Curated Events Platform - Hybrid Python/Go Technical Architecture

Language Strategy & Service Distribution

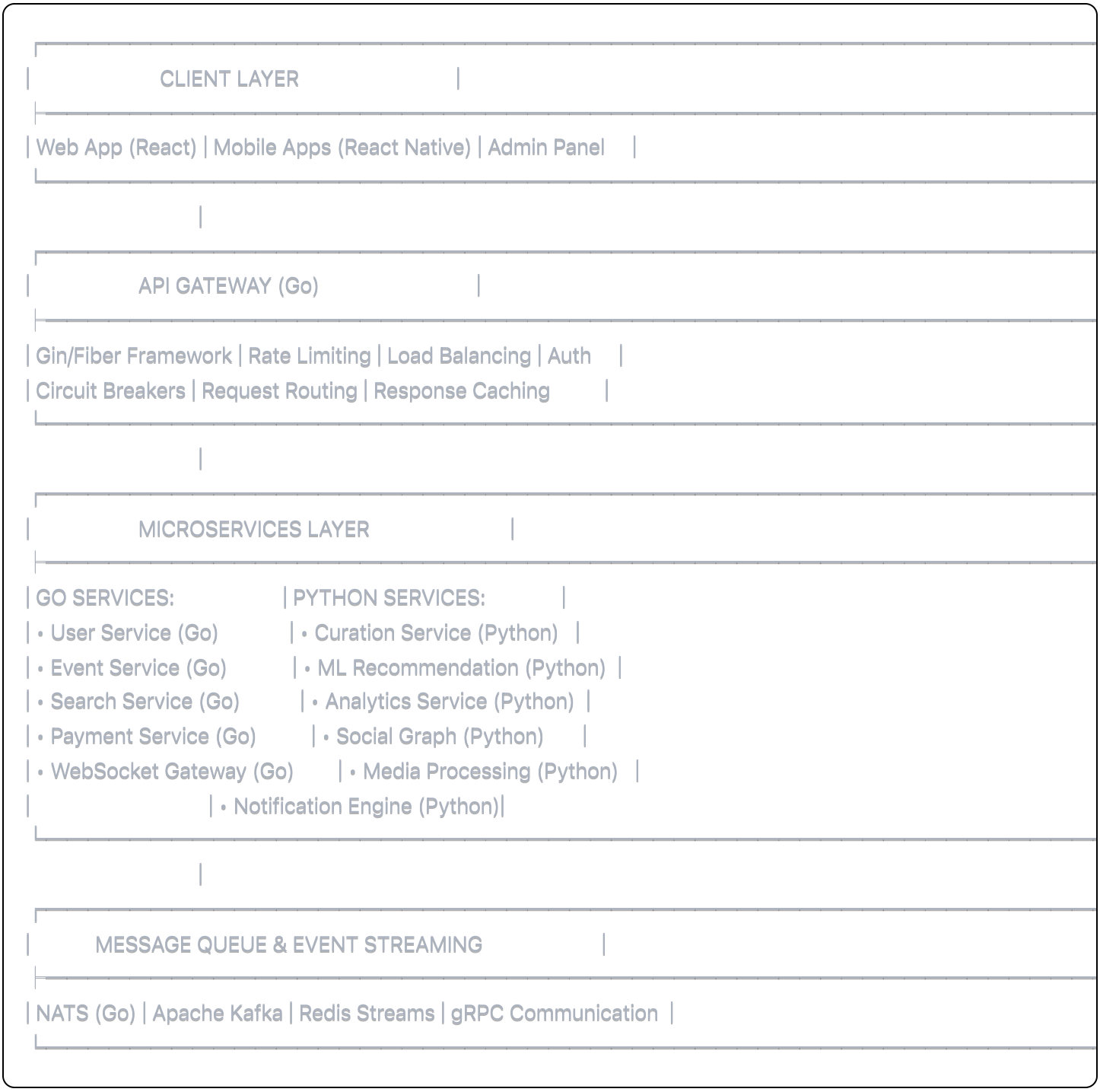
Go Services (High Concurrency & Performance Critical)

- **API Gateway & Load Balancer** - Goroutines for handling massive concurrent connections
- **User Service** - Authentication, sessions, and high-frequency user operations
- **Event Service** - Core CRUD operations with high read/write throughput
- **Search Service** - Real-time indexing and query processing
- **Payment Service** - Financial transactions requiring strict consistency
- **WebSocket Gateway** - Real-time notifications and live updates
- **Message Queue Workers** - High-throughput event processing

Python Services (ML/AI & Complex Business Logic)

- **Curation Service** - AI/ML-powered content analysis and quality scoring
- **Recommendation Engine** - Complex ML algorithms and data processing
- **Analytics Service** - Data science, reporting, and business intelligence
- **Social Service** - Graph algorithms and complex relationship processing
- **Media Processing Service** - Image/video processing and optimization
- **Notification Service** - Template engines and complex routing logic

Updated System Architecture



Core Technology Stack

Go Services Stack

- **Runtime**: Go 1.21+ with goroutines for concurrency
- **Web Framework**: Gin or Fiber for high-performance HTTP
- **gRPC**: For inter-service communication
- **Database**:
  - PostgreSQL with pgx driver (connection pooling)
  - Redis with go-redis for caching

- **Message Queues**: NATS for lightweight messaging
- **Monitoring**: Prometheus client libraries

Python Services Stack

- **Runtime**: Python 3.11+ with asyncio for async operations
- **Web Framework**: FastAPI for high-performance async APIs
- **ML/AI Stack**:
  - scikit-learn, TensorFlow, PyTorch
  - Pandas, NumPy for data processing
  - Celery for background ML tasks
- **Database**:
  - SQLAlchemy with asyncpg for PostgreSQL
  - Motor for async MongoDB operations
- **Message Queues**: aiokafka for Kafka integration

Service Deep Dive

1. API Gateway (Go)

Why Go: Exceptional concurrency handling, low latency, efficient memory usage

```go
```

```go
// High-performance API Gateway with Gin
package main

import (
    "context"
    "time"
    "github.com/gin-gonic/gin"
    "github.com/go-redis/redis/v8"
    "golang.org/x/time/rate"
)

type APIGateway struct {
    router      *gin.Engine
    rateLimit   *rate.Limiter
    redisClient *redis.Client
    services    map[string]ServiceClient
}

// Service registry for dynamic routing
type ServiceClient struct {
    BaseURL     string
    HealthCheck string
    Timeout     time.Duration
    MaxRetries  int
}

func (gw *APIGateway) setupRoutes() {
    // High-concurrency route handling
    gw.router.Use(gw.rateLimitMiddleware())
    gw.router.Use(gw.authMiddleware())
    gw.router.Use(gw.circuitBreakerMiddleware())

    // Service routing with load balancing
    api := gw.router.Group("/api/v1")
    {
        api.Any("/users/*path", gw.proxyToService("user-service"))
        api.Any("/events/*path", gw.proxyToService("event-service"))
        api.Any("/search/*path", gw.proxyToService("search-service"))
        api.Any("/payments/*path", gw.proxyToService("payment-service"))

        // Route to Python services
        api.Any("/curation/*path", gw.proxyToService("curation-service"))
        api.Any("/recommendations/*path", gw.proxyToService("ml-service"))
```

```
        api.Any("/analytics/*path", gw.proxyToService("analytics-service"))
    }
}

func (gw *APIGateway) proxyToService(serviceName string) gin.HandlerFunc {
    return func(c *gin.Context) {
        // Concurrent request handling with goroutines
        go gw.logRequest(c)

        service, exists := gw.services[serviceName]
        if !exists {
            c.JSON(404, gin.H{"error": "Service not found"})
            return
        }

        // Implement circuit breaker pattern
        if !gw.isServiceHealthy(serviceName) {
            c.JSON(503, gin.H{"error": "Service unavailable"})
            return
        }

        // Forward request with timeout
        ctx, cancel := context.WithTimeout(c.Request.Context(), service.Timeout)
        defer cancel()

        // Proxy logic here
        gw.forwardRequest(ctx, c, service)
    }
}
```

