform follows a **4-tier architectural pattern** ensuring separation of concerns and scalability:

Presentation Layer (UI)
↓
API Gateway Layer (HTTP/REST)
↓
Business Logic Layer (Services)
↓
Data Access Layer (Repositories)
↓
Database & External Services

2.2 Core Architecture Principles

Routes (HTTP Entry Point)

- Parse and validate HTTP requests
- Extract JWT authentication tokens
- Route requests to appropriate services
- Return JSON responses with proper status codes
- No business logic at this layer

Services (Business Logic)

- Validate input data against business rules
- Enforce authorization and permissions
- Orchestrate multiple repositories for complex operations
- Handle error scenarios and logging
- Implement domain-specific workflows (e.g., group joining, expense splitting)

Repositories (Data Access)

- Pure CRUD operations against database
- Use SQLAlchemy ORM for type-safe queries
- No business logic or branching
- Handle database connection pooling
- Implement efficient query patterns

Database (Persistence)

- PostgreSQL with SQLAlchemy ORM abstraction
- Automatic query building and parameterization
- Built-in SQL injection prevention

- Transaction management and ACID compliance

2.3 State Management Across Client Applications

Web Frontend (React Context API)

- AuthContext: User session, JWT tokens, login/logout operations
- CurrencyContext: Currency preference (RON/EUR), real-time exchange rates
- ThemeContext: Light/dark mode preference, CSS variable switching

Mobile Applications (Kotlin/iOS)

- TokenDataStore: Secure local storage of JWT tokens via Android DataStore
- In-memory state in ViewModels: Transient application state
- Shared Kotlin Multiplatform logic: Domain models and validation rules

---

3. Backend API Services

3.1 FastAPI Backend Architecture

Framework Selection: FastAPI

- Async-first Python framework with automatic OpenAPI/Swagger documentation
- Type hints integrated with Pydantic for request/response validation
- High performance: Asynchronous request handling, competitive benchmarks
- Development speed: Auto-generated interactive API documentation
- Production ready: Used at scale by major organizations

Key Dependencies

- SQLAlchemy 2.x: ORM for type-safe database operations
- Pydantic: Request/response data validation with JSON Schema
- python-jose: JWT token generation and validation
- passlib + bcrypt: Secure password hashing (bcrypt with salt)
- python-multipart: Form data parsing for file uploads
- httpx: Async HTTP client for external API calls

3.2 Authentication & Authorization

JWT Token Flow

1. User submits credentials to `/auth/login`
2. Backend validates against bcrypt-hashed password in database
3. JWT token generated with user_id and expiration (3 days)
4. Token returned in response AND set in httponly cookie (prevents XSS)
5. Client includes token in Authorization header for authenticated requests
6. Backend validates JWT signature and expiration for every request

Token-based Authorization

- Routes marked with `@auth_required` decorator verify JWT presence
- user_id extracted from token claims (no user submission required)
- Permissions enforced at service layer (e.g., only expense creator can delete)
- Token refresh mechanism available before expiration

Password Security

- Passwords hashed with bcrypt (PBKDF2 equivalent)
- Salt automatically generated per password
- Never stored in plain text
- Validation through `passlib` library
- Password reset mechanism with time-limited reset tokens

3.3 Core API Endpoint Categories

Authentication Endpoints (No JWT required)

- `POST /auth/register` - Create new user account with email/password validation
- `POST /auth/login` - Authenticate and receive JWT token + httponly cookie
- `GET /auth/me` - Return current authenticated user profile
- `POST /auth/logout` - Clear httponly cookie and invalidate session

User Management (All JWT-protected)

- `GET /users/` - List all users in system
- `GET /users/{user_id}` - Retrieve specific user profile
- `PUT /users/{user_id}` - Update user details (name, phone, preferences)
- `DELETE /users/{user_id}` - Deactivate account and cascade deletions
- `GET /users/{user_id}/budget` - Retrieve user's monthly budget limit
- `PUT /users/{user_id}/budget` - Set personal budget constraint
- `GET /users/{user_id}/spent-this-month` - Calculate total spending in current month
- `GET /users/{user_id}/remaining-budget` - Returns budget - spent amount

