CSC 431

ScentMatch

System Architecture Specification (SAS)

Team Members and Roles

Member Name	Role
Zoey Lee	Front End Engineer
Cole Foster	Machine Learning Engineer
Anthony Givans	Machine Learning Engineer

Version History

Version	Date	Author(s)	Change Comments
1.0	April 14, 2025	Team 1	Initial version

Table of Contents

Table of Tables

Table of Figures

System Analysis

System Overview

ScentMatch is a personalized fragrance recommendation system using machine learning to match users with scents based on preferences and feedback. The system transforms user preferences into vector embeddings using a BERT-based model for efficient similarity matching with scent products. Core functionality includes:

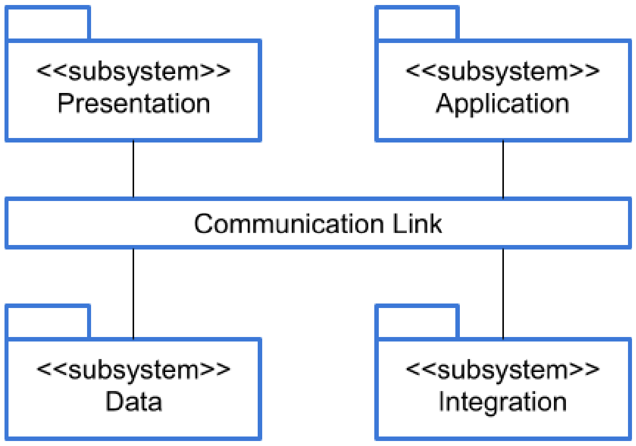
1. User registration and preference collection
2. Transformation of preferences into vector embeddings
3. Similarity-based scent recommendation using vector comparisons
4. Order processing and fulfillment through Amazon API
5. Feedback collection and preference profile refinement

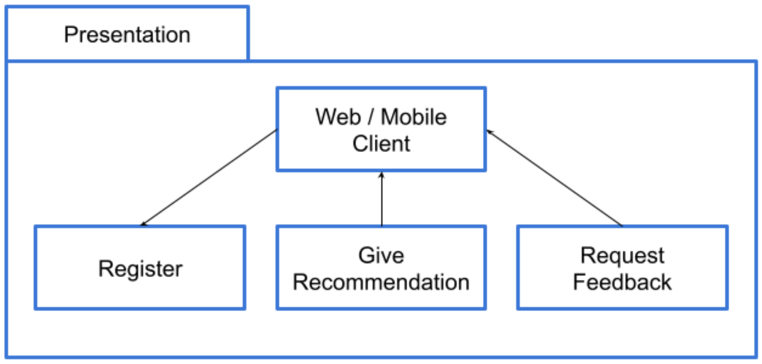
The system operates as a subscription service providing monthly fragrance recommendations with continuous preference refinement. Users receive personalized recommendations every 30 days, review products, and have their preference profiles iteratively refined based on feedback, allowing for increasingly accurate recommendations over time.

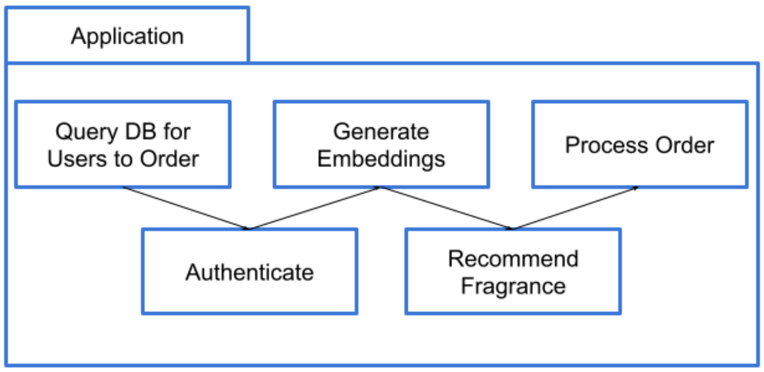
Key architectural priorities include:

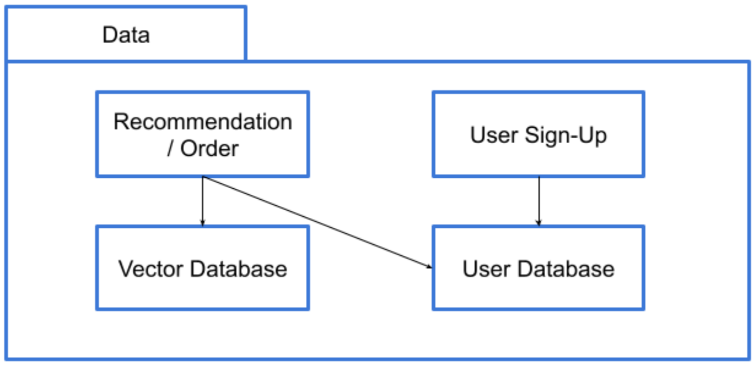
- Real-time recommendation performance (response times ≤ 200ms)
- Scalability to handle 1000 queries per second under peak load
- Security for user data (256-bit encryption, GDPR & CCPA compliance)
- High system reliability (99.9% uptime, MTBF ≥ 10,000 hours)
- Data integrity with < 0.01% error rate

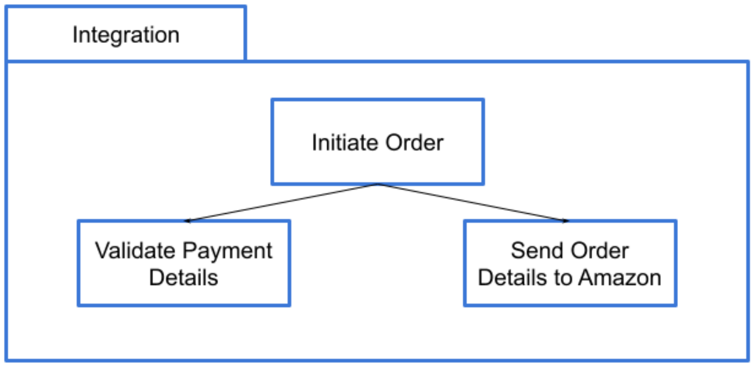
System Diagram


<<subsystem>> Presentation
<<subsystem>> Application
Communication Link
<<subsystem>> Data
<<subsystem>> Integration









System Components

Component	Tier	Description	Technology
Web/Mobile Client	Presentation	User interface for registration, recommendations, feedback	Next.js, React Native
Authentication Service	Application	User registration, login, session management	JWT, OAuth 2.0, FastAPI

Recommendation Engine	Application	Processes user embeddings, generates recommendations	PyTorch, Transformers, FastAPI
Embedding Service	Application	Transforms preferences into vector embeddings	BERT, PyTorch, ONNX
Order Processing Service	Application	Handles orders, payments, fulfillment	Python, FastAPI
Vector Database	Data	Stores user and scent embeddings	PostgreSQL, pgvector
User Database	Data	Stores profiles, preferences, authentication	PostgreSQL
Amazon API Integration	Integration	Interfaces with Amazon for fulfillment	REST API, AWS SDK
Payment Gateway	Integration	Processes payments	Stripe SDK, PCI-DSS

Actor Identification

1. **End Users**: Primary actors who register accounts, set preferences, receive recommendations, place orders, and provide feedback
2. **System Administrators**: Technical personnel responsible for system operation, monitoring, and maintenance
3. **Order Fulfillment Systems (Amazon API)**: External system that processes orders and handles shipping
4. **Payment Processors**: External services that handle financial transactions
5. **Database Systems**: Backend systems for data storage and retrieval
6. **Machine Learning Infrastructure**: Systems supporting the embedding model and recommendation engine

Design Rationale

Architectural Style

ScentMatch uses a **Multi-tier Microservices Architecture** with vector-based recommendation capabilities:

- **Separation of Concerns**: 3-tier architecture (presentation, application, data) isolates different aspects for specialized development and testing. This provides clear boundaries between user interface, business logic, and data storage.