2. User Service (Go)

Why Go: High-frequency operations, session management, concurrent authentication

```
go
```

```go
// User Service with high-concurrency authentication
type UserService struct {
    db          *pgxpool.Pool
    redis       *redis.Client
    jwtSecret   []byte
    rateLimiter *rate.Limiter
}

// Concurrent user authentication
func (us *UserService) AuthenticateUser(ctx context.Context, req *AuthRequest) (*AuthResponse, error) {
    // Use goroutines for parallel operations
    var user User
    var sessionData []byte

    // Parallel database and cache queries
    errChan := make(chan error, 2)

    go func() {
        err := us.db.QueryRow(ctx,
            "SELECT id, email, password_hash, verified FROM users WHERE email = $1",
            req.Email).Scan(&user.ID, &user.Email, &user.PasswordHash, &user.Verified)
        errChan <- err
    }()

    go func() {
        var err error
        sessionData, err = us.redis.Get(ctx, fmt.Sprintf("session:%s", req.Email)).Bytes()
        errChan <- err
    }()

    // Wait for both operations
    for i := 0; i < 2; i++ {
        if err := <-errChan; err != nil && err != redis.Nil {
            return nil, err
        }
    }

    // Verify password concurrently
    if !us.verifyPassword(user.PasswordHash, req.Password) {
        return nil, errors.New("invalid credentials")
    }

    // Generate JWT and update session
```

```
    token, err := us.generateJWT(user)
    if err != nil {
        return nil, err
    }

    return &AuthResponse{Token: token, User: user}, nil
}

// High-performance session management
func (us *UserService) ValidateSession(ctx context.Context, token string) (*User, error) {
    // Use Redis pipeline for batch operations
    pipe := us.redis.Pipeline()

    // Multiple Redis operations in single round trip
    sessionKey := pipe.Get(ctx, fmt.Sprintf("session:%s", token))
    userKey := pipe.HGetAll(ctx, fmt.Sprintf("user:%s", token))

    _, err := pipe.Exec(ctx)
    if err != nil {
        return nil, err
    }

    // Process results concurrently
    var user User
    // Parse session and user data

    return &user, nil
}
```

3. Event Service (Go)

Why Go: High read/write throughput, concurrent CRUD operations, real-time updates

```
go
```

```go
// Event Service with optimized concurrent operations
type EventService struct {
    db         *pgxpool.Pool
    redis      *redis.Client
    searchIndex *elasticsearch.Client
    pubsub     *nats.Conn
}

// Concurrent event creation with multiple data stores
func (es *EventService) CreateEvent(ctx context.Context, event *Event) (*Event, error) {
    // Start database transaction
    tx, err := es.db.Begin(ctx)
    if err != nil {
        return nil, err
    }
    defer tx.Rollback(ctx)

    // Generate ID and timestamps
    event.ID = uuid.New()
    event.CreatedAt = time.Now()
    event.UpdatedAt = time.Now()

    // Insert into PostgreSQL
    err = tx.QueryRow(ctx,
        `INSERT INTO events (id, title, description, creator_id, start_time, end_time,
         location, created_at, updated_at)
         VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9) RETURNING id`,
        event.ID, event.Title, event.Description, event.CreatorID,
        event.StartTime, event.EndTime, event.Location, event.CreatedAt, event.UpdatedAt,
    ).Scan(&event.ID)

    if err != nil {
        return nil, err
    }

    // Commit transaction
    if err = tx.Commit(ctx); err != nil {
        return nil, err
    }

    // Concurrent operations after successful DB insert
    errChan := make(chan error, 3)
```

```
    // Cache in Redis
    go func() {
        eventJSON, _ := json.Marshal(event)
        err := es.redis.Set(ctx, fmt.Sprintf("event:%s", event.ID), eventJSON, time.Hour).Err()
        errChan <- err
    }()

    // Index in Elasticsearch
    go func() {
        _, err := es.searchIndex.Index(
            "events",
            bytes.NewReader(eventJSON),
            es.searchIndex.Index.WithDocumentID(event.ID.String()),
        )
        errChan <- err
    }()

    // Publish event creation
    go func() {
        eventMsg, _ := json.Marshal(map[string]interface{}{
            "type": "event.created",
            "event": event,
        })
        err := es.pubsub.Publish("events", eventMsg)
        errChan <- err
    }()

    // Wait for all operations (best effort)
    for i := 0; i < 3; i++ {
        <-errChan // Log errors but don't fail the request
    }

    return event, nil
}

// High-performance event querying with caching
func (es *EventService) GetEvents(ctx context.Context, filter *EventFilter) ([]*Event, error) {
    cacheKey := es.buildCacheKey(filter)

    // Try cache first
    if cached, err := es.redis.Get(ctx, cacheKey).Result(); err == nil {
        var events []*Event
        if json.Unmarshal([]byte(cached), &events) == nil {
            return events, nil
```

```
    }
  }

  // Build dynamic query with proper indexing
  query := es.buildQuery(filter)

  rows, err := es.db.Query(ctx, query.SQL, query.Args...)
  if err != nil {
    return nil, err
  }
  defer rows.Close()

  var events []*Event
  for rows.Next() {
    var event Event
    err := rows.Scan(
      &event.ID, &event.Title, &event.Description,
      &event.CreatorID, &event.StartTime, &event.EndTime,
      &event.Location, &event.CreatedAt, &event.UpdatedAt,
    )
    if err != nil {
      return nil, err
    }
    events = append(events, &event)
  }

  // Cache results asynchronously
  go func() {
    if eventsJSON, err := json.Marshal(events); err == nil {
      es.redis.Set(context.Background(), cacheKey, eventsJSON, 10*time.Minute)
    }
  }()

  return events, nil
}
```

4. Search Service (Go)

Why Go: Real-time indexing, concurrent query processing, low-latency responses

go

```
// High-performance search service
type SearchService struct {
    es        *elasticsearch.Client
    redis     *redis.Client
    indexPool  sync.Pool
}

func (ss *SearchService) SearchEvents(ctx context.Context, req *SearchRequest) (*SearchResponse, error) {
    // Use object pooling for query builders
    queryBuilder := ss.indexPool.Get().(*QueryBuilder)
    defer ss.indexPool.Put(queryBuilder)

    // Build Elasticsearch query
    esQuery := queryBuilder.Build(req)

    // Concurrent search with aggregations
    var searchResp *esapi.Response
    var aggsResp *esapi.Response

    errChan := make(chan error, 2)

    // Main search query
    go func() {
        var err error
        searchResp, err = ss.es.Search(
            ss.es.Search.WithContext(ctx),
            ss.es.Search.WithIndex("events"),
            ss.es.Search.WithBody(bytes.NewReader(esQuery)),
            ss.es.Search.WithTrackTotalHits(true),
        )
        errChan <- err
    }()

    // Aggregations for facets
    go func() {
        aggsQuery := queryBuilder.BuildAggregations(req)
        var err error
        aggsResp, err = ss.es.Search(
            ss.es.Search.WithContext(ctx),
            ss.es.Search.WithIndex("events"),
            ss.es.Search.WithBody(bytes.NewReader(aggsQuery)),
            ss.es.Search.WithSize(0),
        )
```

```
        errChan <- err
    }()

    // Wait for both queries
    for i := 0; i < 2; i++ {
        if err := <-errChan; err != nil {
            return nil, err
        }
    }

    defer searchResp.Body.Close()
    defer aggsResp.Body.Close()

    // Parse results concurrently
    var events []Event
    var facets SearchFacets

    parseErrChan := make(chan error, 2)

    go func() {
        events, err := ss.parseSearchResults(searchResp.Body)
        parseErrChan <- err
    }()

    go func() {
        facets, err := ss.parseAggregations(aggsResp.Body)
        parseErrChan <- err
    }()

    for i := 0; i < 2; i++ {
        if err := <-parseErrChan; err != nil {
            return nil, err
        }
    }

    return &SearchResponse{
        Events: events,
        Facets: facets,
        Total:  len(events),
    }, nil
}
```