Group Management (JWT-protected)

- `POST /groups/` - Create group with auto-generated invitation code
- `GET /groups/` - List groups with pagination (offset, limit)
- `GET /groups/{group_id}` - Retrieve group details including member count
- `PUT /groups/{group_id}` - Update group name/description
- `DELETE /groups/{group_id}` - Delete group (cascade to expenses/memberships)
- `POST /groups/{group_id}/users/{user_id}` - Add member to group
- `DELETE /groups/{group_id}/leave` - Current user removes themselves from group
- `DELETE /groups/{group_id}/users/{user_id}` - Admin removes user from group
- `GET /groups/{group_id}/users` - List all members in group

- `GET /groups/{group_id}/expenses` - Fetch group expenses with filtering/sorting
- `GET /groups/{group_id}/invite-qr` - Generate QR code with invitation link
- `GET /groups/{group_id}/statistics/user-summary` - User's spending breakdown in group

Expense Management (JWT-protected)

- `POST /expenses/` - Create expense (user_id from JWT, group_id optional)
- `GET /expenses/all` - List all expenses system-wide with advanced filtering
- `GET /expenses/` - List user's expenses across all groups
- `GET /expenses/{expense_id}` - Retrieve specific expense details
- `GET /expenses/group/{group_id}` - List group's expenses
- `PUT /expenses/{expense_id}` - Update expense (creator-only permission)
- `DELETE /expenses/{expense_id}` - Delete expense (creator-only)

Advanced Filtering & Sorting
Query parameters: offset (pagination start), limit (page size, 1-1000), sort_by (default: created_at), order (asc/desc), min_price, max_price, date_from, date_to, category (filter by category name)

Payment Tracking (JWT-protected)

- `POST /expenses/{id}/pay/{user_id}` - Mark user as paid for shared expense
- `DELETE /expenses/{id}/pay/{user_id}` - Unmark payment status
- `GET /expenses/{id}/payments` - List all payment statuses for expense

Category Management (JWT-protected)

- `POST /categories/` - Create user-specific category with optional keywords
- `GET /categories/` - List all system categories and user's custom categories
- `GET /categories/{user_id}` - Retrieve categories for specific user
- `PUT /categories/{id}` - Update category (creator-only)
- `DELETE /categories/{id}` - Delete category (creator-only, cascades)

Receipt Processing (JWT-protected, multipart)

- `POST /receipts/process-receipt` - Upload receipt image for OCR and AI extraction

Activity Logging (JWT-protected)

- `GET /group-logs/{group_id}` - Retrieve JOIN/LEAVE events for audit trail

3.4 Error Handling Strategy

HTTP Status Code Convention

- `200 OK` - Successful GET/PUT request
- `201 Created` - Successful POST creating resource
- `204 No Content` - Successful DELETE operation
- `400 Bad Request` - Validation failure (invalid input)
- `401 Unauthorized` - Missing or invalid JWT token

- `403 Forbidden` - Valid token but insufficient permissions
- `404 Not Found` - Resource doesn't exist
- `500 Internal Server Error` - Unhandled server exception

Error Response Format

```
{
"detail": "Descriptive error message",
"error_code": "SPECIFIC_ERROR_TYPE"
}
```

Custom Exceptions

- `UserNotFound` - User lookup failed
- `InvalidCredentials` - Password/email mismatch
- `PermissionDenied` - User lacks authorization
- `GroupNotFound` - Group lookup failed
- `ExpenseNotFound` - Expense lookup failed
- `InvalidExpenseAmount` - Amount validation failed
- `CategoryNotFound` - Category lookup failed