- **Scalability**: Independent scaling of each tier to handle 1000 queries/second during peak loads while maintaining response times under 200ms. The microservices approach allows for targeted resource allocation to specific components experiencing higher demand.

- **Security**: Clear boundaries between layers protect user data and payment information, with particular isolation of the data tier from direct client access, supporting secure handling of personal and payment data.

- **Maintainability**: Changes to one tier have minimal impact on others, allowing for faster feature delivery and simplified updates.

- **Fault Isolation**: Failures in one microservice don't necessarily cascade to others, improving overall system reliability and supporting the high uptime requirements.

Alternatives considered but rejected include monolithic architecture (limited scalability), serverless architecture (challenges for ML components requiring consistent resources), and event-driven architecture (added complexity for real-time recommendation flows).

Design Patterns

1. **Repository Pattern**
    - **Application**: Used for data access abstraction across all data stores
    - **Implementation**: All data access encapsulated in repository classes with standardized operations
    - **Benefits**: Consistent interface for different data sources, simplified testing, centralized data access logic
    - **Example**: UserRepository provides methods like getUserById() and updateUserPreferences() that abstract the PostgreSQL implementation
2. **Strategy Pattern**
    - **Application**: Employed in the recommendation engine for different algorithms
    - **Implementation**: Family of interchangeable recommendation algorithms
    - **Benefits**: Dynamic selection of strategies based on context, facilitates A/B testing
    - **Example**: RecommendationStrategy interface with implementations like CosineSimilarityStrategy and HybridRecommendationStrategy
3. **Observer Pattern**
    - **Application**: Implemented for subscription management and notifications
    - **Implementation**: One-to-many dependency for state change notifications
    - **Benefits**: Loosely coupled event notification, simplifies subscription cycle workflow

   - **Example**: SubscriptionManager notifies OrderProcessingService and NotificationService when cycle events occur
4. **Factory Pattern**
   - **Application**: Applied to embedding generation for complex object creation
   - **Implementation**: Interface for creating embeddings with subclass implementation decisions
   - **Benefits**: Centralizes creation logic, abstracts complexity, supports different embedding types
   - **Example**: EmbeddingFactory creates appropriate generators like UserPreferenceEmbedder and ProductEmbedder
5. **Adapter Pattern**
   - **Application**: Used for integration with external services
   - **Implementation**: Converts interfaces to expected formats
   - **Benefits**: Consistent interface to varied external APIs, isolates dependencies
   - **Example**: AmazonOrderAdapter converts internal orders to Amazon API format
6. **Circuit Breaker Pattern**
   - **Application**: Implemented for resilient external communication
   - **Implementation**: Monitors failures and prevents operation when systems are faulty
   - **Benefits**: Prevents cascading failures, enables graceful degradation
   - **Example**: ExternalServiceClient implements circuit breaking for all API calls

Framework

1. **Frontend Framework: Next.js with React**
   - **Purpose**: Build responsive web interface for user interaction
   - **Key Features**:
     - Server-side rendering for improved initial page load
     - Static site generation for high-performance content
     - API routes for backend functionality
     - Built-in routing and code splitting
     - TypeScript support for type safety
   - **Rationale**: Provides optimal balance of performance and developer experience for cross-device support
   - **Version**: Next.js 13.0+ with React 18.0+
2. **Backend Framework: FastAPI**
   - **Purpose**: Develop high-performance API endpoints
   - **Key Features**:
     - Asynchronous request handling for improved throughput
     - Automatic OpenAPI documentation

        - Built-in dependency injection
        - Type validation using Pydantic
        - High performance compared to other Python frameworks
    - **Rationale**: Offers the speed needed for real-time recommendation performance requirements
    - **Version**: FastAPI 0.100.0+ with Python 3.10+
3. **ML Framework: PyTorch with Hugging Face Transformers**
    - **Purpose**: Implement BERT-based embedding model
    - **Key Features**:
        - Dynamic computational graph for flexible models
        - GPU acceleration for improved performance
        - Extensive pre-trained models through Hugging Face
        - ONNX export for optimization
        - Rich ecosystem of tools and libraries
    - **Rationale**: Most straightforward path to implementing BERT-based embedding approach
    - **Version**: PyTorch 2.0+ with Transformers 4.30.0+
4. **Database: PostgreSQL with pgvector Extension**
    - **Purpose**: Store application data and vector embeddings
    - **Key Features**:
        - Reliable relational database with ACID compliance
        - pgvector extension for vector operations and indexing
        - Support for approximate nearest neighbor search
        - Robust security features and access controls
        - Excellent performance and scalability
    - **Rationale**: Optimal balance of reliability and vector search capabilities
    - **Version**: PostgreSQL 15+ with pgvector 0.5.0+
5. **Task Queue: Celery with Redis**
    - **Purpose**: Manage asynchronous and scheduled tasks
    - **Key Features**:
        - Distributed task queue for asynchronous processing
        - Scheduled task support for recurring operations
        - Worker management and monitoring
        - Retry mechanisms for failed tasks
    - **Rationale**: Reliable processing for subscription cycle features
    - **Version**: Celery 5.3+ with Redis 7.0+
6. **Monitoring: Prometheus and Grafana**
    - **Purpose**: Monitor system health and metrics
    - **Key Features**:
        - Time-series data collection and storage

