Real-time NBA Player, Game, and Season Analytics

Ben Aoki-Sherwood* Arjun Rao†

1 Project Goals

We will build a scalable, real-time web application that predicts NBA team rankings and playoff probabilities using statistical and machine learning models. The system will:

1. **Real-time data ingestion and updates**: Periodically collect current NBA game results and team and player statistics from the ESPN API and an NBA.com client API, and re-run analytics based on these live data.
2. **Player-, team- and season-level stats prediction**: Use deep learning (LSTM) and mathematical (SpringRank) models trained on recent historical data to predict:
   - Game outcomes
   - Player outcomes:
     - Box scores
     - Playing time
     - Injury
   - Season stats:
     - End-of-season team rankings for each conference
     - Playoff qualification probabilities for each team
     - Expected win-loss records based on current rankings and team schedules
3. **In-game event prediction**: use Poisson processes to predict:
   - Real-time win probabilities during live games
   - Player scoring rates and game milestone probabilities (e.g., "Will Player X score 25+ points tonight?")
   - Event likelihoods (3PM, rebounds, blocks, steals, turnovers, fouls)
   - Expected scoring in specific time intervals
   - Scoring runs (what is the probability that Team A goes on a 10-0 run?)
   - Non-homogeneous Poisson processes to model momentum and clutch performance
4. **Dashboard**: Dashboard interface allows users to view current rankings, game event, score, and season projections, and historical trends with minimal latency

*https://github.com/aoki-sherwoodb
†https://github.com/arjunarao619

2 Software and Hardware Components

2.1 Cloud Platform: GCP

2.2 Data Layer

1. **Cloud SQL (PostgreSQL)** - Relational database for:
   - Historical game results and statistics (2010-present)
   - Team and player information
   - Training dataset storage
   - User query logs
2. **Google Cloud Storage (GCS)** - Object storage for:
   - Trained ML model weights and checkpoints
   - Large datasets and CSV exports
   - Backup and archival data
3. **Memorystore (Redis)** - In-memory cache for:
   - Current team rankings and standings
   - Recent API responses
   - Frequently accessed predictions
   - Session management

2.3 Compute Layer

4. **Google Kubernetes Engine (GKE)** - Container orchestration hosting:
   - Data Ingestion Service (Python FastAPI)
   - ML Training/Inference Service (Python, scikit-learn/TensorFlow)
   - Poisson Event Modeling Service (Python, scipy)
   - API Gateway Service (Node.js/Python)
   - Web Frontend Service (React.js served via nginx)
5. **Docker** - Containerization of all microservices

2.4 Message Queue Layer

6. **Cloud Pub/Sub** - Message queue for:
   - Distributing incoming game results to processing services
   - Triggering model retraining jobs
   - Coordinating between microservices

2.5 APIs

7. **REST APIs** - For:
    - External data ingestion (ESPN, NBA Stats)
    - Internal microservice communication
    - Frontend-backend communication
    - Public API endpoints for client applications

2.6 ML

8. **Python 3.10+** with libraries:
    - scikit-learn (baseline models: Random Forest, Logistic Regression)
    - TensorFlow/Keras (optional neural network models)
    - scipy.stats (Poisson distributions and statistical modeling)
    - numpy (numerical computations and Monte Carlo simulations)
    - pandas (data processing)
    - requests (API calls)

2.7 Frontend

9. **React.js** - Single-page application for user dashboard
10. **Chart.js/D3.js** - Data visualization libraries
11. **Material-UI** - Component library

2.8 Monitoring and Logging

12. **Cloud Logging** - Centralized logging
13. **Cloud Monitoring** - Performance metrics and alerting

2.9 Hardware Resources (GCP)

- **GKE Cluster**: 3-5 n1-standard-2 nodes (2 vCPUs, 7.5GB RAM each)
- **Cloud SQL Instance**: db-n1-standard-1 (1 vCPU, 3.75GB RAM)
- **Memorystore Redis**: 1GB Basic Tier instance

3 Architecture

4 Microservices and Interactions

4.1 Data Flow and Component Interactions

4.1.1 Data Ingestion

1. **Data Ingestion service** runs on a scheduled basis (e.g., every 30 minutes during the NBA season)
2. It makes REST API calls to external sources (ESPN API, NBA Stats API) to fetch:

- Recent game results
- Updated team standings
- Player performance statistics

3. The service validates and transforms the raw data into a standardized format
4. Validated data is published to **Cloud Pub/Sub** topic `new-game-results`
5. The ingestion service also writes raw data to **Cloud SQL** for historical record-keeping