---

4. Frontend Applications

4.1 Web Frontend Architecture (React + TypeScript)

Technology Stack

- React 18.2.0: Component-based UI framework with hooks
- TypeScript 4.9.3: Type-safe JavaScript with full IDE support
- Vite 4.5.14: Build tool providing <1ms HMR (Hot Module Replacement)
- Axios 1.13.1: HTTP client with interceptor support for authentication
- Bootstrap 5.3.8: Responsive CSS framework
- Recharts 3.4.1: React charting library for data visualization
- Context API: Global state management without external libraries

Component Organization

Core Components (Authentication & Navigation)

- `LoginForm.tsx` - Email/password input with credential validation
- `RegisterForm.tsx` - New user registration with password confirmation
- `Dashboard.tsx` - Home page with budget summary and shortcuts

Feature Components (Main Functionality)

- `Groups.tsx` - List user's groups with create/join functionality
- `GroupDetail.tsx` - Group overview, members, expenses, invite code
- `Receipts.tsx` - Hub component routing to upload/manual/camera

- `ReceiptsUpload.tsx` - Drag-drop receipt image upload
- `ReceiptsManual.tsx` - Manual expense entry form
- `ReceiptsCamera.tsx` - Device camera integration for receipt capture
- `ReceiptsView.tsx` - Expenses list with filtering and edit/delete actions
- `Categories.tsx` - Create custom expense categories with keyword tagging
- `Data.tsx` - Analytics dashboard with Recharts visualizations
- `Profile.tsx` - User settings (name, email, budget, currency preference)
- `ChatBot.tsx` - AI assistant for expense suggestions
- `ThemeToggle.tsx` - Light/dark mode switcher

Global State Management via Context API

AuthContext

- Properties: user (current user object), isAuthenticated (boolean), token (JWT)
- Methods: login(email, password), register(email, password, name), logout()
- Side Effects: Stores token in localStorage, validates on app load

CurrencyContext

- Properties: currency (RON/EUR), exchangeRate (number), formatted amounts
- Methods: setCurrency(currency), getFormattedAmount(amount)
- Behavior: Caches exchange rates locally, updates daily

ThemeContext

- Properties: isDarkMode (boolean), theme (CSS variable set)
- Methods: toggleTheme(), setTheme(theme)
- Behavior: Persists preference in localStorage, applies CSS variables

Service Layer Architecture

`api-client.ts` - Axios Configuration

- baseURL: `http://localhost:8000/api` (or production URL)
- timeout: 30 seconds
- withCredentials: true (includes cookies in requests)
- Interceptors: Adds Authorization header with JWT token

`auth-service.ts` - Authentication

- `login(email, password)` - POST to `/auth/login`
- `register(email, password, name)` - POST to `/auth/register`
- `logout()` - POST to `/auth/logout`
- `getCurrentUser()` - GET `/auth/me`
- `refreshToken()` - Handles token refresh

`group-service.ts` - Group Operations

- `createGroup(name, description)` - POST `/groups/`
- `getGroups()` - GET `/groups/`

- `getGroupDetails(groupId)` - GET `/groups/{id}`
- `joinGroup(invitationCode)` - POST with invitation code
- `leaveGroup(groupId)` - DELETE `/groups/{id}/leave`
- `getGroupMembers(groupId)` - GET `/groups/{id}/users`

`receipt-service.ts` - Receipt Processing

- `uploadReceipt(file)` - POST `/receipts/process-receipt` (multipart)
- `getReceiptHistory()` - GET processed receipts
- Handles image preview and OCR result display

`expense-service.ts` - Expense Management

- `createExpense(expense)` - POST `/expenses/`
- `getExpenses(filters)` - GET `/expenses/` with query parameters
- `updateExpense(id, updates)` - PUT `/expenses/{id}`
- `deleteExpense(id)` - DELETE `/expenses/{id}`
- `getExpensesByGroup(groupId)` - GET `/expenses/group/{id}`