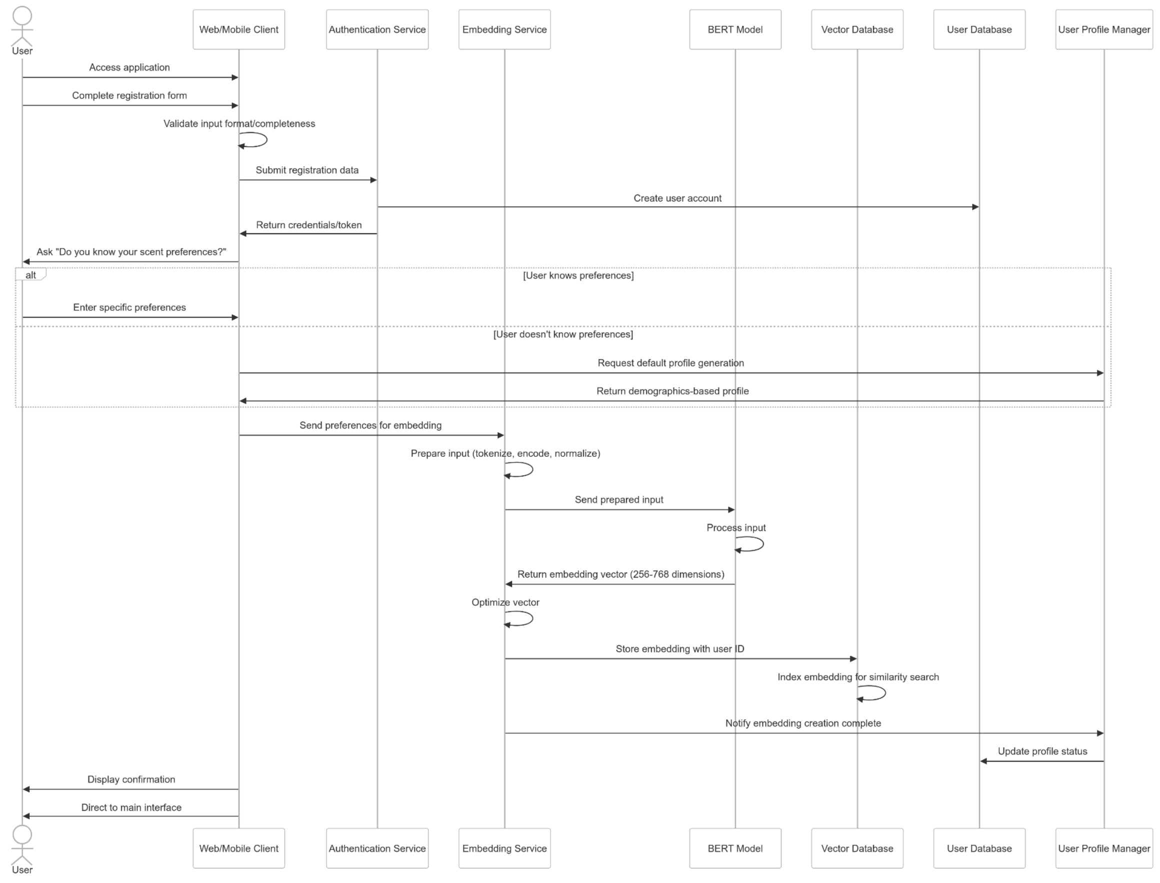
        - Alerting and notification capabilities
        - Comprehensive dashboard creation
        - Extensive integration options
    - **Rationale**: Needed to ensure system meets reliability requirements
    - **Version**: Prometheus 2.45+ with Grafana 10.0+

7. **Deployment: Kubernetes**

    - **Purpose**: Manage containerized services
    - **Key Features**:
        - Container orchestration and management
        - Automatic scaling and load balancing
        - Self-healing capabilities
        - Service discovery and configuration
    - **Rationale**: Provides orchestration for scaling and reliability requirements
    - **Version**: Kubernetes 1.27+

Functional Design

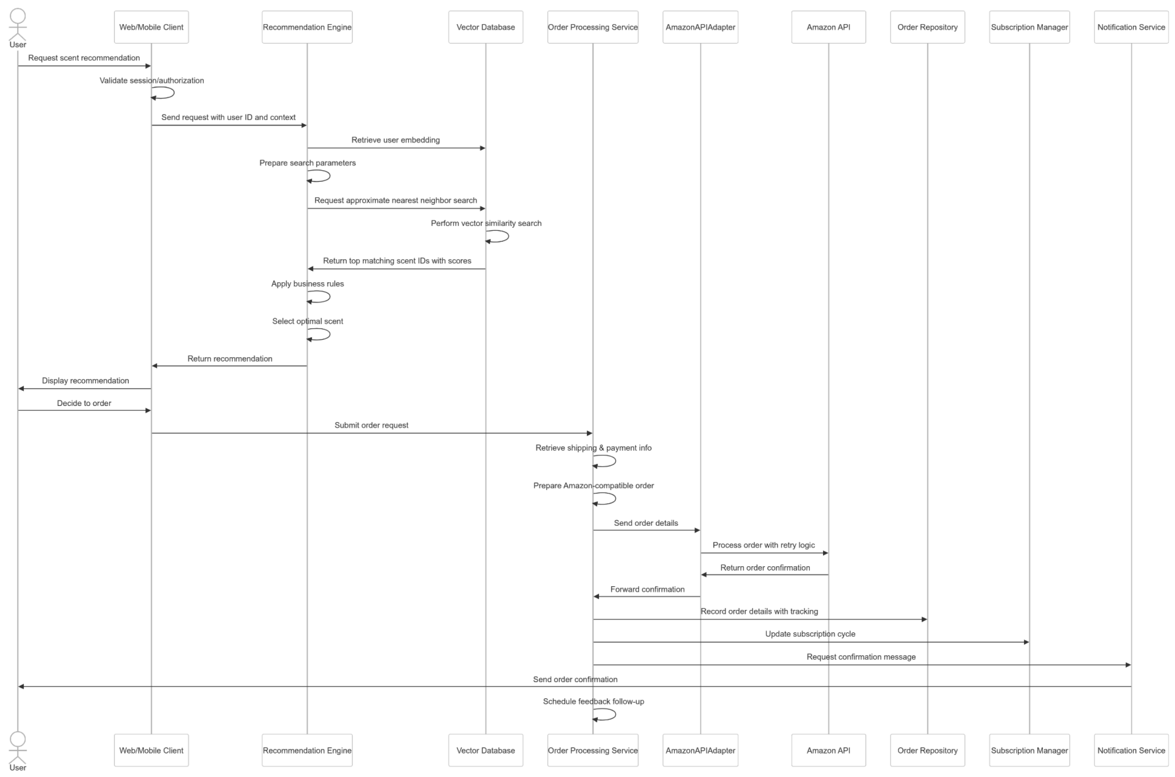
User Registration Workflow


User
Web/Mobile Client
Authentication Service
Embedding Service
BERT Model
Vector Database
User Database
User Profile Manager
Access application
Complete registration form
Validate input format/completeness
Submit registration data
Create user account
Return credentials/token
Ask "Do you know your scent preferences?"
alt
[User knows preferences]
Enter specific preferences
[User doesn't know preferences]
Request default profile generation
Return demographics-based profile
Send preferences for embedding
Prepare input (tokenize, encode, normalize)
Send prepared input
Process input
Return embedding vector (256-768 dimensions)
Optimize vector
Store embedding with user ID
Index embedding for similarity search
Notify embedding creation complete
Update profile status
Display confirmation
Direct to main interface

Key Steps:

1. **Registration Initiation**:
    - User accesses ScentMatch application and completes registration form
    - Frontend validates input for format and completeness
    - Authentication Service creates user account and generates credentials
2. **Preference Collection**:
    - System asks "Do you know your scent preferences?"
    - If yes: User enters specific preferences (fragrance families, notes, intensity)
    - If no: System assigns default profile based on demographics
3. **Embedding Generation**:
    - Embedding Service prepares input for BERT model:
        - Textual preferences are tokenized
        - Categorical preferences are encoded
        - Demographic information is normalized
    - BERT model processes input and generates embedding vector (256-768 dimensions)
    - Embedding Service optimizes the vector
4. **Data Storage**:
    - Vector Database stores embedding with user ID reference
    - Vector Database indexes embedding for similarity search
    - User Profile Manager updates profile status
5. **Completion**:
    - Frontend displays confirmation and directs user to main interface
    - User is now ready to receive personalized recommendations