5. Curation Service (Python)

Why Python: ML/AI libraries, complex algorithms, data processing

```python
```

```python
# AI-powered curation service with async processing
from fastapi import FastAPI
from sqlalchemy.ext.asyncio import AsyncSession
import asyncio
import aioredis
from transformers import pipeline
import torch

class CurationService:
    def __init__(self):
        self.content_classifier = pipeline("text-classification",
                                           model="distilbert-base-uncased")
        self.image_analyzer = torch.jit.load("models/image_quality_model.pt")
        self.spam_detector = pipeline("text-classification",
                                      model="models/spam-detector")

    async def screen_event(self, event_data: dict) -> CurationResult:
        """AI-powered event screening with parallel processing"""

        # Run multiple AI models concurrently
        tasks = [
            self.analyze_content_quality(event_data["description"]),
            self.detect_spam_indicators(event_data),
            self.analyze_image_quality(event_data.get("cover_image")),
            self.check_completeness(event_data),
            self.verify_venue_authenticity(event_data.get("venue"))
        ]

        # Execute all analyses in parallel
        results = await asyncio.gather(*tasks, return_exceptions=True)

        content_score, spam_score, image_score, completeness, venue_valid = results

        # Calculate overall quality score
        overall_score = self.calculate_weighted_score({
            "content_quality": content_score,
            "spam_probability": 1 - spam_score,  # Invert spam score
            "image_quality": image_score,
            "completeness": completeness,
            "venue_authenticity": venue_valid
        })

        # Make curation decision
```

```
    recommendation = self.make_curation_decision(overall_score, results)

    return CurationResult(
        overall_score=overall_score,
        recommendation=recommendation,
        detailed_scores=results,
        processed_at=datetime.utcnow()
    )

async def analyze_content_quality(self, description: str) -> float:
    """Analyze content quality using transformer models"""
    if not description:
        return 0.0

    # Use asyncio for CPU-intensive NLP processing
    loop = asyncio.get_event_loop()

    # Run in thread pool to avoid blocking
    quality_result = await loop.run_in_executor(
        None,
        self.content_classifier,
        description
    )

    # Extract quality score from classifier output
    quality_score = max(result["score"] for result in quality_result
                        if result["label"] == "HIGH_QUALITY")

    # Additional heuristics
    length_score = min(len(description) / 500, 1.0)  # Prefer detailed descriptions
    readability_score = await self.calculate_readability(description)

    return (quality_score * 0.6 + length_score * 0.2 + readability_score * 0.2)

async def detect_spam_indicators(self, event_data: dict) -> float:
    """Multi-layered spam detection"""
    spam_indicators = []

    # Check text for spam patterns
    description = event_data.get("description", "")
    title = event_data.get("title", "")

    # Run spam detection on text
    spam_result = await asyncio.get_event_loop().run_in_executor(
```

```
            None,
            self.spam_detector,
            f"{title} {description}"
        )

        text_spam_score = spam_result[0]["score"] if spam_result[0]["label"] == "SPAM" else 0

        # Check for suspicious patterns
        suspicious_patterns = [
            len(re.findall(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\\(\\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+', description)) > 3,
            description.count('!') > 5,
            len(description.split()) < 10,
            any(word in description.lower() for word in ["free money", "get rich", "guaranteed"])
        ]

        pattern_score = sum(suspicious_patterns) / len(suspicious_patterns)

        return 1 - (text_spam_score * 0.7 + pattern_score * 0.3)

    async def analyze_image_quality(self, image_url: str) -> float:
        """AI-powered image quality analysis"""
        if not image_url:
            return 0.5  # Neutral score for missing images

        try:
            # Download and process image asynchronously
            async with aiohttp.ClientSession() as session:
                async with session.get(image_url) as response:
                    image_data = await response.read()

            # Process image with PyTorch model
            image_tensor = self.preprocess_image(image_data)

            with torch.no_grad():
                quality_score = self.image_analyzer(image_tensor).item()

            return quality_score

        except Exception as e:
            logger.error(f"Image analysis failed: {e}")
            return 0.3  # Low score for processing failures

# Async batch processing for high throughput
class BatchCurationProcessor:
```

```python
    def __init__(self, curation_service: CurationService):
        self.curation_service = curation_service
        self.batch_size = 50

    async def process_batch(self, events: List[dict]) -> List[CurationResult]:
        """Process events in optimized batches"""
        # Split into batches to prevent memory issues
        batches = [events[i:i + self.batch_size]
                   for i in range(0, len(events), self.batch_size)]

        all_results = []

        for batch in batches:
            # Process each batch concurrently
            batch_tasks = [
                self.curation_service.screen_event(event)
                for event in batch
            ]

            batch_results = await asyncio.gather(*batch_tasks)
            all_results.extend(batch_results)

            # Small delay between batches to prevent overwhelming downstream services
            await asyncio.sleep(0.1)

        return all_results
```

6. Recommendation Engine (Python)

Why Python: Advanced ML algorithms, data science libraries, complex feature engineering

```python
```

```python
# High-performance recommendation engine
import asyncio
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import redis.asyncio as aioredis
from concurrent.futures import ThreadPoolExecutor

class EventRecommendationEngine:
    def __init__(self):
        self.collaborative_model = CollaborativeFilteringModel()
        self.content_model = ContentBasedModel()
        self.deep_model = DeepRecommendationModel()
        self.redis = aioredis.from_url("redis://localhost")
        self.executor = ThreadPoolExecutor(max_workers=4)

    async def get_recommendations(self, user_id: str, limit: int = 20) -> List[dict]:
        """Hybrid recommendation system with async processing"""

        # Fetch user data and event catalog concurrently
        user_data, events_data, user_history = await asyncio.gather(
            self.get_user_profile(user_id),
            self.get_events_catalog(),
            self.get_user_interaction_history(user_id)
        )

        # Run different recommendation algorithms in parallel
        recommendation_tasks = [
            self.get_collaborative_recommendations(user_id, user_history, limit * 2),
            self.get_content_based_recommendations(user_data, events_data, limit * 2),
            self.get_deep_learning_recommendations(user_id, limit * 2),
            self.get_trending_recommendations(user_data["location"], limit),
        ]

        # Execute all recommendation algorithms concurrently
        collab_recs, content_recs, deep_recs, trending_recs = await asyncio.gather(
            *recommendation_tasks
        )

        # Combine and rank recommendations
        final_recommendations = await self.hybrid_ranking(
            collab_recs, content_recs, deep_recs, trending_recs,
```

```
            user_data, limit
        )

        # Cache results for faster subsequent requests
        await self.cache_recommendations(user_id, final_recommendations)

        return final_recommendations

    async def apply_diversity_optimization(self, event_scores: dict,
                                           user_data: dict, limit: int) -> List[dict]:
        """Optimize recommendations for diversity and user engagement"""

        # Sort by total score
        sorted_events = sorted(event_scores.items(),
                               key=lambda x: x[1]["total_score"],
                               reverse=True)

        selected_events = []
        category_counts = {}
        time_slots = {}

        for event_id, event_data in sorted_events:
            if len(selected_events) >= limit:
                break

            event = event_data["event"]
            category = event.get("category", "other")
            time_slot = self.get_time_slot(event.get("start_time"))

            # Diversity constraints
            if category_counts.get(category, 0) >= limit // 3:  # Max 1/3 from same category
                continue
            if time_slots.get(time_slot, 0) >= limit // 2:  # Max 1/2 in same time slot
                continue

            selected_events.append(event_data)
            category_counts[category] = category_counts.get(category, 0) + 1
            time_slots[time_slot] = time_slots.get(time_slot, 0) + 1

        return selected_events

# Real-time feature store for ML models
class FeatureStore:
    def __init__(self):
```

```
        self.redis = aioredis.from_url("redis://localhost")
        self.batch_size = 1000

    async def update_user_features(self, user_id: str, features: dict):
        """Update user features in real-time"""
        feature_key = f"features:user:{user_id}"

        # Use Redis hash for efficient partial updates
        await self.redis.hset(feature_key, mapping=features)
        await self.redis.expire(feature_key, 86400)  # 24 hour expiry

    async def batch_update_event_features(self, events_features: List[dict]):
        """Batch update event features for efficiency"""
        pipe = self.redis.pipeline()

        for event_feature in events_features:
            event_id = event_feature["event_id"]
            features = event_feature["features"]

            pipe.hset(f"features:event:{event_id}", mapping=features)
            pipe.expire(f"features:event:{event_id}", 3600)  # 1 hour expiry

        await pipe.execute()
```