4.1.2 Model Training and Inference

1. **ML Training/Inference service** subscribes to the `new-game-results` topic
2. When new game data arrives:
   - The service updates its training dataset in **Cloud SQL**
   - If enough new data has accumulated (e.g., 1-4 new games or daily on days when games are played), trigger a retraining job
3. Model training process:
   - Fetches historical data from **Cloud SQL** (last 5 seasons)
   - Trains ensemble models (Random Forest, Logistic Regression)
   - Validates model performance using cross-validation
   - Saves model weights and metadata to **Google Cloud Storage**
4. After training, the service generates fresh predictions for all teams
5. Predictions are written to both:
   - **Cloud SQL** (persistent storage with timestamps)
   - **Memorystore Redis** (fast cache with 1-hour TTL)

4.1.3 Poisson Process Modeling

1. **Poisson Event Modeling service** subscribes to both `new-game-results` and `live-game-events` topics
2. **Rate parameter estimation**:
   - Service queries **Cloud SQL** for play-by-play data
   - Calculates scoring rates for teams/players:
     - Points per minute (overall and situational)
     - Three-point attempt rates
     - Turnover rates
     - Foul rates
   - Adjusts rates based on:
     - Opponent defensive ratings
     - Home/away status

- Rest days
- Recent form (last 5 games)

- Stores computed rate parameters in **Redis** with 24-hour TTL

3. **Live game predictions**:
   - When a live game event occurs (score update, timeout, etc.):
     - Service fetches current game state (score, time remaining, possessions)
     - Retrieves team λ parameters from **Redis**
     - Runs Monte Carlo simulations (10,000 iterations):
       * Simulates remaining game using dual Poisson processes
       * One process for each team
       * Accounts for time remaining and possession count
     - Calculates win probability, expected final score, and confidence intervals
     - Publishes results to `prediction-requests` topic
     - Caches live probabilities in **Redis** (1-minute TTL for rapidly changing values)
4. **Player event predictions**:
   - Calculates probabilities for player milestones:
     - "Will Player X score 25+ points tonight?"
     - "Probability of 5+ three-pointers made"
     - "Risk of fouling out"
   - Uses player-specific rate parameters adjusted for matchup difficulty
   - Results stored in **Redis** and **Cloud SQL**
5. **Model persistence**:
   - Fitted Poisson rate parameters saved to **GCS** daily
   - Historical accuracy metrics logged to **Cloud SQL**
   - Enables backtesting and model improvement

4.1.4 API Requests

1. User visits the web application frontend
2. **Frontend (React)** sends HTTPS requests to the **API Gateway Service**
3. API Gateway checks authentication/authorization
4. For season-end prediction requests:
   - Gateway first checks **Redis cache** for recent predictions
   - If cache hit: Returns cached data immediately
   - If cache miss: Queries **Cloud SQL** for latest predictions from ML Training Service
   - Query results are cached in Redis for future requests

5. For live game probability requests:
    - Gateway checks **Redis** for most recent Poisson-based win probabilities
    - If not cached or stale, sends request to **Poisson Event Modeling Service** via Pub/Sub
    - Service computes fresh probability using Monte Carlo simulation and caches result
    - Returns win probability, expected score, and confidence intervals
6. For player event predictions:
    - Gateway queries cached Poisson rate parameters from **Redis**
    - If cache miss, **Poisson Service** computes rates from play-by-play data in **Cloud SQL**
    - Returns probabilities for requested milestones (e.g., "25+ points tonight")
7. For historical data requests:
    - Gateway queries **Cloud SQL** directly
    - Results may be cached based on query patterns
8. Responses are sent back to the frontend as JSON

4.1.5 Real-time Updates

1. When critical events occur (playoff scenarios change, major upsets):
    - Ingestion Service publishes to `model-retrain-trigger` topic
    - ML Service receives trigger and prioritizes immediate retraining
    - Once new predictions are ready, a notification is published
    - Frontend can poll for updates or use WebSocket connections for live updates
2. During live games:
    - Ingestion Service publishes to `live-game-events` topic every 30-60 seconds
    - **Poisson Event Modeling Service** immediately recalculates win probabilities
    - Updated probabilities pushed to **Redis**
    - Frontend displays real-time probability charts that update as game progresses

4.1.6 Containerized Deployment

1. All services are packaged as **Docker containers**
2. **GKE** manages container lifecycle:
    - Auto-scaling based on CPU/memory usage
    - Load balancing across multiple pod replicas
    - Health checks and automatic restarts
    - Rolling updates for zero-downtime deployments
3. Services communicate within the cluster using Kubernetes service discovery

4.1.7 Message Queue Coordination

Cloud Pub/Sub decouples services and enables:

- Asynchronous processing (ingestion doesn't wait for training)
- Fault tolerance (messages are persisted if subscribers are down)
- Scalability (multiple service instances can subscribe to same topic)
- Priority handling (urgent retraining can use separate topic)

