Cross-Service Communication Plan for AI-Based HR & Payroll System

REST-Based Communication (Current Phase)

Service Architecture and Responsibilities

The current system architecture employs a REST-based communication pattern between core microservices, establishing a synchronous request-response model that facilitates immediate data consistency and straightforward debugging during the MVP phase. The primary services include the Employee Service, which manages employee lifecycle operations and maintains master data; the Payroll Service, responsible for salary calculations, tax computations, and payment processing; the Leave Service, handling vacation requests, approvals, and balance tracking; the Authentication Service, managing user credentials, JWT token generation, and authorization policies; and the AI Service, providing intelligent recommendations for compensation, performance analytics, and predictive insights.

Each service maintains clear boundaries and responsibilities, with the Employee Service serving as the foundational data layer containing employee profiles, organizational hierarchy, and employment contracts. The Payroll Service orchestrates complex financial calculations by consuming employee data through REST endpoints, while the Leave Service maintains its own state for leave balances and approval workflows. The Authentication Service acts as a centralized security gateway, issuing JWT tokens that carry user identity and role information across service boundaries.

JWT Authentication and Security Model

The authentication mechanism relies on JSON Web Tokens (JWT) implementing a bearer token strategy where each service validates incoming requests by verifying the JWT signature against a shared secret or public key. The Authentication Service generates tokens containing user identity, roles, permissions, and expiration timestamps, ensuring that cross-service communications maintain security context without requiring database lookups on every request. Services implement middleware that intercepts incoming requests, extracts the JWT from the Authorization header, validates the token's integrity, and populates the request context with user information.

Token refresh mechanisms ensure continuous authentication without frequent user login prompts, while role-based access control (RBAC) is enforced at the service level by examining JWT claims. The system employs short-lived access tokens (15 minutes) paired with longer-lived refresh tokens (7 days) to balance security with user experience. Service-to-service communication utilizes machine-to-machine tokens with extended validity periods and service-specific scopes, enabling automated processes like payroll generation to access multiple services without human intervention.

Endpoint Contracts and FastAPI Implementation

Service contracts define explicit API boundaries using OpenAPI specifications automatically generated by FastAPI decorators and Pydantic models. The Employee Service exposes endpoints such as `GET /api/v1/employees/{employee_id}` returning comprehensive employee profiles, `POST /api/v1/employees` for creating new employee records, and `PUT /api/v1/employees/{employee_id}` for updates. Response models utilize Pydantic schemas

ensuring type safety and automatic validation, with example contracts including employee creation requests containing personal information, job details, and organizational assignments.

The Payroll Service implements endpoints like POST `/api/v1/payroll/calculate` accepting employee IDs and pay period parameters, returning detailed salary breakdowns including base pay, allowances, deductions, and tax calculations. Integration endpoints such as GET `/api/v1/payroll/history/{employee_id}` provide historical payroll data for reporting and compliance purposes. The Leave Service offers POST `/api/v1/leave/request` for submission workflows, PUT `/api/v1/leave/{request_id}/approve` for management actions, and GET `/api/v1/leave/balance/{employee_id}` for current leave balances.

FastAPI's dependency injection system facilitates clean separation of concerns, with database connections, authentication validation, and business logic components injected into endpoint handlers. The framework's automatic documentation generation creates interactive API documentation accessible through Swagger UI, enabling developers to test endpoints and understand service contracts without external documentation. Response caching leverages FastAPI's background tasks for expensive operations, while request validation ensures data integrity before processing.

Advantages and Limitations of REST in MVP Phase

REST-based communication provides immediate advantages during the MVP phase, including simplified debugging through direct HTTP request inspection, predictable request-response patterns that developers understand intuitively, and automatic API documentation generation through FastAPI. The synchronous nature of REST calls ensures immediate consistency, making it easier to implement complex business logic that requires real-time data validation across services. Error handling follows standard HTTP status codes, providing clear feedback for client applications and facilitating troubleshooting.