7. Analytics Service (Python)

Why Python: Data science libraries, statistical analysis, reporting

python

```python
# Advanced analytics service with real-time processing
import asyncio
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import asyncpg
from clickhouse_driver import Client as ClickHouseClient
import plotly.graph_objects as go
import plotly.express as px

class AnalyticsService:
    def __init__(self):
        self.clickhouse = ClickHouseClient(host='localhost')
        self.redis = aioredis.from_url("redis://localhost")
        self.ml_pipeline = MLAnalyticsPipeline()

    async def generate_event_analytics(self, event_id: str,
                                       time_range: dict) -> dict:
        """Generate comprehensive event analytics"""

        # Run multiple analytics queries concurrently
        analytics_tasks = [
            self.get_event_engagement_metrics(event_id, time_range),
            self.get_user_demographics(event_id),
            self.get_conversion_funnel(event_id, time_range),
            self.get_social_media_metrics(event_id, time_range),
            self.predict_attendance_trends(event_id)
        ]

        results = await asyncio.gather(*analytics_tasks)

        engagement_metrics, demographics, funnel, social_metrics, predictions = results

        # Generate visualizations
        charts = await self.generate_analytics_charts(
            engagement_metrics, demographics, funnel, social_metrics
        )

        return {
            "event_id": event_id,
            "engagement": engagement_metrics,
            "demographics": demographics,
            "conversion_funnel": funnel,
```

```
        "social_metrics": social_metrics,
        "predictions": predictions,
        "charts": charts,
        "generated_at": datetime.utcnow().isoformat()
    }

async def get_event_engagement_metrics(self, event_id: str,
                                       time_range: dict) -> dict:
    """Real-time engagement metrics from ClickHouse"""

    query = """
    SELECT
        toHour(timestamp) as hour,
        countIf(event_type = 'view') as views,
        countIf(event_type = 'interested') as interested,
        countIf(event_type = 'share') as shares,
        countIf(event_type = 'save') as saves,
        uniqIf(user_id, event_type = 'view') as unique_viewers,
        avgIf(session_duration, event_type = 'view') as avg_session_duration
    FROM analytics_events
    WHERE event_id = %(event_id)s
    AND timestamp BETWEEN %(start_time)s AND %(end_time)s
    GROUP BY hour
    ORDER BY hour
    """

    # Execute query asynchronously
    loop = asyncio.get_event_loop()
    result = await loop.run_in_executor(
        None,
        self.clickhouse.execute,
        query,
        {
            'event_id': event_id,
            'start_time': time_range['start'],
            'end_time': time_range['end']
        }
    )

    # Process results into structured format
    hourly_metrics = []
    for row in result:
        hourly_metrics.append({
            "hour": row[0],
```

```
            "views": row[1],
            "interested": row[2],
            "shares": row[3],
            "saves": row[4],
            "unique_viewers": row[5],
            "avg_session_duration": row[6]
        })

    # Calculate summary statistics
    total_views = sum(m["views"] for m in hourly_metrics)
    total_interested = sum(m["interested"] for m in hourly_metrics)
    engagement_rate = (total_interested / total_views) if total_views > 0 else 0

    return {
        "hourly_metrics": hourly_metrics,
        "summary": {
            "total_views": total_views,
            "total_interested": total_interested,
            "engagement_rate": engagement_rate,
            "peak_hour": max(hourly_metrics, key=lambda x: x["views"])["hour"]
        }
    }

async def get_user_demographics(self, event_id: str) -> dict:
    """Analyze user demographics for event attendees"""

    query = """
    SELECT
        u.age_group,
        u.gender,
        u.location_city,
        u.interests,
        COUNT(*) as count
    FROM analytics_events ae
    JOIN users u ON ae.user_id = u.id
    WHERE ae.event_id = %(event_id)s
    AND ae.event_type IN ('interested', 'attending')
    GROUP BY u.age_group, u.gender, u.location_city, u.interests
    """

    loop = asyncio.get_event_loop()
    result = await loop.run_in_executor(
        None, self.clickhouse.execute, query, {'event_id': event_id}
    )
```

```
        # Process demographics data
        demographics = {
            "age_groups": {},
            "gender_distribution": {},
            "top_cities": {},
            "interest_overlap": {}
        }

        for row in result:
            age_group, gender, city, interests, count = row

            demographics["age_groups"][age_group] = demographics["age_groups"].get(age_group, 0) + count
            demographics["gender_distribution"][gender] = demographics["gender_distribution"].get(gender, 0) + co
            demographics["top_cities"][city] = demographics["top_cities"].get(city, 0) + count

            for interest in interests:
                demographics["interest_overlap"][interest] = demographics["interest_overlap"].get(interest, 0) + count

        return demographics

    async def predict_attendance_trends(self, event_id: str) -> dict:
        """ML-based attendance prediction"""

        # Get historical data for similar events
        historical_data = await self.get_similar_events_data(event_id)

        # Use ML pipeline for predictions
        loop = asyncio.get_event_loop()
        predictions = await loop.run_in_executor(
            None,
            self.ml_pipeline.predict_attendance,
            event_id, historical_data
        )

        return {
            "predicted_attendance": predictions["final_attendance"],
            "confidence_interval": predictions["confidence_interval"],
            "trend_factors": predictions["factors"],
            "similar_events_count": len(historical_data)
        }

# Real-time analytics dashboard data preparation
class DashboardDataPipeline:
```

```python
def __init__(self):
    self.redis = aioredis.from_url("redis://localhost")
    self.update_interval = 30  # seconds

async def start_real_time_updates(self):
    """Start background task for real-time dashboard updates"""
    while True:
        try:
            await self.update_dashboard_metrics()
            await asyncio.sleep(self.update_interval)
        except Exception as e:
            logger.error(f"Dashboard update failed: {e}")
            await asyncio.sleep(self.update_interval * 2)  # Back off on error

async def update_dashboard_metrics(self):
    """Update key metrics for real-time dashboards"""

    # Calculate metrics concurrently
    tasks = [
        self.calculate_platform_metrics(),
        self.calculate_trending_events(),
        self.calculate_user_activity_metrics(),
        self.calculate_curation_metrics(),
        self.calculate_revenue_metrics()
    ]

    results = await asyncio.gather(*tasks, return_exceptions=True)

    # Store in Redis for fast dashboard access
    dashboard_data = {
        "platform_metrics": results[0],
        "trending_events": results[1],
        "user_activity": results[2],
        "curation_metrics": results[3],
        "revenue_metrics": results[4],
        "last_updated": datetime.utcnow().isoformat()
    }

    await self.redis.set("dashboard:realtime",
                         json.dumps(dashboard_data),
                         ex=300)  # 5 minute expiry
```