`category-service.ts` - Category Management

- `createCategory(title, keywords)` - POST `/categories/`
- `getCategories()` - GET `/categories/`
- `updateCategory(id, updates)` - PUT `/categories/{id}`
- `deleteCategory(id)` - DELETE `/categories/{id}`

`exchange-rate.ts` - Currency Conversion

- `getExchangeRate(from, to)` - Fetches real-time RON/EUR rate
- Implements local caching with expiration
- Used by CurrencyContext for formatting

Responsive Design Strategy

- Bootstrap 5.3.8 grid system (12-column)
- Mobile-first CSS approach
- Media queries for tablets/desktops
- Flexbox layouts for component alignment
- Touch-friendly button sizes (min 48px)

Styling & Theming

- CSS Variables in `App.css` for theme switching
- Light mode: Light backgrounds, dark text
- Dark mode: Dark backgrounds, light text
- Runtime theme switching (no page refresh)

---

5. Mobile Development Framework

5.1 Kotlin Multiplatform Mobile (KMM) Architecture

Strategic Choice: Code Sharing
The mobile app uses Kotlin Multiplatform to share business logic (models, repositories, API clients) while maintaining completely native UIs on both Android and iOS. This approach provides:

- Single source of truth for business rules
- Native UI performance and UX conventions
- Faster feature development (50% code reuse)
- Platform-specific capabilities when needed

Project Structure

```
Mobile/
├── androidApp/ # Android-specific code
├── iosApp/ # iOS-specific code
└── shared/ # Kotlin Multiplatform shared code
├── data/ # Models, API clients
├── domain/ # Business logic
└── platform.kt # Platform abstraction
```

5.2 Android Application

Minimum SDK: API 21 (Android 5.0)
Target SDK: API 34 (Android 14)

Technology Stack

- Jetpack Compose: Modern declarative UI framework
- XML Layouts: Legacy layout support for specific screens
- MVVM Architecture: Clean separation between UI and logic
- Retrofit + OkHttp: HTTP client with interceptors
- Android DataStore: Secure key-value storage for tokens

Authentication & Token Management

TokenDataStore

- Stores JWT token in encrypted SharedPreferences (DataStore wrapper)
- Auto-retrieves token before app start
- Automatically invalidates on expiration
- Secure against app backup extraction

TokenAuthInterceptor

- OkHttp interceptor adding JWT to Authorization header
- Triggers login flow if token invalid
- Handles token refresh automatically
- Prevents token exposure in app logs

MVVM ViewModel Pattern

ViewModels maintain screen state and handle business logic:

- `ExpenseViewModel` - Expense list state, filtering, sorting, pagination
- `GroupsViewModel` - Group membership, creation, joining, leaving
- `AnalyticsViewModel` - Statistics aggregation, chart data preparation
- `ProfileViewModel` - User settings, preferences, budget updates
- `AuthViewModel` - Login/register state, session management

Each ViewModel includes:

- LiveData/StateFlow for reactive state management
- Error handling with custom error classes
- Loading state indicators (isLoading, isError, errorMessage)
- Repository dependency injection via Factory pattern

UI Components & Screens

Authentication Flow

- `SplashActivity` - Initial load, checks token validity
- `LoginActivity` - Email/password input with validation
- `SignUpActivity` - Registration with password confirmation

Main Application Screens

- `MainActivity` - Bottom navigation hub for main features
- `ExpensesScreen` - List with filters (category, date range, amount)
- `GroupsScreen` - User's groups with create/join options
- `GroupDetailsScreen` - Members, expenses, invite options
- `AnalyticsScreen` - Charts (pie, line, bar) for spending breakdown
- `ReceiptScreen` - Camera/upload integration for receipt capture
- `ProfileScreen` - Account settings, preferences
- `QrCaptureActivity` - QR scanner for group joining or sharing

Data Models (Network Layer)

Request DTOs

- `LoginRequest` - email, password
- `RegisterRequest` - email, password, name
- `ExpenseCreate` - amount, description, category, group_id (optional)
- `GroupCreate` - name, description (optional)
- `CategoryCreate` - title, keywords (array)