Recommendation Workflow

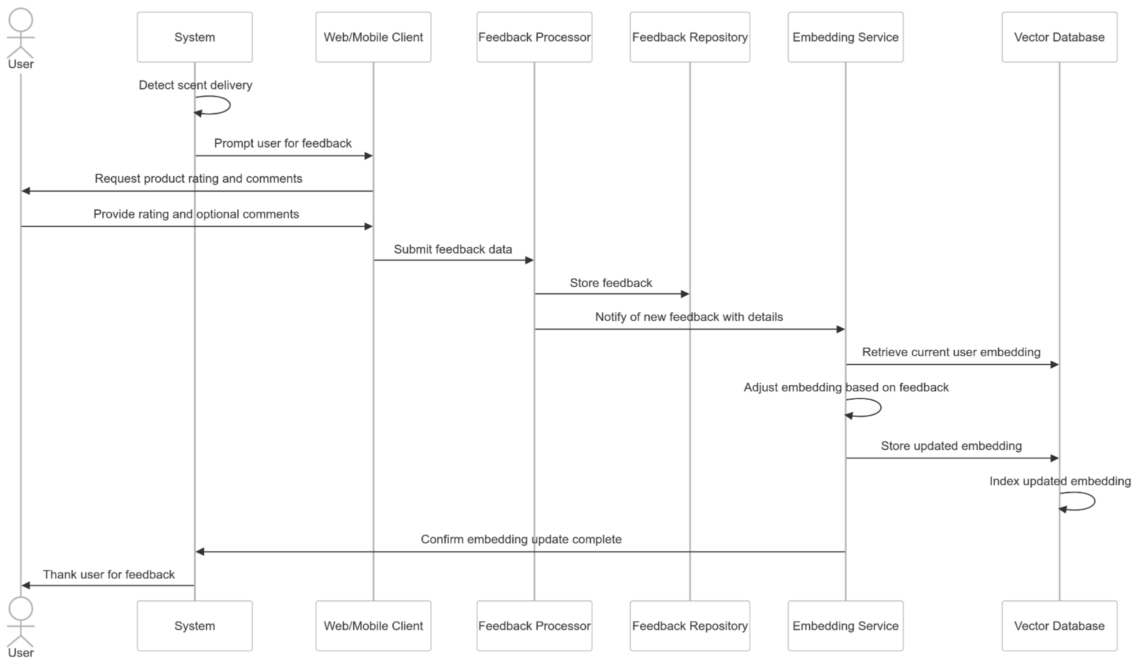

User
Web/Mobile Client
Recommendation Engine
Vector Database
Order Processing Service
AmazonAPIAdapter
Amazon API
Order Repository
Subscription Manager
Notification Service
Request scent recommendation
Validate session/authorization
Send request with user ID and context
Retrieve user embedding
Prepare search parameters
Request approximate nearest neighbor search
Perform vector similarity search
Return top matching scent IDs with scores
Apply business rules
Select optimal scent
Return recommendation
Display recommendation
Decide to order
Submit order request
Retrieve shipping & payment info
Prepare Amazon-compatible order
Send order details
Process order with retry logic
Return order confirmation
Forward confirmation
Record order details with tracking
Update subscription cycle
Request confirmation message
Send order confirmation
Schedule feedback follow-up

Key Steps:

1. **Recommendation Request**:
   - User requests scent recommendation through application interface
   - Frontend validates user session and authorization
   - Request sent to Recommendation Engine with user ID and context
2. **Similarity Computation**:
   - Recommendation Engine retrieves user embedding from Vector Database
   - Engine prepares search parameters (filters, algorithm configuration)
   - Vector Database performs approximate nearest neighbor (ANN) search
   - Search uses indexed embeddings for sub-100ms performance
   - Database returns top matching scent IDs with similarity scores
3. **Scent Selection**:
   - Recommendation Engine applies business rules:
     - Excludes previously recommended scents
     - Applies diversity rules for variety
     - Considers seasonal and trending factors
   - Engine selects optimal scent with highest relevance
4. **Order Placement**:

    - User reviews recommendation and decides to order
    - Order Processing Service retrieves shipping and payment information
    - Service prepares order in Amazon-compatible format
    - AmazonAPIAdapter handles API communication with retry logic
    - Amazon API processes order and returns confirmation
5. **Order Finalization**:
    - Order details recorded in Order Repository with tracking information
    - Subscription Manager updates user's subscription cycle
    - Notification Service sends confirmation to user
    - System schedules follow-up for feedback collection

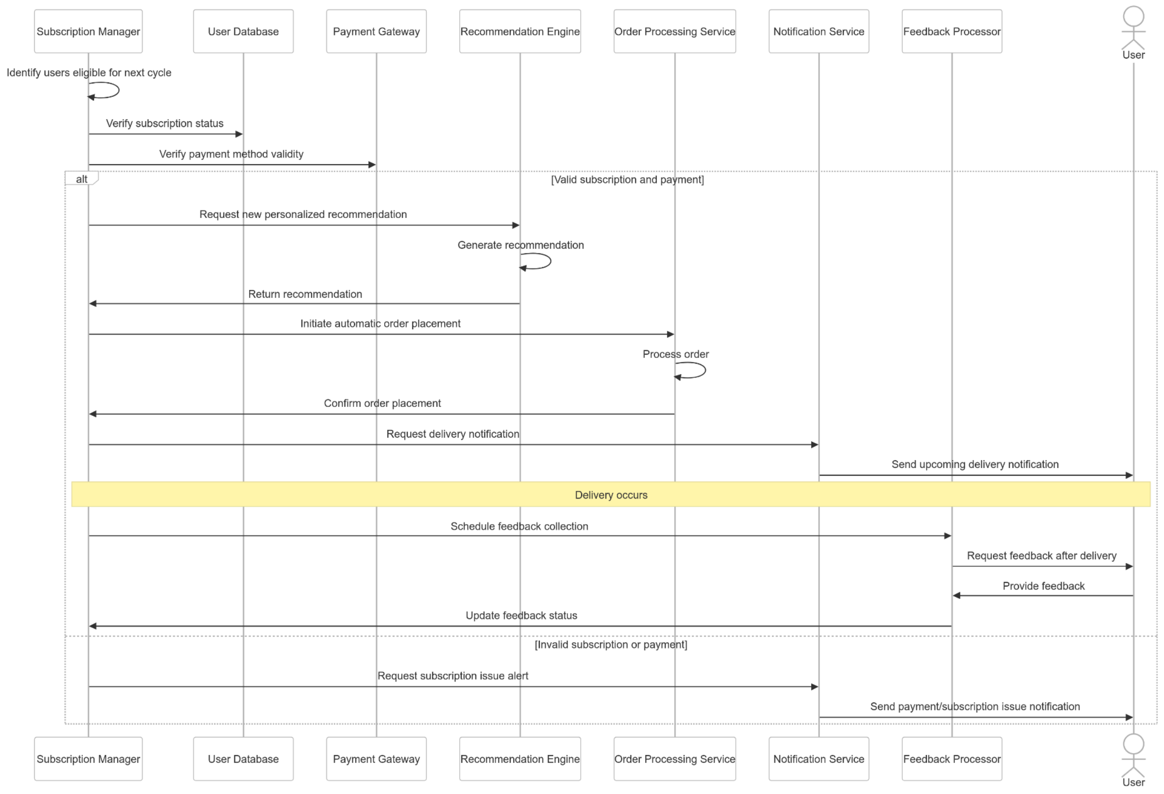
Feedback Collection Workflow



Key Steps:

1. System detects scent delivery and prompts for feedback
2. User provides rating and optional comments
3. Feedback Processor stores feedback and notifies Embedding Service
4. Embedding Service adjusts user embedding based on feedback
5. Updated embedding stored for improved future recommendations