8. WebSocket Gateway (Go)

Why Go: Excellent concurrency for real-time connections, low memory overhead

```go
```

```go
// High-performance WebSocket gateway for real-time features
package main

import (
    "context"
    "encoding/json"
    "log"
    "net/http"
    "sync"
    "time"

    "github.com/gorilla/websocket"
    "github.com/nats-io/nats.go"
    "github.com/go-redis/redis/v8"
)

type WebSocketGateway struct {
    clients    sync.Map          // Concurrent-safe client storage
    upgrader   websocket.Upgrader
    nats       *nats.Conn
    redis      *redis.Client
    broadcasts chan []byte
}

type Client struct {
    ID       string
    UserID   string
    Conn     *websocket.Conn
    Send     chan []byte
    Rooms    map[string]bool
    LastSeen time.Time
}

func NewWebSocketGateway() *WebSocketGateway {
    return &WebSocketGateway{
        upgrader: websocket.Upgrader{
            CheckOrigin: func(r *http.Request) bool { return true },
            BufferSize:  1024,
        },
        broadcasts: make(chan []byte, 1000),
    }
}
```

```go
func (gw *WebSocketGateway) HandleConnection(w http.ResponseWriter, r *http.Request) {
    conn, err := gw.upgrader.Upgrade(w, r, nil)
    if err != nil {
        log.Printf("WebSocket upgrade failed: %v", err)
        return
    }

    client := &Client{
        ID:       generateClientID(),
        UserID:   r.Header.Get("X-User-ID"),
        Conn:     conn,
        Send:     make(chan []byte, 256),
        Rooms:    make(map[string]bool),
        LastSeen: time.Now(),
    }

    // Store client
    gw.clients.Store(client.ID, client)
    defer gw.clients.Delete(client.ID)

    // Start goroutines for handling this client
    go gw.handleClientWrites(client)
    go gw.handleClientReads(client)

    // Subscribe to user-specific channels
    go gw.subscribeToUserChannels(client)
}

func (gw *WebSocketGateway) handleClientReads(client *Client) {
    defer func() {
        client.Conn.Close()
        gw.clients.Delete(client.ID)
    }()

    client.Conn.SetReadLimit(512)
    client.Conn.SetReadDeadline(time.Now().Add(60 * time.Second))
    client.Conn.SetPongHandler(func(string) error {
        client.Conn.SetReadDeadline(time.Now().Add(60 * time.Second))
        client.LastSeen = time.Now()
        return nil
    })

    for {
        var message map[string]interface{}
```

```
        err := client.Conn.ReadJSON(&message)
        if err != nil {
            if websocket.IsUnexpectedCloseError(err, websocket.CloseGoingAway, websocket.CloseAbnormalClosure) {
                log.Printf("WebSocket error: %v", err)
            }
            break
        }

        // Handle different message types
        go gw.handleClientMessage(client, message)
    }
}

func (gw *WebSocketGateway) handleClientWrites(client *Client) {
    ticker := time.NewTicker(54 * time.Second)
    defer func() {
        ticker.Stop()
        client.Conn.Close()
    }()

    for {
        select {
        case message, ok := <-client.Send:
            client.Conn.SetWriteDeadline(time.Now().Add(10 * time.Second))
            if !ok {
                client.Conn.WriteMessage(websocket.CloseMessage, []byte{})
                return
            }

            if err := client.Conn.WriteMessage(websocket.TextMessage, message); err != nil {
                return
            }

        case <-ticker.C:
            client.Conn.SetWriteDeadline(time.Now().Add(10 * time.Second))
            if err := client.Conn.WriteMessage(websocket.PingMessage, nil); err != nil {
                return
            }
        }
    }
}

func (gw *WebSocketGateway) handleClientMessage(client *Client, message map[string]interface{}) {
    messageType, ok := message["type"].(string)
```

```
    if !ok {
        return
    }

    switch messageType {
    case "join_room":
        room, ok := message["room"].(string)
        if ok {
            client.Rooms[room] = true
            gw.redis.SAdd(context.Background(), fmt.Sprintf("room:%s", room), client.UserID)
        }

    case "leave_room":
        room, ok := message["room"].(string)
        if ok {
            delete(client.Rooms, room)
            gw.redis.SRem(context.Background(), fmt.Sprintf("room:%s", room), client.UserID)
        }

    case "event_update":
        // Handle real-time event updates
        go gw.broadcastEventUpdate(message)

    case "typing":
        // Handle typing indicators
        go gw.handleTypingIndicator(client, message)
    }
}

// High-performance broadcasting to multiple clients
func (gw *WebSocketGateway) BroadcastToRoom(room string, message []byte) {
    // Get all clients in room concurrently
    roomClients := make([]*Client, 0)

    gw.clients.Range(func(key, value interface{}) bool {
        client := value.(*Client)
        if client.Rooms[room] {
            roomClients = append(roomClients, client)
        }
        return true
    })

    // Send to all clients concurrently
    var wg sync.WaitGroup
```

```
    for _, client := range roomClients {
        wg.Add(1)
        go func(c *Client) {
            defer wg.Done()
            select {
            case c.Send <- message:
            case <-time.After(time.Second):
                // Client not responding, close connection
                close(c.Send)
                gw.clients.Delete(c.ID)
            }
        }(client)
    }

    wg.Wait()
}

// NATS integration for cross-service real-time events
func (gw *WebSocketGateway) setupNATSSubscriptions() {
    // Subscribe to event updates
    gw.nats.Subscribe("events.updated", func(msg *nats.Msg) {
        var eventUpdate map[string]interface{}
        if err := json.Unmarshal(msg.Data, &eventUpdate); err != nil {
            return
        }

        eventID := eventUpdate["event_id"].(string)
        broadcastMsg, _ := json.Marshal(map[string]interface{}{
            "type": "event_updated",
            "data": eventUpdate,
        })

        // Broadcast to all clients interested in this event
        gw.BroadcastToRoom(fmt.Sprintf("event:%s", eventID), broadcastMsg)
    })

    // Subscribe to user notifications
    gw.nats.Subscribe("notifications.*", func(msg *nats.Msg) {
        var notification map[string]interface{}
        if err := json.Unmarshal(msg.Data, &notification); err != nil {
            return
        }

        userID := notification["user_id"].(string)
```

```
        broadcastMsg, _ := json.Marshal(map[string]interface{}{
            "type": "notification",
            "data": notification,
        })

        // Send to specific user
        gw.BroadcastToRoom(fmt.Sprintf("user:%s", userID), broadcastMsg)
    })
}
```

9. Social Service (Python)

Why Python: Complex graph algorithms, social network analysis, relationship processing

python

```python
# Advanced social networking service with graph algorithms
import asyncio
import networkx as nx
from neo4j import AsyncGraphDatabase
import aioredis
from typing import List, Dict, Set

class SocialGraphService:
    def __init__(self):
        self.neo4j = AsyncGraphDatabase.driver(
            "bolt://localhost:7687",
            auth=("neo4j", "password")
        )
        self.redis = aioredis.from_url("redis://localhost")
        self.graph_cache = {}

    async def build_social_graph(self, user_id: str, depth: int = 2) -> nx.Graph:
        """Build social graph using NetworkX for analysis"""

        cache_key = f"social_graph:{user_id}:{depth}"
        cached_graph = await self.redis.get(cache_key)

        if cached_graph:
            return nx.node_link_graph(json.loads(cached_graph))

        # Build graph from Neo4j data
        async with self.neo4j.session() as session:
            # Get friends and their relationships
            query = """
            MATCH (u:User {id: $user_id})-[:FRIENDS*1..2]-(friend:User)
            OPTIONAL MATCH (friend)-[r:FRIENDS]-(mutual:User)
            RETURN u, friend, mutual, r
            """

            result = await session.run(query, user_id=user_id)

            # Build NetworkX graph
            G = nx.Graph()
            async for record in result:
                user = record["u"]
                friend = record["friend"]
                mutual = record["mutual"]
```

```
            G.add_node(user["id"], **user)
            G.add_node(friend["id"], **friend)
            G.add_edge(user["id"], friend["id"])

            if mutual:
                G.add_node(mutual["id"], **mutual)
                G.add_edge(friend["id"], mutual["id"])

    # Cache the graph
    graph_data = nx.node_link_data(G)
    await self.redis.set(cache_key, json.dumps(graph_data), ex=3600)

    return G

async def find_mutual_friends(self, user1_id: str, user2_id: str) -> List[Dict]:
    """Find mutual friends between two users"""

    # Use concurrent queries for both users' friends
    friends1_task = self.get_user_friends(user1_id)
    friends2_task = self.get_user_friends(user2_id)

    friends1, friends2 = await asyncio.gather(friends1_task, friends2_task)

    # Find intersection
    friends1_ids = {f["id"] for f in friends1}
    friends2_ids = {f["id"] for f in friends2}
    mutual_ids = friends1_ids.intersection(friends2_ids)

    # Get detailed info for mutual friends
    mutual_friends = []
    for friend in friends1:
        if friend["id"] in mutual_ids:
            mutual_friends.append(friend)

    return mutual_friends

async def suggest_friends(self, user_id: str, limit: int = 10) -> List[Dict]:
    """Advanced friend suggestions using graph algorithms"""

    # Build social graph
    graph = await self.build_social_graph(user_id, depth=3)

    # Use multiple algorithms for friend suggestions
    suggestions = {}
```

```
    # 1. Friends of friends
    fof_suggestions = await self.friends_of_friends_suggestions(graph, user_id)
    for suggestion in fof_suggestions:
        suggestions[suggestion["id"]] = suggestion
        suggestions[suggestion["id"]]["score"] = suggestion.get("score", 0) + 0.3

    # 2. Common interests based suggestions
    interest_suggestions = await self.interest_based_suggestions(user_id)
    for suggestion in interest_suggestions:
        if suggestion["id"] in suggestions:
            suggestions[suggestion["id"]]["score"] += 0.2
        else:
            suggestions[suggestion["id"]] = suggestion
            suggestions[suggestion["id"]]["score"] = 0.2

    # 3. Location based suggestions
    location_suggestions = await self.location_based_suggestions(user_id)
    for suggestion in location_suggestions:
        if suggestion["id"] in suggestions:
            suggestions[suggestion["id"]]["score"] += 0.1
        else:
            suggestions[suggestion["id"]] = suggestion
            suggestions[suggestion["id"]]["score"] = 0.1

    # 4. Activity based suggestions (events attended together)
    activity_suggestions = await self.activity_based_suggestions(user_id)
    for suggestion in activity_suggestions:
        if suggestion["id"] in suggestions:
            suggestions[suggestion["id"]]["score"] += 0.4
        else:
            suggestions[suggestion["id"]] = suggestion
            suggestions[suggestion["id"]]["score"] = 0.4

    # Sort by score and return top suggestions
    sorted_suggestions = sorted(
        suggestions.values(),
        key=lambda x: x["score"],
        reverse=True
    )

    return sorted_suggestions[:limit]

async def friends_of_friends_suggestions(self, graph: nx.Graph, user_id: str) -> List[Dict]:
```

```
    """Find friends of friends who aren't already friends"""

    if user_id not in graph:
        return []

    user_friends = set(graph.neighbors(user_id))
    suggestions = {}

    # Look at friends of each friend
    for friend_id in user_friends:
        if friend_id not in graph:
            continue

        friends_of_friend = set(graph.neighbors(friend_id))

        # Potential suggestions are friends of friends who aren't already friends
        potential = friends_of_friend - user_friends - {user_id}

        for potential_friend in potential:
            if potential_friend not in suggestions:
                suggestions[potential_friend] = {
                    "id": potential_friend,
                    "mutual_friends": [],
                    "score": 0
                }

            # Add mutual friend
            suggestions[potential_friend]["mutual_friends"].append(friend_id)
            suggestions[potential_friend]["score"] += 1  # More mutual friends = higher score

    return list(suggestions.values())

async def analyze_social_influence(self, user_id: str) -> Dict:
    """Analyze user's social influence using graph centrality measures"""

    graph = await self.build_social_graph(user_id, depth=3)

    if user_id not in graph:
        return {"influence_score": 0, "metrics": {}}

    # Calculate various centrality measures
    try:
        betweenness = nx.betweenness_centrality(graph)
        closeness = nx.closeness_centrality(graph)
```

```
            eigenvector = nx.eigenvector_centrality(graph)
            degree = nx.degree_centrality(graph)

            user_metrics = {
                "betweenness_centrality": betweenness.get(user_id, 0),
                "closeness_centrality": closeness.get(user_id, 0),
                "eigenvector_centrality": eigenvector.get(user_id, 0),
                "degree_centrality": degree.get(user_id, 0),
                "total_connections": graph.degree(user_id)
            }

            # Calculate overall influence score
            influence_score = (
                user_metrics["betweenness_centrality"] * 0.3 +
                user_metrics["closeness_centrality"] * 0.2 +
                user_metrics["eigenvector_centrality"] * 0.3 +
                user_metrics["degree_centrality"] * 0.2
            )

            return {
                "influence_score": influence_score,
                "metrics": user_metrics,
                "rank_percentile": self.calculate_percentile_rank(influence_score, list(eigenvector.values()))
            }

        except Exception as e:
            logger.error(f"Social influence analysis failed: {e}")
            return {"influence_score": 0, "metrics": {}}

# Real-time social activity processing
class SocialActivityProcessor:
    def __init__(self):
        self.redis = aioredis.from_url("redis://localhost")
        self.activity_buffer = []
        self.buffer_size = 100

    async def process_social_activity(self, activity: Dict):
        """Process social activities in real-time"""

        # Add to buffer for batch processing
        self.activity_buffer.append(activity)

        # Process immediate notifications
        await self.send_immediate_notifications(activity)
```

```
        # Batch process when buffer is full
        if len(self.activity_buffer) >= self.buffer_size:
            await self.batch_process_activities()

    async def send_immediate_notifications(self, activity: Dict):
        """Send real-time notifications for social activities"""

        activity_type = activity.get("type")
        user_id = activity.get("user_id")

        # Get user's friends for notifications
        friends = await self.get_user_friends(user_id)

        notification_tasks = []

        for friend in friends:
            # Create personalized notification
            notification = {
                "type": "social_activity",
                "user_id": friend["id"],
                "actor_id": user_id,
                "activity": activity,
                "timestamp": datetime.utcnow().isoformat()
            }

            # Send notification asynchronously
            notification_tasks.append(
                self.send_notification(friend["id"], notification)
            )

        # Send all notifications concurrently
        await asyncio.gather(*notification_tasks, return_exceptions=True)

    async def batch_process_activities(self):
        """Batch process accumulated social activities"""

        activities = self.activity_buffer.copy()
        self.activity_buffer.clear()

        # Group activities by type for efficient processing
        grouped_activities = {}
        for activity in activities:
            activity_type = activity.get("type")
```

```
            if activity_type not in grouped_activities:
                grouped_activities[activity_type] = []
            grouped_activities[activity_type].append(activity)

        # Process each type of activity
        processing_tasks = []
        for activity_type, activity_list in grouped_activities.items():
            processing_tasks.append(
                self.process_activity_type(activity_type, activity_list)
            )

        await asyncio.gather(*processing_tasks, return_exceptions=True)
```