5 Debugging and Testing

5.1 Unit Testing

- **Component-level tests** for each microservice
- **Framework**: pytest for Python services, Jest for React frontend
- **Coverage targets**: ¿80% code coverage
- **Testing approach**:
  - Static external API calls using historical games from ESPN API
  - Verify ML model input/output shapes
  - Test cache hit/miss scenarios

5.2 Integration Testing

- **Service-to-service communication tests**
- **Pub/Sub message flow verification**:
  - Publish test messages and verify subscribers process them correctly
  - Test message acknowledgment and retry logic
- **Database interaction tests**:
  - Test CRUD operations against test PostgreSQL database
  - Verify transaction handling and rollback scenarios
- **Cache integration tests**:
  - Verify Redis caching behavior
  - Test cache invalidation logic

5.3 End-to-End Testing

- **Full pipeline tests**:
  - Simulate game data ingestion to model training to prediction generation to API response pipeline
  - Use staging environment with production-like configuration
- **Load testing**:
  - Apache JMeter or Locust to simulate concurrent users

  - Test API Gateway under load (100+ requests/second)
  - Verify auto-scaling behavior in GKE

5.4 Model Validation and Testing

ML Models:

- Training data validation:
  - Check for data quality issues (missing values, clip outlier statistics)
  - Verify temporal consistency (no data leakage)
- Model performance metrics:
  - Accuracy, precision, recall for playoff predictions
  - Mean Absolute Error (MAE) for win-loss projections
  - Backtesting on previous seasons (2022-2024)
- A/B testing framework:
  - Train and deploy several model architectures and compare performance on both historical and current season data

Poisson process model development and validation:

- **Rate parameter validation**:
  - Cross-validation: Train on games 1-70, test on games 71-82 of a season
  - Verify rate parameter estimates stabilize after enough games
  - Compare estimated rates to actual observed rates using Chi-square goodness-of-fit tests
- **Win probability accuracy**:
  - Calibration plots: Are predicted 70% win probabilities actually winning 70% of the time?
  - Brier score: Measure accuracy of probabilistic predictions
  - Compare to baseline models (Vegas odds, SpringRank)
  - Track accuracy across different game situations (close games, blowouts, clutch time)
- **Monte Carlo simulation validation**:
  - Verify convergence: Results stable with 10,000 simulations
  - Compare Monte Carlo to analytical solutions for simple cases
  - Test computational performance (simulations complete in $\sim$real-time)
- **Player event prediction validation**:
  - Backtest player milestone predictions on historical games
  - Calculate ROI if used for betting (theoretical analysis only)
  - Verify confidence intervals contain true outcomes at expected rates
- **Assumption checks**:

  - Test for independence: Are consecutive baskets independent? (probably not!)
  - Test for constant rate: Do rates change during game? (we expect difference for 1st quarter vs 4th)
  - Implement piecewise Poisson models if needed (different λ for different game periods)
- **Edge cases**:
  - Extreme scores (50-point leads, overtime games)
  - Unusual rates (defensive battles, high-scoring shootouts)

5.5 Debugging Tools and Techniques

- **Cloud logging**:
  - Structured logging with correlation IDs across services
  - Log levels: DEBUG for development, INFO for production
  - Searchable logs aggregated in Google Cloud Console
- **Cloud monitoring**:
  - Custom metrics for:
    * API response times
    * ML model training duration
    * Poisson simulation computation time
    * Prediction accuracy over time (both ML and Poisson models)
    * Cache hit rates (especially for Poisson rate parameters)
    * Live game probability update frequency
  - Alerts for:
    * Error rate spikes
    * API latency ¿ 500ms
    * Failed model training jobs
    * Poisson simulations taking ¿200ms
    * Win probability calculation errors
    * Unusual rate parameter values (potential data quality issues)
- **Distributed tracing**:
  - Cloud Trace to track request flow across microservices
  - Identify bottlenecks in the request pipeline
- **Local development**:
  - Docker Compose for running full stack locally
  - Minikube for local Kubernetes testing
  - Environment parity (dev/staging/prod)

5.6 Data Quality Monitoring

- **Ingestion monitoring**:
  - Track API availability and response times
  - Alert on data schema changes
  - Validate data completeness (all expected games ingested)
- **Anomaly detection**:
  - Flag unusual prediction patterns
  - Detect model drift (degrading accuracy over time)

6 Cloud Technologies Used

6.1 Message Queues

- **Technology Used**: Google Cloud Pub/Sub
- **For**: Asynchronous message passing between Data Ingestion Service, ML Training Service, Poisson Process Service, and API Gateway

6.2 Databases

- **Technology Used**: Cloud SQL (PostgreSQL)
- **For**: Persistent storage of historical game data, team statistics, player information, and prediction history