Subscription Cycle Workflow


Subscription Manager
User Database
Payment Gateway
Recommendation Engine
Order Processing Service
Notification Service
Feedback Processor
User
Identify users eligible for next cycle
Verify subscription status
Verify payment method validity
alt
[Valid subscription and payment]
Request new personalized recommendation
Generate recommendation
Return recommendation
Initiate automatic order placement
Process order
Confirm order placement
Request delivery notification
Send upcoming delivery notification
Delivery occurs
Schedule feedback collection
Request feedback after delivery
Provide feedback
Update feedback status
[Invalid subscription or payment]
Request subscription issue alert
Send payment/subscription issue notification

Key Steps:

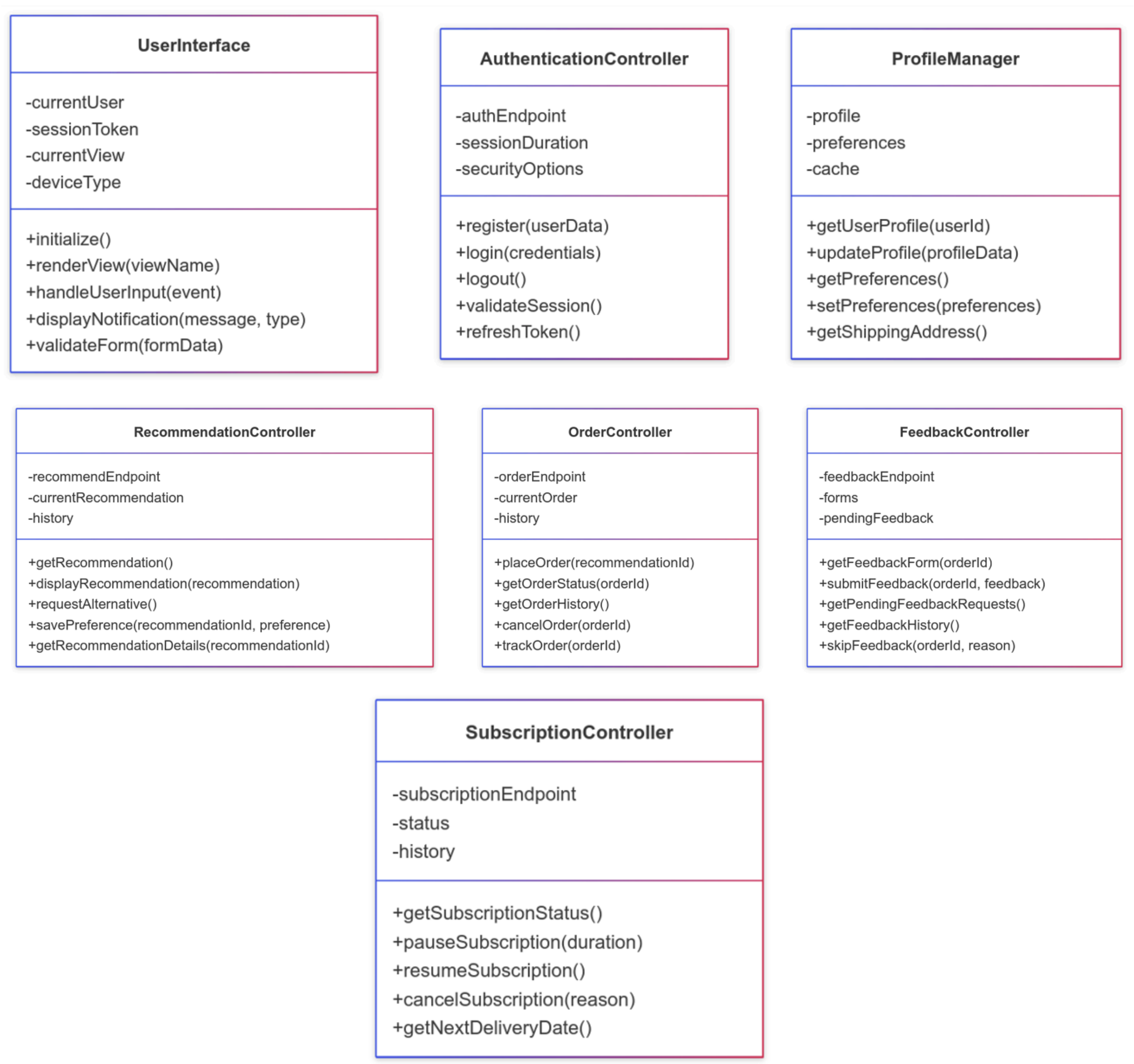
1. Subscription Manager identifies users eligible for next cycle
2. System verifies subscription status and payment method
3. Recommendation Engine generates new personalized recommendation
4. Order automatically placed through Order Processing Service
5. User notified of upcoming delivery
6. Feedback collected after delivery for continuous improvement

Structural Design

Key Components

Web Application Components

The class diagram below represents the key components for user interface and interaction:



Each component has specific responsibilities:

1. **UserInterface**: Central component for user interaction
    - Methods:
        - `initialize()`: Set up the interface
        - `renderView(viewName)`: Display specific view
        - `handleUserInput(event)`: Process user actions
        - `displayNotification(message, type)`: Show notifications
        - `validateForm(formData)`: Client-side validation
2. **AuthenticationController**: Handle user registration and authentication

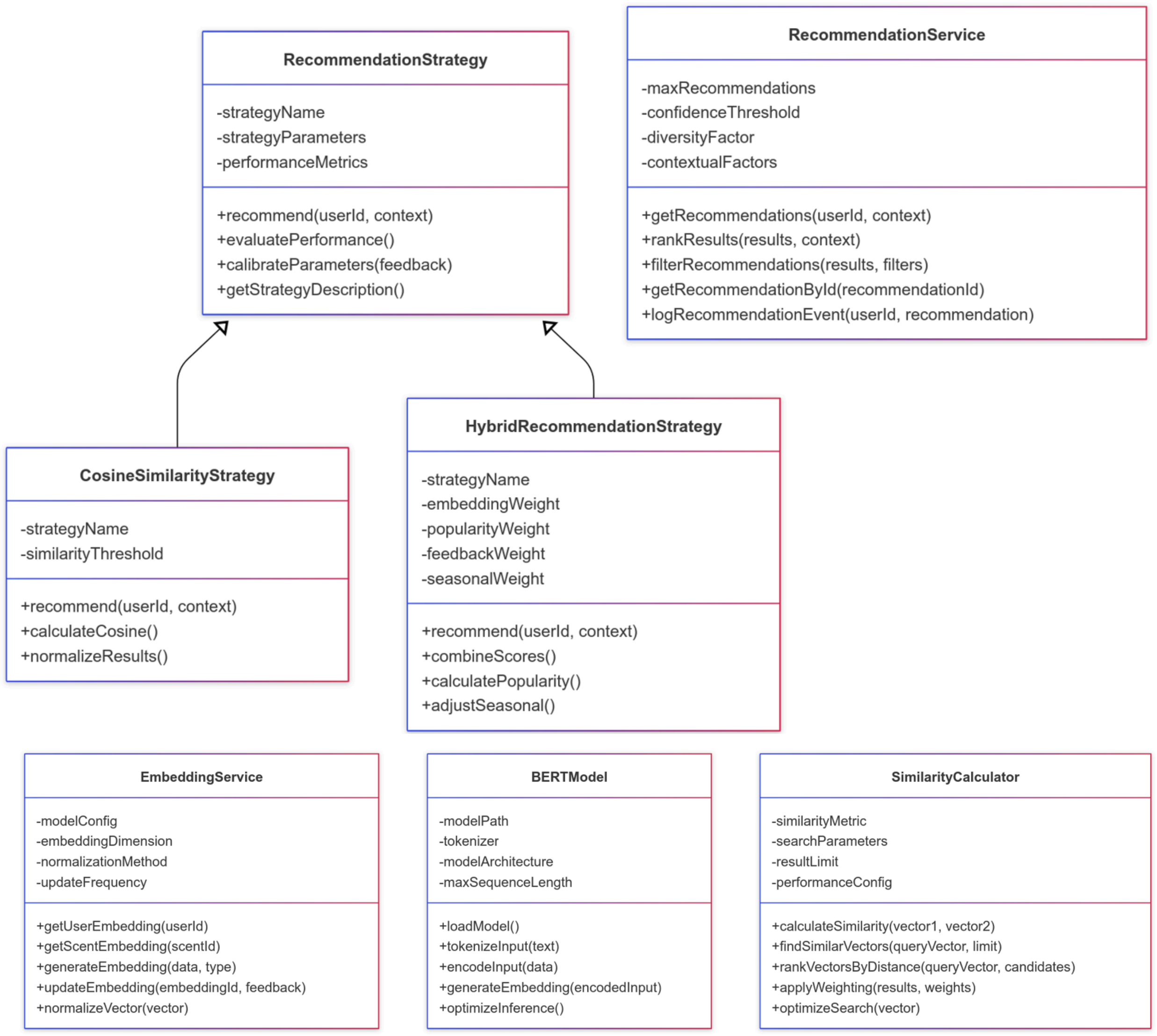
- Methods:
    - `register(userData)`: Create new user
    - `login(credentials)`: Authenticate user
    - `logout()`: End user session
    - `validateSession()`: Check token validity
    - `refreshToken()`: Renew authentication token