Response DTOs

- `LoginResponse` - user object, token
- `UserData` - id, email, name, budget, created_at
- `Expense` - id, user_id, amount, description, category, created_at
- `Group` - id, name, invitation_code, created_at
- `GroupStatistics` - total_expenses, member_count, user_balance

- `Category` - id, title, keywords, user_id

5.3 iOS Application

Minimum iOS Version: 14.0

Technology Stack

- SwiftUI: Apple's modern declarative UI framework
- URLSession: Native HTTP client
- Combine: Reactive programming framework

Current Development Status
The iOS app is built on the KMM-generated structure:

- Native SwiftUI views for all screens
- Integration with shared Kotlin logic for business rules
- Platform-specific features (FaceID, contacts, camera)
- Ready for full feature expansion with parity to Android

Architecture Approach

- Observables: SwiftUI @ObservedObject and @StateObject
- Environment Objects: Global state (similar to React Context)
- Data Models: Codable Swift structs mirrored from Kotlin
- Services: Native wrapper around shared KMM logic

---

6. AI-Powered Receipt Processing

6.1 Architecture & Integration

Serverless Approach
Receipt processing is implemented as an AWS Lambda function triggered via API Gateway, eliminating infrastructure management and providing automatic scaling.

End-to-End Flow

1. User uploads receipt image from mobile/web app
2. Client sends multipart POST to `/receipts/process-receipt`
3. Request routed through API Gateway to Lambda function
4. Lambda invokes AWS Textract service for text extraction
5. ML models analyze extracted text for structure (merchant, date, items, total)
6. Response returns JSON with extracted fields
7. Client parses response and populates expense form
8. User confirms/edits and creates expense record

Extracted Data Fields

Merchant Information

- Vendor/store name
- Location (if visible)
- Contact information

Transaction Details

- Date and time
- Total amount
- Tax amount
- Subtotal

Line Items

- Item descriptions
- Individual prices
- Quantities

Payment Details

- Payment method (cash, card)
- Card last 4 digits (if applicable)
- Change amount

Processing Capabilities

- Handles multiple receipt formats (thermal paper, printed, digital)
- Multi-language support (English, Romanian, others)
- Confidence scoring for each extracted field
- Fallback values for low-confidence extractions

Error Handling

- Invalid image formats: Returns 400 with error message
- No text detected: Returns 400 with OCR failure message
- Timeout (processing > 30s): Returns 408 with retry suggestion
- Service unavailable: Returns 503 with fallback to manual entry

Integration with Expense Creation
Extracted data pre-populates the expense creation form:

- Amount field auto-filled from total
- Date set from receipt date
- Description contains merchant name
- Category suggested based on keywords
- User can edit before final submission

Cost Optimization

- Lambda pricing: $0.20 per million requests + $0.0000166667 per GB-second
- Typical receipt processing: <1 second, <128MB memory
- Estimated cost: ~$0.000004 per receipt
- CloudWatch logging for monitoring and troubleshooting

7. Cloud Infrastructure & Deployment

7.1 AWS Architecture Overview

Deployment Strategy
GitPushForce runs on Amazon AWS using a **managed services approach** prioritizing reliability, security, and operational simplicity over infrastructure management.

Service Topology

Entry Point

- Route53: DNS resolution and domain management
- CloudFront: Global CDN for static assets and API caching
- WAF: Layer 7 protection against web attacks
- Shield: Layer 3/4 DDoS protection

Application Tier

- Application Load Balancer: Distributes traffic to compute
- EC2 Auto Scaling Group: FastAPI backend instances
- Running on t3.medium or t3.large instances
- Auto-scales based on CPU metrics

Data Tier

- RDS PostgreSQL: Managed relational database
- Multi-AZ replication across availability zones
- Automated backups (35-day retention)
- Read replicas for scaling read operations