However, REST communication introduces coupling between services, as calling services must wait for responses before proceeding, potentially creating cascading failures when dependent services experience latency or outages. The synchronous model limits scalability when handling high-volume operations like bulk payroll processing, where sequential API calls can create bottlenecks. Network latency accumulates across multiple service calls, and the lack of native retry mechanisms requires custom implementation for handling transient failures. Additionally, REST's stateless nature necessitates repeated authentication and authorization checks, adding overhead to each request.

Part 2: Message Bus Architecture (Future Phase)

Asynchronous Communication Strategy

The future architecture transitions to an event-driven messaging system utilizing Apache Kafka as the primary message broker, enabling asynchronous communication patterns that decouple services and improve system resilience. This architectural shift supports horizontal scaling by allowing services to process events independently, reducing direct dependencies and enabling fault tolerance through persistent message queues. The message bus architecture facilitates complex workflows where multiple services react to single events, such as employee onboarding triggering actions across payroll setup, access provisioning, and training enrollment.

Kafka's distributed architecture provides high availability and fault tolerance through topic partitioning and replication, ensuring message delivery even during broker failures. The system

implements separate Kafka clusters for different environments (development, staging, production) with appropriate retention policies and partition strategies based on message volume and processing requirements. Consumer groups enable parallel processing of messages while maintaining ordering within partitions, crucial for financial operations where transaction sequence matters.

Domain Events and Service Interactions

The event-driven architecture defines specific domain events that represent meaningful business occurrences, with each event carrying sufficient context for consuming services to process independently. The employee_created event includes complete employee profile data, organizational assignments, and metadata required for downstream processing, published by the Employee Service when new employees are onboarded. This event triggers the Payroll Service to establish salary structures, the Leave Service to initialize leave balances, and the AI Service to begin building employee performance baselines.

The leave_approved event contains leave request details, approval workflow information, and employee context, enabling the Payroll Service to adjust salary calculations for unpaid leave and the AI Service to update attendance analytics. The payroll_generated event includes comprehensive payroll information, tax calculations, and payment instructions, triggering the Finance Service to initiate bank transfers and the Reporting Service to update financial dashboards. Additional events like employee_terminated, salary_updated, and performance_reviewed create a comprehensive audit trail while enabling real-time system updates.

Message Payload Structure and Schema Management

Event payloads follow standardized JSON Schema definitions ensuring consistency across service boundaries and enabling automated validation. The employee_created event structure includes header information with event metadata such as event ID, timestamp, correlation ID, and source service, alongside payload data containing employee details, employment information, and organizational context. Schema versioning strategies accommodate evolving business requirements while maintaining backward compatibility, utilizing semantic versioning and schema registry integration.

```
{
  "header": {
    "event_id": "uuid-v4",
    "event_type": "employee_created",
    "timestamp": "2025-06-11T10:30:00Z",
    "correlation_id": "uuid-v4",
    "source_service": "employee-service",
    "schema_version": "1.2.0"
  },
```

```
"payload": {
    "employee_id": "EMP-2025-001",
    "personal_info": {
      "first_name": "John",
      "last_name": "Doe",
      "email": "john.doe@company.com",
      "phone": "+91-9876543210"
    },
    "employment_details": {
      "department": "Engineering",
      "position": "Senior Software Engineer",
      "start_date": "2025-06-15",
      "salary": 1200000,
      "employment_type": "full_time"
    },
    "organizational_context": {
      "manager_id": "EMP-2024-156",
      "location": "Bangalore",
      "cost_center": "ENG-001"
    }
  }
}
```

The schema registry maintains version history and compatibility rules, enabling producers to publish events using current schemas while consumers can process events using compatible older versions. This approach supports rolling deployments where services are updated incrementally without requiring simultaneous updates across all components.

Broker Configuration and Reliability Mechanisms

Kafka broker configuration emphasizes durability and performance through appropriate replication factors, acknowledgment settings, and retention policies. Topics utilize a minimum replication factor of three across different availability zones, ensuring message persistence during individual broker failures. Producer configurations specify acknowledgment requirements (acks=all) ensuring messages are committed to all replicas before considering sends successful, while idempotent producers prevent duplicate messages during retry scenarios.