3. **ProfileManager**: Manage user profile information
    - Methods:
        - `getUserProfile(userId)`: Retrieve profile
        - `updateProfile(profileData)`: Update profile
        - `getPreferences()`: Get user preferences
        - `setPreferences(preferences)`: Update preferences
        - `getShippingAddress()`: Get shipping information
4. **RecommendationController**: Handle recommendation display
    - Methods:
        - `getRecommendation()`: Request recommendation
        - `displayRecommendation(recommendation)`: Show recommendation
        - `requestAlternative()`: Get different recommendation
        - `savePreference(recommendationId, preference)`: Record preference
        - `getRecommendationDetails(recommendationId)`: Get details
5. **OrderController**: Manage order placement and history
    - Methods:
        - `placeOrder(recommendationId)`: Create new order
        - `getOrderStatus(orderId)`: Check order status
        - `getOrderHistory()`: Get past orders
        - `cancelOrder(orderId)`: Cancel pending order
        - `trackOrder(orderId)`: Get tracking information
6. **FeedbackController**: Collect and submit user feedback
    - Methods:
        - `getFeedbackForm(orderId)`: Get feedback form
        - `submitFeedback(orderId, feedback)`: Submit feedback
        - `getPendingFeedbackRequests()`: Get orders needing feedback
        - `getFeedbackHistory()`: Get past feedback
        - `skipFeedback(orderId, reason)`: Opt out of feedback

7. **SubscriptionController**: Manage subscription settings
   - Methods:
     - `getSubscriptionStatus()`: Get current status
     - `pauseSubscription(duration)`: Temporarily pause
     - `resumeSubscription()`: Resume paused subscription
     - `cancelSubscription(reason)`: End subscription
     - `getNextDeliveryDate()`: Get upcoming delivery

Recommendation Engine Components

The class diagram below represents the key components that implement the vector-based recommendation functionality:



Each component has specific responsibilities:

1. **RecommendationService**: Central coordinator that manages the recommendation process
   - Methods:
     - `getRecommendations(userId, context)`: Generate personalized recommendations
     - `rankResults(results, context)`: Order recommendations by relevance
     - `filterRecommendations(results, filters)`: Apply business rules to filter results
     - `getRecommendationById(recommendationId)`: Retrieve specific recommendation
     - `logRecommendationEvent(userId, recommendation)`: Record analytics data
2. **EmbeddingService**: Handles generation and management of vector embeddings
   - Methods:
     - `getUserEmbedding(userId)`: Retrieve user's vector representation
     - `getScentEmbedding(scentId)`: Retrieve scent's vector representation
     - `generateEmbedding(data, type)`: Create new embedding from input data
     - `updateEmbedding(embeddingId, feedback)`: Modify existing embedding with feedback
     - `normalizeVector(vector)`: Standardize vector format for consistent processing
3. **BERTModel**: Core ML component that transforms input data into vector embeddings
   - Methods:
     - `loadModel()`: Initialize the BERT model
     - `tokenizeInput(text)`: Convert text to tokens for model input
     - `encodeInput(data)`: Prepare model input in required format
     - `generateEmbedding(encodedInput)`: Create embedding vector
     - `optimizeInference()`: Apply ONNX optimizations for performance
4. **SimilarityCalculator**: Performs vector similarity operations
   - Methods:
     - `calculateSimilarity(vector1, vector2)`: Compute similarity score

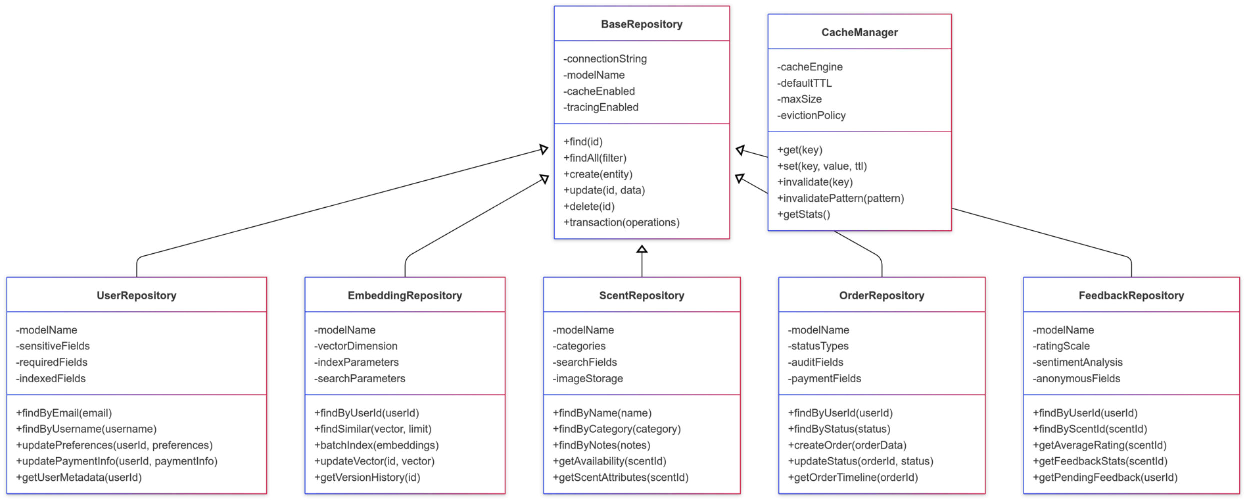
        - `findSimilarVectors(queryVector, limit)`: Find nearest neighbors in vector space
        - `rankVectorsByDistance(queryVector, candidates)`: Sort by similarity
        - `applyWeighting(results, weights)`: Adjust scores based on weights
        - `optimizeSearch(vector)`: Prepare vector for optimized search
5. **RecommendationStrategy**: Abstract interface for different recommendation algorithms