Inter-Service Communication

gRPC Services (Go ↔ Go)

```
go

// High-performance gRPC communication between Go services
service UserService {
    rpc GetUser(GetUserRequest) returns (GetUserResponse);
    rpc ValidateToken(ValidateTokenRequest) returns (ValidateTokenResponse);
    rpc UpdateUserPreferences(UpdatePreferencesRequest) returns (UpdatePreferencesResponse);
}

service EventService {
    rpc CreateEvent(CreateEventRequest) returns (CreateEventResponse);
    rpc GetEvents(GetEventsRequest) returns (stream EventResponse);
    rpc UpdateEventStatus(UpdateStatusRequest) returns (UpdateStatusResponse);
}
```

HTTP/REST + Message Queues (Python ↔ Go)

```
python
```

```python
# Python services communicate with Go services via HTTP and message queues
class ServiceCommunication:
    def __init__(self):
        self.http_client = aiohttp.ClientSession()
        self.nats_client = nats.connect("nats://localhost:4222")

    async def call_go_service(self, service_name: str, endpoint: str, data: dict):
        """Call Go service via HTTP"""
        url = f"http://{service_name}:8080{endpoint}"
        async with self.http_client.post(url, json=data) as response:
            return await response.json()

    async def publish_event(self, subject: str, data: dict):
        """Publish event to NATS for Go services"""
        await self.nats_client.publish(subject, json.dumps(data).encode())
```

Performance Optimizations

Go Services Optimizations

- **Connection Pooling**: PostgreSQL connection pools with pgxpool
- **Goroutine Pools**: Reuse goroutines for request handling
- **Memory Optimization**: Object pooling for frequently used structures
- **Caching**: Multi-level caching with Redis and in-memory caches

Python Services Optimizations

- **Async Everywhere**: AsyncIO for all I/O operations
- **Batch Processing**: Group operations to reduce database calls
- **ML Model Caching**: Cache model predictions and feature vectors
- **Vectorized Operations**: NumPy/Pandas for efficient data processing
- **Background Tasks**: Celery for heavy ML computations

Deployment & Infrastructure

Container Configuration

Go Services Dockerfile

```dockerfile
```

```
# Multi-stage build for Go services
FROM golang:1.21-alpine AS builder

WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download

COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /app/main .
CMD ["./main"]
```

Python Services Dockerfile

dockerfile

```
# Optimized Python container with ML libraries
FROM python:3.11-slim

# Install system dependencies for ML libraries
RUN apt-get update && apt-get install -y \
    gcc \
    g++ \
    libpq-dev \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

# Install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Kubernetes Deployment Strategy

yaml

```yaml
# Go Service Deployment (High Concurrency)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service-go
spec:
  replicas: 5
  selector:
    matchLabels:
      app: user-service-go
  template:
    metadata:
      labels:
        app: user-service-go
    spec:
      containers:
      - name: user-service
        image: events-platform/user-service:latest
        ports:
        - containerPort: 8080
        env:
        - name: GOMAXPROCS
          value: "4"
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: url
        resources:
          requests:
            memory: "128Mi"
            cpu: "100m"
          limits:
            memory: "256Mi"
            cpu: "500m"
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
          initialDelaySeconds: 10
          periodSeconds: 10
        readinessProbe:
          httpGet:
```

```yaml
          path: /ready
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 5

---
# Python Service Deployment (ML/AI Workloads)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: curation-service-python
spec:
  replicas: 3
  selector:
    matchLabels:
      app: curation-service-python
  template:
    metadata:
      labels:
        app: curation-service-python
    spec:
      containers:
      - name: curation-service
        image: events-platform/curation-service:latest
        ports:
        - containerPort: 8000
        env:
        - name: PYTHONUNBUFFERED
          value: "1"
        - name: WORKERS
          value: "4"
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "2Gi"
            cpu: "2"
        livenessProbe:
          httpGet:
            path: /health
            port: 8000
```

```
    initialDelaySeconds: 30
    periodSeconds: 15
```

Service Mesh Configuration (Istio)

yaml

```yaml
# Service mesh configuration for hybrid architecture
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: events-platform-routing
spec:
  http:
  # Route high-frequency requests to Go services
  - match:
    - uri:
        prefix: "/api/v1/users"
    - uri:
        prefix: "/api/v1/events"
    - uri:
        prefix: "/api/v1/search"
    route:
    - destination:
        host: go-services
        subset: high-performance
      weight: 100
    timeout: 5s
    retries:
      attempts: 3
      perTryTimeout: 2s