Consumer configurations implement automatic offset management with periodic commits, enabling exactly-once processing semantics for critical financial operations. Dead Letter Queue (DLQ) mechanisms handle messages that consistently fail processing, routing them to separate topics for manual investigation and reprocessing. The system implements exponential backoff retry strategies with configurable maximum retry attempts, allowing transient failures to resolve automatically while preventing infinite retry loops.

Message retention policies balance storage costs with operational requirements, maintaining financial events for extended periods while applying shorter retention for operational events. Partition strategies consider message keys and processing requirements, ensuring related events are processed in order while enabling parallel processing of independent operations.

Integration with Celery and FastAPI Background Tasks

During the transition period, the system leverages Celery for complex background processing while maintaining FastAPI background tasks for lightweight operations. Celery workers consume Kafka messages for compute-intensive tasks like payroll calculation, AI model training, and bulk data processing, providing distributed task execution with Redis or RabbitMQ as the message broker. This hybrid approach enables gradual migration from synchronous to asynchronous patterns while maintaining system stability.

FastAPI background tasks handle immediate response requirements where clients need acknowledgment of successful event publication without waiting for complete processing. The system publishes events to Kafka topics from background tasks, enabling quick API responses while ensuring reliable event delivery. Celery workers subscribe to Kafka topics for long-running processes, providing scalable processing capabilities that can be adjusted based on workload demands.

Part 3: External APIs and Third-Party Integrations

Salary Benchmarking and Market Data Integration

The AI Service integrates with Glassdoor's API to fetch real-time salary benchmarking data, enabling intelligent compensation recommendations based on market trends, location factors, and role specifications. This integration involves authenticated API calls using Glassdoor's partner credentials, with requests specifying job titles, experience levels, and geographic locations to retrieve relevant salary ranges. The system processes API responses to extract statistical data including median salaries, percentile distributions, and market growth trends, storing this information in a dedicated cache layer for rapid access during compensation analysis.

The integration implements sophisticated caching strategies with TTL (Time To Live) configurations varying based on data volatility, maintaining fresh market data while respecting API rate limits. The system validates incoming salary data against predefined ranges and statistical models to identify anomalies or outdated information, ensuring recommendation accuracy. Error handling mechanisms provide fallback compensation suggestions using historical data when external APIs are unavailable, maintaining system functionality during third-party service disruptions.

Learning Management System Integration

The Employee Service integrates with Coursera's API to synchronize employee learning progress, course completions, and skill development tracking. This integration enables automated skill gap analysis by comparing employee competencies against role requirements and industry standards.

The system authenticates with Coursera using OAuth 2.0 credentials, managing token refresh cycles and scope-based access to employee learning data. API calls retrieve course enrollment status, progress tracking, and completion certificates, updating employee profiles with verified skill credentials.

The integration implements batch processing for large-scale data synchronization, processing course completion updates during off-peak hours to minimize API call frequency. Data validation ensures course information accuracy and prevents duplicate skill entries, while privacy controls ensure employee learning data is handled according to organizational policies. The system generates learning analytics reports combining Coursera data with internal performance metrics, providing insights for talent development initiatives.

Financial Technology and Payment Processing

The Payroll Service integrates with Razorpay's API suite for comprehensive payment processing, including salary disbursements, tax payments, and vendor settlements. This integration encompasses multiple API endpoints for payment creation, status tracking, and transaction reconciliation, utilizing Razorpay's webhook notifications for real-time payment updates. The system maintains secure credential management using encrypted configuration files and environment variables, implementing proper secret rotation policies for API keys and webhook secrets.

Payment processing workflows include pre-validation of recipient bank details, fraud detection checks, and compliance verification before initiating transactions. The system handles various payment methods including NEFT, RTGS, and UPI transfers, selecting optimal payment channels based on transaction amounts and urgency requirements. Webhook handlers process payment status updates asynchronously, updating payroll records and triggering notification workflows for successful or failed transactions.