Serverless Processing

- API Gateway: HTTP endpoint for Lambda
- AWS Lambda: Serverless receipt processing
- CloudWatch: Logs and metrics from all services

Secrets & Keys

- Parameter Store: Encrypted storage for database credentials, API keys
- KMS: Master key management with automatic rotation
- IAM Roles: Service-to-service authentication

7.2 Core AWS Services

Route53 (DNS & Domain Registrar)

- Registers and manages the domain name
- Hosts DNS records in managed zone
- Health check routing for failover scenarios

- Traffic policies for advanced routing (latency-based, weighted)

CloudFront (Content Delivery Network)

- Distributes static web app files from S3
- Caches at 400+ edge locations globally
- Reduces latency for international users
- Origin Access Control (OAC) restricts S3 access

S3 Integration

- Stores compiled React app (dist/ folder)
- Static assets: CSS, JavaScript, images, fonts
- Versioning enabled for rollback capability
- Lifecycle policies for old version cleanup

Application Load Balancer (ALB)

- Single stable DNS endpoint for backend API
- Health checks monitor EC2 instances (port 8000, /health)
- Distributes traffic across ASG instances
- Session stickiness (optional, for WebSocket support)
- SSL/TLS termination (HTTPS offloading)

EC2 Auto Scaling Group (ASG)

- Maintains desired number of FastAPI instances
- Scales horizontally based on CloudWatch metrics
- Launch template specifies: instance type, AMI, security groups, IAM role
- Availability: Spans 2-3 availability zones for fault tolerance

Scaling Configuration

- Minimum instances: 2 (high availability)
- Desired: 3-5 (adjustable)
- Maximum: 10 (cost limit)
- Scale-out trigger: Average CPU > 70% for 2 minutes
- Scale-in trigger: Average CPU < 30% for 10 minutes
- Cool-down period: 5 minutes (prevents rapid oscillation)

RDS PostgreSQL

- Managed database eliminating administrative overhead
- Automated minor version patching (during maintenance window)
- Multi-AZ replication: Synchronous standby in different AZ
- Automatic failover: <1 minute RTO (Recovery Time Objective)
- Backup strategy:
    - Daily automated snapshots
    - 35-day retention period
    - Point-in-time recovery to any moment in 35 days

    - Cross-region backup replication (optional)

Database Specifications

- Instance type: db.t3.small to db.t3.large (adjustable)
- Storage: 100-1000 GB (auto-scaling enabled)
- Connection pooling: pgBouncer for efficient connection management
- Monitoring: Enhanced monitoring with OS metrics

Lambda + API Gateway (Serverless Receipt Processing)

- API Gateway: HTTP endpoint at `/receipts/process-receipt`
- Routes to Lambda function automatically
- Lambda function: Orchestrates Textract, processes results
- Execution role: IAM permissions for Textract and CloudWatch

Lambda Configuration

- Runtime: Python 3.11 (async-compatible)
- Memory: 512 MB (adjustable, 128-10,240 MB range)
- Timeout: 30 seconds (sufficient for receipt processing)
- Concurrency: 100 (burst capacity)

Billing Model

- Free tier: 1 million requests/month
- After: $0.20 per million requests
- Plus: $0.0000166667 per GB-second of execution
- Cost per receipt: ~$0.000003-0.000005

Parameter Store (Secrets Management)

- Stores encrypted database credentials
- API keys for external services
- Environment variables for application configuration
- Encryption: KMS key (automatic rotation)

Access Pattern

- EC2 instances assume IAM role
- Role grants permission to read specific parameters
- Parameters injected as environment variables
- Credentials never appear in code

KMS (Key Management Service)

- Master encryption key for Parameter Store
- Master key for RDS database encryption
- Automatic annual key rotation
- CloudTrail audit log of all key usage
- Fine-grained access control via IAM policies

CloudWatch (Monitoring & Logging)