6.3 Key-Value Stores

- **Technology Used**: Memorystore (Redis)
- **For**: High-speed caching of current rankings, recent predictions, and API responses

6.4 Containerization

- **Technology Used**: Docker containers orchestrated by Google Kubernetes Engine (GKE)
- **For**: Containerization and orchestration of all microservices (ML inference/training, Poisson process modeling, API gateway)

6.5 Storage Services

- **Technology Used**: Google Cloud Storage (GCS)
- **For**: Object storage for trained ML model weights, checkpoints, and large datasets

6.6 RPC / API Interfaces

- **Technology Used**: REST APIs (internal and external)
- **For**: Communication between frontend and backend, between microservices, and with external data providers

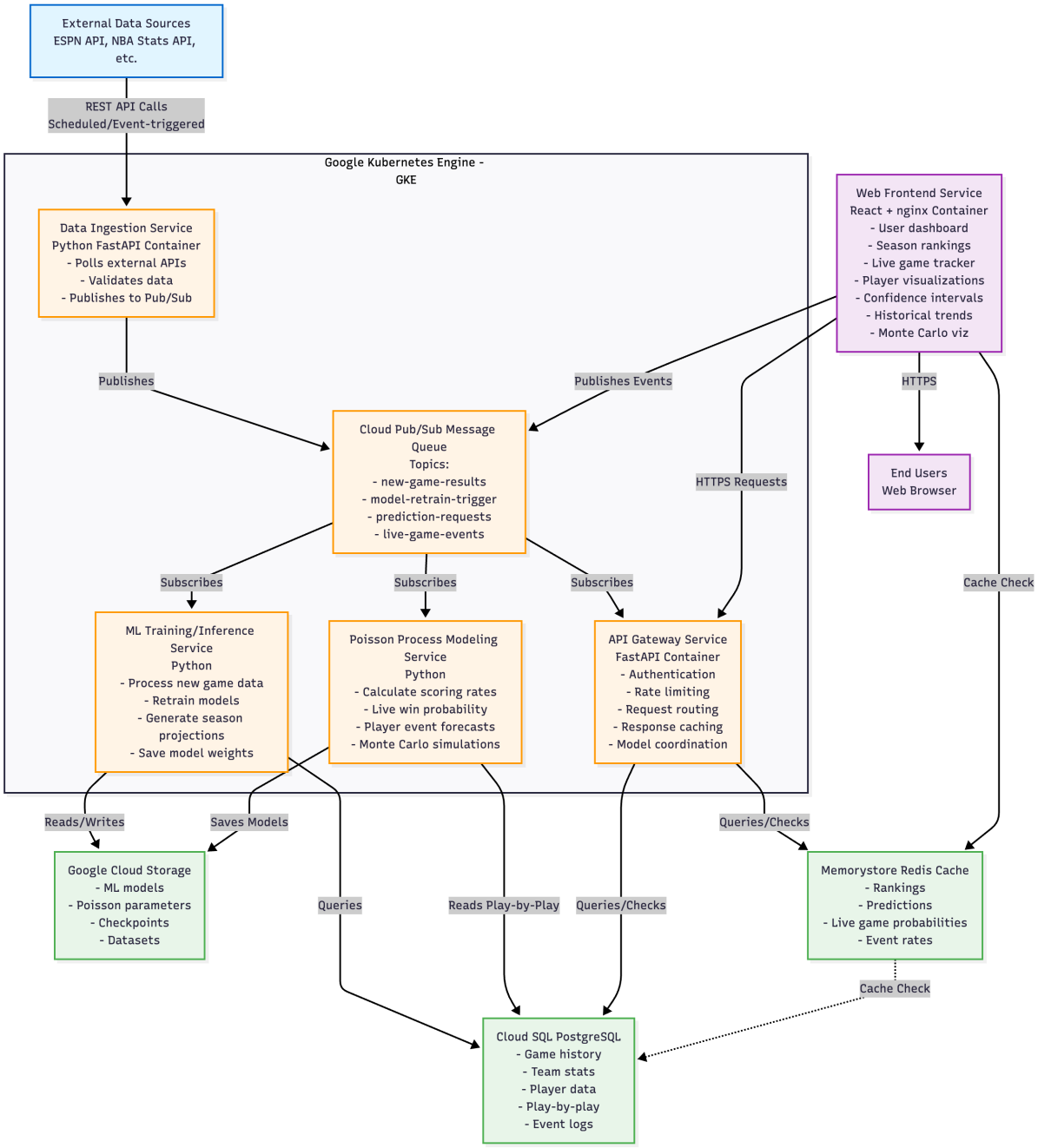

Figure 1: System Architecture Diagram