  # Route ML/AI requests to Python services
  - match:
    - uri:
        prefix: "/api/v1/recommendations"
    - uri:
        prefix: "/api/v1/curation"
    - uri:
        prefix: "/api/v1/analytics"
    route:
    - destination:
        host: python-services
        subset: ml-processing
      weight: 100
    timeout: 30s
    retries:
      attempts: 2
      perTryTimeout: 15s
```

```yaml
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: go-services
spec:
  host: go-services
  subsets:
  - name: high-performance
    labels:
      version: v1
    trafficPolicy:
      connectionPool:
        tcp:
          maxConnections: 100
        http:
          http1MaxPendingRequests: 50
          maxRequestsPerConnection: 10

---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: python-services
spec:
  host: python-services
  subsets:
  - name: ml-processing
    labels:
      version: v1
    trafficPolicy:
      connectionPool:
        tcp:
          maxConnections: 50
        http:
          http1MaxPendingRequests: 20
          maxRequestsPerConnection: 5
```

Message Queue Architecture

NATS Configuration (Go Services)

```go
```

```go
// NATS setup for Go services - lightweight and fast
func setupNATS() *nats.Conn {
    nc, err := nats.Connect("nats://localhost:4222",
        nats.MaxReconnects(10),
        nats.ReconnectWait(time.Second),
        nats.DisconnectErrHandler(func(nc *nats.Conn, err error) {
            log.Printf("NATS disconnected: %v", err)
        }),
        nats.ReconnectHandler(func(nc *nats.Conn) {
            log.Printf("NATS reconnected to %v", nc.ConnectedUrl())
        }),
    )
    if err != nil {
        log.Fatal(err)
    }
    return nc
}

// High-throughput event publishing
func (s *EventService) publishEventUpdate(event *Event) {
    eventData, _ := json.Marshal(map[string]interface{}{
        "type": "event.updated",
        "event_id": event.ID,
        "data": event,
        "timestamp": time.Now().Unix(),
    })

    // Publish to multiple subjects for fan-out
    subjects := []string{
        "events.updated",
        fmt.Sprintf("events.%s.updated", event.Category),
        fmt.Sprintf("users.%s.events", event.CreatorID),
    }

    for _, subject := range subjects {
        s.nats.Publish(subject, eventData)
    }
}
```

Kafka Integration (Python Services)

```python
```

```python
# Kafka setup for Python services - complex event processing
class KafkaEventProcessor:
    def __init__(self):
        self.producer = AIOKafkaProducer(
            bootstrap_servers='localhost:9092',
            value_serializer=lambda x: json.dumps(x).encode('utf-8'),
            compression_type='gzip'
        )
        self.consumer = AIOKafkaConsumer(
            'ml-events', 'analytics-events',
            bootstrap_servers='localhost:9092',
            group_id='python-services',
            value_deserializer=lambda m: json.loads(m.decode('utf-8'))
        )

    async def start(self):
        await self.producer.start()
        await self.consumer.start()

        # Start consumer loop
        asyncio.create_task(self.consume_events())

    async def consume_events(self):
        async for msg in self.consumer:
            try:
                await self.process_event(msg.value)
            except Exception as e:
                logger.error(f"Event processing failed: {e}")

    async def process_event(self, event_data: dict):
        event_type = event_data.get('type')

        if event_type == 'event.created':
            # Trigger ML curation pipeline
            await self.trigger_curation(event_data['event_id'])

        elif event_type == 'user.interaction':
            # Update recommendation models
            await self.update_user_features(event_data)

        elif event_type == 'event.completed':
            # Analyze event success metrics
            await self.analyze_event_performance(event_data['event_id'])
```

```
    async def publish_ml_result(self, topic: str, result: dict):
        """Publish ML processing results"""
        await self.producer.send(topic, result)
```

Database Architecture & Optimization

PostgreSQL Configuration

sql

```sql
-- Optimized PostgreSQL setup for hybrid architecture

-- Events table with partitioning
CREATE TABLE events (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    title VARCHAR(255) NOT NULL,
    description TEXT,
    creator_id UUID NOT NULL,
    category VARCHAR(50) NOT NULL,
    start_time TIMESTAMPTZ NOT NULL,
    end_time TIMESTAMPTZ NOT NULL,
    location JSONB,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
) PARTITION BY RANGE (start_time);

-- Create monthly partitions
CREATE TABLE events_2024_01 PARTITION OF events
    FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');
CREATE TABLE events_2024_02 PARTITION OF events
    FOR VALUES FROM ('2024-02-01') TO ('2024-03-01');
-- ... additional partitions

-- Optimized indexes
CREATE INDEX CONCURRENTLY idx_events_creator_time ON events (creator_id, start_time);
CREATE INDEX CONCURRENTLY idx_events_category_time ON events (category, start_time);
CREATE INDEX CONCURRENTLY idx_events_location_gin ON events USING GIN (location);
CREATE INDEX CONCURRENTLY idx_events_search ON events USING GIN (to_tsvector('english', title || ' ' || desc

-- Users table with proper indexing
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    profile JSONB,
    preferences JSONB,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX CONCURRENTLY idx_users_email ON users (email);
CREATE INDEX CONCURRENTLY idx_users_profile_gin ON users USING GIN (profile);
CREATE INDEX CONCURRENTLY idx_users_preferences_gin ON users USING GIN (preferences);
```

```
-- Connection pooling configuration
-- postgresql.conf optimizations:
-- max_connections = 200
-- shared_buffers = 256MB
-- effective_cache_size = 1GB
-- work_mem = 4MB
-- maintenance_work_mem = 64MB
-- checkpoint_completion_target = 0.9
-- wal_buffers = 16MB
-- default_statistics_target = 100
```

Redis Configuration

redis

```
# Redis configuration for caching and sessions
# redis.conf optimizations:

# Memory optimizations
maxmemory 2gb
maxmemory-policy allkeys-lru

# Persistence for critical data
save 900 1
save 300 10
save 60 10000

# Network optimizations
tcp-keepalive 300
timeout 0

# Performance tuning
databases 16
tcp-backlog 511
```

MongoDB Configuration (Event Metadata)

javascript

```
// MongoDB setup for flexible event metadata
db.createCollection("event_metadata", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["event_id", "metadata_type"],
      properties: {
        event_id: { bsonType: "string" },
        metadata_type: {
          enum: ["media", "social", "analytics", "ml_features"]
        },
        data: { bsonType: "object" },
        created_at: { bsonType: "date" },
        updated_at: { bsonType: "date" }
      }
    }
  }
});

// Optimized indexes
db.event_metadata.createIndex({ "event_id": 1, "metadata_type": 1 });
db.event_metadata.createIndex({ "created_at": 1 });
db.event_metadata.createIndex({ "data.category": 1, "data.location": "2dsphere" });
```

Monitoring & Observability

Prometheus Metrics (Go Services)

```
go
```

```go
// Custom metrics for Go services
var (
    httpRequestsTotal = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "http_requests_total",
            Help: "Total number of HTTP requests",
        },
        []string{"method", "endpoint", "status"},
    )

    httpRequestDuration = prometheus.NewHistogramVec(
        prometheus.HistogramOpts{
            Name: "http_request_duration_seconds",
            Help: "Duration of HTTP requests",
            Buckets: prometheus.DefBuckets,
        },
        []string{"method", "endpoint"},
    )

    databaseConnections = prometheus.NewGaugeVec(
        prometheus.GaugeOpts{
            Name: "database_connections_active",
            Help: "Number of active database connections",
        },
        []string{"database"},
    )

    goroutinesActive = prometheus.NewGauge(
        prometheus.GaugeOpts{
            Name: "goroutines_active",
            Help: "Number of active goroutines",
        },
    )
)

func init() {
    prometheus.MustRegister(httpRequestsTotal)
    prometheus.MustRegister(httpRequestDuration)
    prometheus.MustRegister(databaseConnections)
    prometheus.MustRegister(goroutinesActive)
}

// Middleware for automatic metrics collection
```

```
func metricsMiddleware() gin.HandlerFunc {
    return func(c *gin.Context) {
        start := time.Now()

        c.Next()

        duration := time.Since(start).Seconds()
        status := strconv.Itoa(c.Writer.Status())

        httpRequestsTotal.WithLabelValues(c.Request.Method, c.Request.URL.Path, status).Inc()
        httpRequestDuration.WithLabelValues(c.Request.Method, c.Request.URL.Path).Observe(duration)
    }
}
```

Application Monitoring (Python Services)