Metrics Collection

- EC2 CPU utilization, memory, network
- ALB request count, response time, error rates
- RDS database connections, query latency, storage
- Lambda invocation count, duration, error rate

Dashboards

- Real-time view of system health
- Key metrics: CPU, request rate, database connections
- Error rate trending and alerts

Log Aggregation

- FastAPI application logs to CloudWatch
- Lambda function logs captured automatically
- Log retention: 30 days (configurable)
- Log insights: Query logs for debugging

Alarms & Notifications

- High CPU alarm: Triggers manual investigation
- High error rate alarm: Pages on-call engineer
- Database connection limit: Warns of potential bottleneck
- Lambda errors: Email notification to team

SNS (Simple Notification Service)

- Publishes notifications to multiple channels
- Email for critical alerts
- Slack integration for engineering team
- Subscriber: On-call rotation

Events Triggering Notifications

- ASG scales out (new instance launched)
- ASG scales in (instance terminating)
- CloudWatch alarm triggered (critical metric)
- Lambda error rate exceeds threshold

WAF (Web Application Firewall)

- Protects ALB from Layer 7 attacks
- Rules prevent: XSS, SQL injection, CSRF, bot traffic
- IP reputation filtering: Blocks known malicious sources
- Rate limiting: Prevents brute force attacks
- Managed rules from AWS and third parties

Shield (DDoS Protection)

- Automatic protection against Layer 3/4 attacks
- Detects volumetric attacks (UDP floods)
- Mitigates without application involvement
- Shield Standard: Included free with AWS
- Shield Advanced: Optional enhanced protection

Certificate Manager (ACM)

- Manages SSL/TLS certificates
- HTTPS encryption for CloudFront distribution
- HTTPS termination at ALB
- Auto-renewal: 30 days before expiration
- No additional cost

GuardDuty (Threat Detection - Currently Disabled)

- Machine learning-based security monitoring
- Analyzes VPC flow logs and DNS queries
- Reviews CloudTrail events for suspicious activity
- Identifies compromised credentials, unauthorized access
- Can be enabled for enhanced security posture

---

8. Security & Infrastructure Management

8.1 Security Architecture

Network Security

- VPC (Virtual Private Cloud): Isolated network environment
- Public Subnets: ALB and NAT gateway (internet-accessible)
- Private Subnets: RDS database (internet-blocked)
- Security Groups: Stateful firewall rules
    - ALB: Allows port 443 (HTTPS) and 80 (HTTP→HTTPS redirect)
    - EC2: Allows port 8000 (from ALB only)
    - RDS: Allows port 5432 (from EC2 only)
- Network ACLs: Stateless filtering (secondary layer)

Data Security

- Encryption at rest: RDS encryption with KMS master key
- Encryption in transit: HTTPS everywhere (TLS 1.3)
- Database credentials: Never in code, only in Parameter Store
- API keys: Encrypted in Parameter Store, rotated regularly

Access Control

- IAM roles: Service-to-service authentication (EC2→RDS, Lambda→Textract)
- Least privilege: Each service has minimal required permissions
- Root account: MFA protected, not used for operations
- API authentication: JWT tokens with 3-day expiration

Input Validation

- Pydantic: Backend request validation with type checking
- React TypeScript: Frontend type safety
- SQL parameterization: Automatic via SQLAlchemy (SQL injection prevention)

8.2 Scalability Architecture

Horizontal Scaling

- EC2 ASG: Automatically scales backend instances
- RDS read replicas: Scale read-heavy queries
- Lambda: Automatically parallelizes invocations

Vertical Scaling

- Upgrade instance types without code changes
- RDS: Increase allocated storage incrementally
- Memory/CPU adjustable for Lambda

Caching Strategy

- CloudFront: Static asset caching (24-hour default TTL)
- RDS Query cache: Frequently accessed data
- Client-side: React component memoization

Database Optimization

- Indexes: Strategic on frequently-queried columns
- Connection pooling: Efficient connection reuse
- Query optimization: Minimize N+1 queries