    - Methods:
        - `recommend(userId, context)`: Generate recommendations using specific strategy
        - `evaluatePerformance()`: Assess strategy effectiveness
        - `calibrateParameters(feedback)`: Auto-tune parameters based on feedback
        - `getStrategyDescription()`: Get algorithm details for explanation
6. **Strategy Implementations**:

    - **CosineSimilarityStrategy**: Uses cosine distance between vectors
    - **HybridRecommendationStrategy**: Combines multiple signals (embeddings, popularity, feedback, seasonal)

Data Management Components

The class diagram below represents the key components for data storage and retrieval:



Each component has specific responsibilities:

1. **BaseRepository**: Abstract base for all repositories

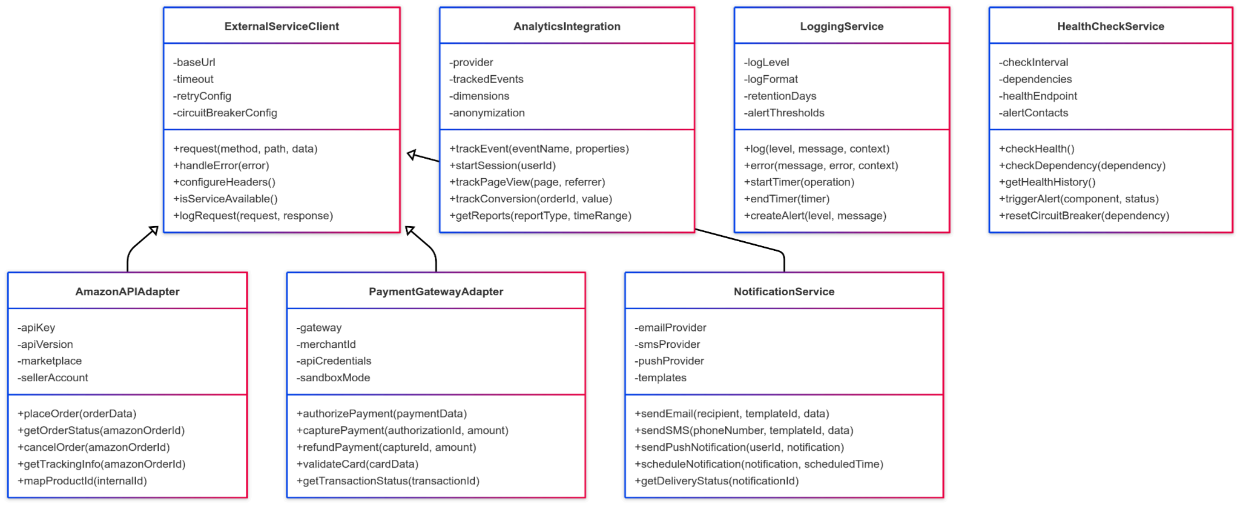
- Methods:
  - `find(id)`: Find entity by primary key
  - `findAll(filter)`: Find multiple entities
  - `create(entity)`: Create new entity
  - `update(id, data)`: Update entity
  - `delete(id)`: Remove entity
  - `transaction(operations)`: Execute atomic operations

2. **UserRepository**: Manages user profile data
   - Methods:
     - `findByEmail(email)`: Find user by email
     - `findByUsername(username)`: Find user by username
     - `updatePreferences(userId, preferences)`: Update user preferences
     - `updatePaymentInfo(userId, paymentInfo)`: Update payment information
     - `getUserMetadata(userId)`: Get user-related metadata
3. **EmbeddingRepository**: Store and retrieve vector embeddings
   - Methods:
     - `findByUserId(userId)`: Get user embedding
     - `findSimilar(vector, limit)`: Find similar vectors
     - `batchIndex(embeddings)`: Index multiple vectors
     - `updateVector(id, vector)`: Modify embedding
     - `getVersionHistory(id)`: Get historical versions
4. **ScentRepository**: Manage fragrance catalog data
   - Methods:
     - `findByName(name)`: Find by name
     - `findByCategory(category)`: Find by category
     - `findByNotes(notes)`: Find by fragrance notes
     - `getAvailability(scentId)`: Check stock status
     - `getScentAttributes(scentId)`: Get detailed attributes
5. **OrderRepository**: Store order data and history
   - Methods:
     - `findByUserId(userId)`: Get user orders
     - `findByStatus(status)`: Get orders by status
     - `createOrder(orderData)`: Create new order
     - `updateStatus(orderId, status)`: Change status

        - `getOrderTimeline(orderId)`: Get status history
6. **FeedbackRepository**: Store user feedback and ratings
    - Methods:
        - `findByUserId(userId)`: Get user feedback
        - `findByScentId(scentId)`: Get product feedback
        - `getAverageRating(scentId)`: Calculate average
        - `getFeedbackStats(scentId)`: Get statistical summary
        - `getPendingFeedback(userId)`: Get awaiting feedback
7. **CacheManager**: Handle data caching for performance
    - Methods:
        - `get(key)`: Retrieve from cache
        - `set(key, value, ttl)`: Store in cache
        - `invalidate(key)`: Remove from cache
        - `invalidatePattern(pattern)`: Remove matching keys
        - `getStats()`: Get cache performance metrics

Integration Components

The class diagram below represents the key components for external system integration:


ExternalServiceClient
-baseUrl
-timeout
-retryConfig
-circuitBreakerConfig
+request(method, path, data)
+handleError(error)
+configureHeaders()
+isServiceAvailable()
+logRequest(request, response)
AnalyticsIntegration
-provider
-trackedEvents
-dimensions
-anonymization
+trackEvent(eventName, properties)
+startSession(userId)
+trackPageView(page, referrer)
+trackConversion(orderId, value)
+getReports(reportType, timeRange)
LoggingService
-logLevel
-logFormat
-retentionDays
-alertThresholds
+log(level, message, context)
+error(message, error, context)
+startTimer(operation)
+endTimer(timer)
+createAlert(level, message)
HealthCheckService
-checkInterval
-dependencies
-healthEndpoint
-alertContacts
+checkHealth()
+checkDependency(dependency)
+getHealthHistory()
+triggerAlert(component, status)
+resetCircuitBreaker(dependency)
AmazonAPIAdapter
-apiKey
-apiVersion
-marketplace
-sellerAccount
+placeOrder(orderData)
+getOrderStatus(amazonOrderId)
+cancelOrder(amazonOrderId)
+getTrackingInfo(amazonOrderId)
+mapProductId(internalId)
PaymentGatewayAdapter
-gateway
-merchantId
-apiCredentials
-sandboxMode
+authorizePayment(paymentData)
+capturePayment(authorizationId, amount)
+refundPayment(captureId, amount)
+validateCard(cardData)
+getTransactionStatus(transactionId)
NotificationService
-emailProvider
-smsProvider
-pushProvider
-templates
+sendEmail(recipient, templateId, data)
+sendSMS(phoneNumber, templateId, data)
+sendPushNotification(userId, notification)
+scheduleNotification(notification, scheduledTime)
+getDeliveryStatus(notificationId)

Each component has specific responsibilities:

1. **ExternalServiceClient**: Base class for external service integration
    - Methods:
        - `request(method, path, data)`: Make API request
        - `handleError(error)`: Process and standardize errors

        - `configureHeaders()`: Set up authorization headers
        - `isServiceAvailable()`: Check service health
        - `logRequest(request, response)`: Log for monitoring
2. **AmazonAPIAdapter**: Interface with Amazon API for order fulfillment
    - Methods:
        - `placeOrder(orderData)`: Submit new order
        - `getOrderStatus(amazonOrderId)`: Check order status
        - `cancelOrder(amazonOrderId)`: Cancel pending order
        - `getTrackingInfo(amazonOrderId)`: Get tracking details
        - `mapProductId(internalId)`: Map to Amazon product ID
3. **PaymentGatewayAdapter**: Handle payment processing
    - Methods:
        - `authorizePayment(paymentData)`: Authorize payment
        - `capturePayment(authorizationId, amount)`: Complete payment
        - `refundPayment(captureId, amount)`: Issue refund
        - `validateCard(cardData)`: Verify card details
        - `getTransactionStatus(transactionId)`: Check status
4. **NotificationService**: Manage communication with users
    - Methods:
        - `sendEmail(recipient, templateId, data)`: Send email
        - `sendSMS(phoneNumber, templateId, data)`: Send SMS
        - `sendPushNotification(userId, notification)`: Send push
        - `scheduleNotification(notification, scheduledTime)`: Schedule future
        - `getDeliveryStatus(notificationId)`: Check delivery status
5. **AnalyticsIntegration**: Collect and analyze system metrics
    - Methods:
        - `trackEvent(eventName, properties)`: Record event
        - `startSession(userId)`: Begin user session
        - `trackPageView(page, referrer)`: Record page visit
        - `trackConversion(orderId, value)`: Record purchase
        - `getReports(reportType, timeRange)`: Generate analytics
6. **LoggingService**: Centralized logging and monitoring
    - Methods:
        - `log(level, message, context)`: Record log entry
        - `error(message, error, context)`: Log error

        - `startTimer(operation)`: Begin timing operation
        - `endTimer(timer)`: End timing and record
        - `createAlert(level, message)`: Generate system alert
7. **HealthCheckService**: Monitor system and dependency health
    - Methods:
        - `checkHealth()`: Perform comprehensive check
        - `checkDependency(dependency)`: Check specific dependency
        - `getHealthHistory()`: Get status history
        - `triggerAlert(component, status)`: Send alert
        - `resetCircuitBreaker(dependency)`: Force reset

References

Design Patterns and Architectural Styles

1. Circuit Breaker Pattern
    - Microsoft. (2024). Circuit Breaker Pattern - Azure Architecture Center. Microsoft Learn. Retrieved from https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker
    - Reselman, B. (2023, January 12). The pros and cons of the Circuit Breaker architecture pattern. Red Hat. Retrieved from https://www.redhat.com/en/blog/circuit-breaker-architecture-pattern
    - Java Design Patterns. (2024). Circuit Breaker Pattern in Java: Enhancing System Resilience. Retrieved from https://java-design-patterns.com/patterns/circuit-breaker/
2. Microservices Architecture
    - GeeksforGeeks. (2024, October 22). What is Circuit Breaker Pattern in Microservices? Retrieved from https://www.geeksforgeeks.org/what-is-circuit-breaker-pattern-in-microservices/
    - GeeksforGeeks. (2024, November 11). Software Architectural Patterns in System Design. Retrieved from https://www.geeksforgeeks.org/design-patterns-architecture/
    - Malandrino, P. (2023, November 3). Architecture Patterns: The Circuit-Breaker. DZone. Retrieved from https://dzone.com/articles/architecture-patterns-the-circuit-breaker

Technologies and Frameworks

3. PostgreSQL and pgvector

    ○ pgvector. (2024). pgvector: Open-source vector similarity search for Postgres. GitHub. Retrieved from https://github.com/pgvector/pgvector
    ○ Supabase. (2023, February 6). Storing OpenAI embeddings in Postgres with pgvector. Retrieved from https://supabase.com/blog/openai-embeddings-postgres-vector
    ○ Supabase. (2024). pgvector: Embeddings and vector similarity. Supabase Docs. Retrieved from https://supabase.com/docs/guides/database/extensions/pgvector
    ○ DataCamp. (2024, August 6). pgvector Tutorial: Integrate Vector Search into PostgreSQL. Retrieved from https://www.datacamp.com/tutorial/pgvector-tutorial

4. BERT and Embeddings

    ○ Hugging Face. (2024). BERT. Retrieved from https://huggingface.co/docs/transformers/en/model_doc/bert
    ○ Analytics Vidhya. (2023, September 13). Creating BERT Embeddings with Hugging Face Transformers. Retrieved from https://www.analyticsvidhya.com/blog/2023/08/bert-embeddings/
    ○ Scaler. (2023, April 14). Extracting embeddings from pre-trained BERT. Huggingface Transformers. Retrieved from https://www.scaler.com/topics/nlp/huggingface-transformers/
    ○ Hugging Face. (2024). Getting Started With Embeddings. Retrieved from https://huggingface.co/blog/getting-started-with-embeddings
    ○ BERTopic. (2024). Embeddings. Retrieved from https://maartengr.github.io/BERTopic/getting_started/embeddings/embeddings.html

5. FastAPI and Model Deployment

    ○ Venelin. (2024). Deploy BERT for Sentiment Analysis as REST API using PyTorch, Transformers by Hugging Face and FastAPI. Curiously. Retrieved from https://curiousily.com/posts/deploy-bert-for-sentiment-analysis-as-rest-api-using-pytorch-transformers-by-hugging-face-and-fastapi/
    ○ sshkhr. (2024). BERTdeploy: A simple example of deploying a pre-trained BERT model as a REST API. GitHub. Retrieved from https://github.com/sshkhr/BERTdeploy

6. Next.js and Vector Search

    ○ Vercel. (2024). Vercel Postgres pgvector Starter. Retrieved from https://vercel.com/templates/next.js/postgres-pgvector
    ○ Supabase. (2024). Vector search with Next.js and OpenAI. Supabase Docs. Retrieved from https://supabase.com/docs/guides/ai/examples/nextjs-vector-search

Additional Resources

7. Vector Databases and Search

    ○ Severalnines. (2024, March 20). Vector Similarity Search with PostgreSQL's pgvector - A Deep Dive. Retrieved from https://severalnines.com/blog/vector-similarity-search-with-postgresqls-pgvector-a-deep-dive/
    ○ Timescale. (2024, August 22). Similarity Search on PostgreSQL Using OpenAI Embeddings and Pgvector. Retrieved from https://www.timescale.com/blog/similarity-search-on-postgresql-using-openai-embeddings-and-pgvector
    ○ Tembo. (2024). Unleashing the power of vector embeddings with PostgreSQL. Retrieved from https://tembo.io/blog/pgvector-and-embedding-solutions-with-postgres

8. Tools and Integrations

    ○ Stack Overflow. (2024). Using pgvector-python in FastAPI to get most similar data. Retrieved from https://stackoverflow.com/questions/77697282/using-pgvector-python-in-fastapi-to-get-most-similar-data
    ○ Captum. (2024). Interpreting BERT Models (Part 1). Retrieved from https://captum.ai/tutorials/Bert_SQUAD_Interpret