```
python
```

```python
# Comprehensive monitoring for Python services
import time
import psutil
from prometheus_client import Counter, Histogram, Gauge, generate_latest

# Custom metrics
ml_model_predictions = Counter('ml_model_predictions_total',
                               'Total ML model predictions',
                               ['model_name', 'status'])
ml_model_latency = Histogram('ml_model_prediction_seconds',
                             'ML model prediction latency',
                             ['model_name'])
ml_model_accuracy = Gauge('ml_model_accuracy',
                          'Current model accuracy',
                          ['model_name'])

# System metrics
memory_usage = Gauge('python_memory_usage_bytes', 'Memory usage in bytes')
cpu_usage = Gauge('python_cpu_usage_percent', 'CPU usage percentage')

class MetricsCollector:
    def __init__(self):
        self.start_time = time.time()

    async def collect_system_metrics(self):
        """Collect system-level metrics"""
        while True:
            try:
                # Memory usage
                memory_info = psutil.virtual_memory()
                memory_usage.set(memory_info.used)

                # CPU usage
                cpu_percent = psutil.cpu_percent(interval=1)
                cpu_usage.set(cpu_percent)

                await asyncio.sleep(30)  # Collect every 30 seconds

            except Exception as e:
                logger.error(f"Metrics collection failed: {e}")
                await asyncio.sleep(60)

    def track_ml_prediction(self, model_name: str):
```

```
    """Decorator to track ML predictions"""
    def decorator(func):
        async def wrapper(*args, **kwargs):
            start_time = time.time()
            try:
                result = await func(*args, **kwargs)
                ml_model_predictions.labels(model_name=model_name, status='success').inc()
                return result
            except Exception as e:
                ml_model_predictions.labels(model_name=model_name, status='error').inc()
                raise
            finally:
                latency = time.time() - start_time
                ml_model_latency.labels(model_name=model_name).observe(latency)
        return wrapper
    return decorator

# Health check endpoints
@app.get("/health")
async def health_check():
    return {"status": "healthy", "timestamp": datetime.utcnow().isoformat()}

@app.get("/metrics")
async def metrics():
    return Response(generate_latest(), media_type="text/plain")
```

Performance Benchmarks & Scaling

Expected Performance Characteristics

Go Services Performance

- **User Service**: 10,000+ concurrent authentications/second
- **Event Service**: 5,000+ event operations/second
- **Search Service**: 1,000+ complex queries/second
- **WebSocket Gateway**: 50,000+ concurrent connections
- **Memory Usage**: 50-200MB per service instance
- **Response Time**: <10ms for cached requests, <100ms for database queries

Python Services Performance

- **Curation Service**: 100+ AI analyses/second

- **Recommendation Engine**: 500+ recommendation requests/second
- **Analytics Service**: 1,000+ metric calculations/second
- **Memory Usage**: 500MB-2GB per service instance (due to ML models)
- **Response Time**: <500ms for simple ML tasks, <5s for complex analysis

Auto-scaling Configuration

yaml

```yaml
# Horizontal Pod Autoscaler for Go services
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: user-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: user-service-go
  minReplicas: 3
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80

---
# HPA for Python services (different scaling characteristics)
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: curation-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: curation-service-python
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
```

```
    target:
      type: Utilization
      averageUtilization: 60  # Lower threshold due to ML workloads
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 75
```

This hybrid Python/Go architecture maximizes the strengths of both languages:

- **Go services** handle high-concurrency, low-latency operations with excellent resource efficiency
- **Python services** leverage the rich ML/AI ecosystem for complex business logic and data processing
- **Message queues** provide loose coupling and scalability between services
- **Optimized databases** support both transactional integrity and analytical workloads
- **Comprehensive monitoring** ensures system health and performance visibility

The architecture can scale from handling thousands to millions of users while maintaining sub-second response times for most operations.recommendations

```
async def get_collaborative_recommendations(self, user_id: str,
                                            user_history: List[dict],
                                            limit: int) -> List[dict]:
    """Collaborative filtering with async matrix operations"""

    # Use thread pool for CPU-intensive computations
    loop = asyncio.get_event_loop()

    # Find similar users based on event attendance patterns
    similar_users = await loop.run_in_executor(
        self.executor,
        self.collaborative_model.find_similar_users,
        user_id, user_history
    )

    # Get events liked by similar users
    candidate_events = []
    for similar_user_id, similarity_score in similar_users:
        similar_user_events = await self.get_user_liked_events(similar_user_id)

        for event in similar_user_events:
            if event["id"] not in [h["event_id"] for h in user_history]:
                event["collab_score"] = similarity_score
                candidate_events.append(event)

    # Score and rank events
    scored_events = await loop.run_in_executor(
        self.executor,
        self.collaborative_model.score_events,
        candidate_events, user_history
    )

    return scored_events[:limit]

async def get_content_based_recommendations(self, user_data: dict,
                                            events_data: List[dict],
                                            limit: int) -> List[dict]:
    """Content-based filtering with TF-IDF and feature matching"""

    loop = asyncio.get_event_loop()

    # Extract user preferences
    user_interests = user_data.get("interests", [])
```

```
user_categories = user_data.get("preferred_categories", [])
user_location = user_data.get("location", {})

# Create user profile vector
user_profile = await loop.run_in_executor(
    self.executor,
    self.content_model.create_user_profile,
    user_interests, user_categories
)

# Vectorize event descriptions and features
event_vectors = await loop.run_in_executor(
    self.executor,
    self.content_model.vectorize_events,
    events_data
)

# Calculate similarity scores
similarity_scores = await loop.run_in_executor(
    self.executor,
    cosine_similarity,
    [user_profile], event_vectors
)

# Add location and time preferences
enhanced_scores = []
for i, (event, score) in enumerate(zip(events_data, similarity_scores[0])):
    # Calculate distance penalty
    distance_score = self.calculate_distance_score(
        user_location, event.get("location", {})
    )

    # Time preference score
    time_score = self.calculate_time_preference_score(
        user_data.get("preferred_times", []),
        event.get("start_time")
    )

    final_score = score * 0.6 + distance_score * 0.2 + time_score * 0.2
    enhanced_scores.append((event, final_score))

# Sort by score and return top recommendations
enhanced_scores.sort(key=lambda x: x[1], reverse=True)
return [{"event": event, "content_score": score}
```

```
        for event, score in enhanced_scores[:limit]]

async def get_deep_learning_recommendations(self, user_id: str, limit: int) -> List[dict]:
    """Neural network-based recommendations"""

    # Prepare input features for the deep model
    user_features = await self.prepare_user_features(user_id)
    event_features = await self.prepare_event_features()

    loop = asyncio.get_event_loop()

    # Run neural network inference
    predictions = await loop.run_in_executor(
        self.executor,
        self.deep_model.predict,
        user_features, event_features
    )

    # Convert predictions to recommendations
    recommendations = []
    for event_id, score in predictions:
        event_data = await self.get_event_by_id(event_id)
        recommendations.append({
            "event": event_data,
            "deep_score": score
        })

    return sorted(recommendations, key=lambda x: x["deep_score"], reverse=True)[:limit]

async def hybrid_ranking(self, collab_recs: List[dict],
                         content_recs: List[dict],
                         deep_recs: List[dict],
                         trending_recs: List[dict],
                         user_data: dict,
                         limit: int) -> List[dict]:
    """Combine multiple recommendation sources with weighted scoring"""

    # Create unified event scoring
    event_scores = {}

    # Weight different recommendation sources
    weights = {
        "collaborative": 0.35,
        "content": 0.25,
```

```
    "deep": 0.25,
    "trending": 0.15
}

# Aggregate scores from all sources
for recs, score_key, weight in [
    (collab_recs, "collab_score", weights["collaborative"]),
    (content_recs, "content_score", weights["content"]),
    (deep_recs, "deep_score", weights["deep"]),
    (trending_recs, "trending_score", weights["trending"])
]:
    for rec in recs:
        event_id = rec["event"]["id"]
        score = rec.get(score_key, 0) * weight

        if event_id in event_scores:
            event_scores[event_id]["total_score"] += score
            event_scores[event_id]["sources"].append(score_key)
        else:
            event_scores[event_id] = {
                "event": rec["event"],
                "total_score": score,
                "sources": [score_key]
            }

# Apply diversity and business rules
final_recommendations = await self.apply_diversity_optimization(
    event_scores, user_data, limit
)

return final_
```