8.3 High Availability & Disaster Recovery

Redundancy

- Multi-AZ RDS: Automatic failover to standby (< 5 minutes)
- Multi-instance ASG: Service continues if one instance fails
- CloudFront edge locations: Global distribution prevents single point of failure

Backup Strategy

- Daily automated RDS snapshots (35-day retention)
- Cross-region replication: Data survives region failure
- Point-in-time recovery: Restore to any moment in 35 days
- AWS Backup: Centralized backup management

Recovery Procedures

- RTO (Recovery Time Objective): < 5 minutes (RDS failover)
- RPO (Recovery Point Objective): < 1 minute (continuous replication)
- Incident response: Automated alarms → SNS notification → team response

---

9. Project Outcomes & Technical Achievements

9.1 Architectural Accomplishments

Multi-Platform Development

- Single REST API serving web, Android, and iOS clients
- Kotlin Multiplatform sharing 50% of mobile code
- Consistent business logic across all platforms
- Platform-specific UI optimizations maintained

Database Design Excellence

- Normalized schema with 7 interconnected tables
- Composite primary keys for complex relationships
- Comprehensive audit trail (GROUPLOG table)
- Flexible design supporting personal and shared expenses

Full-Stack Technology Integration

- Modern async backend (FastAPI) with 70% test coverage
- Type-safe frontend (React + TypeScript)
- Native mobile experiences (Kotlin/SwiftUI)
- Serverless AI integration (Lambda + Textract)

Production-Ready Infrastructure

- AWS managed services eliminating operational burden
- High availability through multi-AZ deployment
- Automatic scaling responding to demand
- Comprehensive monitoring and alerting

9.2 Technical Quality Metrics

Code Quality

- Linting: Ruff enforces 15+ code style rules
- Testing: 70% minimum coverage requirement
- Documentation: Swagger auto-documentation via FastAPI
- Type safety: TypeScript (frontend), Python hints (backend), Kotlin (mobile)

Performance Targets

- API response time: < 200ms for 95th percentile

- Database query time: < 100ms for typical queries
- Frontend page load: < 3 seconds
- Receipt processing: < 30 seconds per image

Security Posture

- HTTPS encryption: All data in transit
- Database encryption: All data at rest
- JWT authentication: Every API request validated
- Input validation: All user input sanitized
- Audit logging: Complete activity trail

9.3 DevOps Excellence

CI/CD Pipeline

- 4-stage automated validation (lint → test → coverage → integration)
- Every commit triggers full test suite
- Main branch protection requires passing checks
- Automatic deployment on merge

Infrastructure as Code

- CloudFormation/Terraform (Git-tracked)
- Reproducible deployments
- Version-controlled configuration
- Disaster recovery testing

Monitoring & Observability

- CloudWatch dashboards: Real-time system health
- Alerting: Critical issues notify team immediately
- Logging: Centralized log aggregation
- Metrics: Performance tracking and trending

9.4 Team Collaboration

Development Workflow

- 10 developers working in parallel
- Feature branches with code review requirement
- Automated testing gates prevent regressions
- Clear separation of concerns enables parallel work

Communication Infrastructure

- Jira: Task tracking and discussion
- Pull Requests: Code review and knowledge sharing
- Discord: Real-time team communication
- Weekly standups: Progress and blocker discussion

Documentation

- Comprehensive README files for each component
- API documentation via Swagger/OpenAPI
- Database schema documentation with diagrams
- Setup guides for local development

Conclusion

GitPushForce demonstrates enterprise-grade software engineering across backend development, frontend design, mobile platforms, cloud infrastructure, and DevOps automation. The project successfully implements a complex financial application with group expense sharing, AI-powered receipt processing, and production-ready deployment on AWS.

The architecture emphasizes scalability, security, and maintainability, with comprehensive testing, monitoring, and documentation supporting a team of 10 developers delivering production-quality software in an academic setting.