Identity Verification and Compliance APIs

The Employee Service integrates with government APIs for PAN card verification and bank account validation, ensuring compliance with regulatory requirements and preventing fraudulent registrations. PAN verification involves secure API calls to Income Tax Department services, validating employee-provided PAN numbers against official records. The system implements request queuing and retry mechanisms to handle API rate limits and temporary service unavailability.

Bank account verification utilizes penny drop mechanisms through banking APIs, confirming account ownership and validity before processing salary payments. The integration includes account holder name matching algorithms to detect discrepancies between employee records and bank account details. Compliance reporting features generate audit trails for verification activities, supporting regulatory compliance and internal audit requirements.

Service Adapter Pattern and Internal APIs

The system implements service adapter patterns to standardize interactions with diverse external APIs, providing consistent internal interfaces regardless of third-party API variations. Each adapter encapsulates specific API authentication, request formatting, response parsing, and error handling logic, enabling seamless integration switching if vendor relationships change. The adapters implement common interfaces defined through abstract base classes, ensuring consistent behavior across different external service integrations.

Internal service adapters handle protocol translation between REST and message-based communication, enabling gradual migration from synchronous to asynchronous patterns. These adapters provide circuit breaker functionality, automatically switching to fallback mechanisms when external services experience outages. Rate limiting implementations ensure external API usage remains within contractual limits while optimizing performance through intelligent request batching and caching strategies.

Security and Key Management

Secure key management utilizes HashiCorp Vault or AWS Secrets Manager for storing and rotating API credentials, webhook secrets, and encryption keys. The system implements proper secret lifecycle management including automated rotation schedules, secure distribution to application instances, and audit logging for secret access. Environment-specific configurations ensure development, staging, and production environments maintain isolated credential sets.

API security implementations include request signing for sensitive operations, IP whitelisting for webhook endpoints, and comprehensive request validation to prevent injection attacks. The system maintains separate API keys for different functional areas, enabling granular access control and simplified credential management. Network security measures include VPN connections for sensitive integrations and certificate pinning for critical API endpoints.

Fallback Mechanisms and Resilience

Comprehensive fallback mechanisms ensure system functionality during external service disruptions, including cached data utilization, degraded functionality modes, and manual override capabilities. The system maintains local copies of critical reference data such as tax tables, salary benchmarks, and regulatory information, enabling continued operation during API outages. Circuit breaker patterns prevent cascading failures by temporarily disabling failing integrations while maintaining core system functionality.

Monitoring and alerting systems track external API performance, response times, and error rates, providing early warning of potential service degradation. The system implements automatic retry logic with exponential backoff for transient failures while maintaining manual intervention capabilities for persistent issues. Disaster recovery procedures include alternative service providers and manual processing workflows for critical operations that cannot be delayed.

Executive Summary

This Cross-Service Communication Plan outlines a comprehensive strategy for building and scaling an AI-powered HR and Payroll system that grows with organizational needs. The plan progresses through three distinct phases, each designed to balance immediate functionality with long-term scalability and reliability.

The current phase utilizes REST-based communication, which provides a solid foundation for the MVP by enabling direct, immediate communication between services like Employee Management, Payroll Processing, Leave Management, and AI Analytics. This approach allows teams to build and debug the system quickly, with clear request-response patterns that are easy to understand and troubleshoot. While this method works well for initial deployment, it creates dependencies between services that can limit performance as the system grows.

The future phase transitions to an event-driven architecture using message queues, fundamentally changing how services communicate by allowing them to work independently and asynchronously. Instead of services directly calling each other, they publish and subscribe to business events like "employee hired," "payroll processed," or "leave approved." This approach dramatically improves system resilience, scalability, and performance by eliminating bottlenecks and enabling services to process work at their own pace.

The third component addresses external integrations with services like Glassdoor for salary benchmarking, Coursera for employee training, Razorpay for payment processing, and government APIs for identity verification. These integrations provide the system with real-world data and capabilities while maintaining security and reliability through proper authentication, caching, and fallback mechanisms.

The overall architecture ensures that HR stakeholders can rely on accurate, real-time information for decision-making, while the technical foundation scales to support growing organizations without compromising performance or reliability. The planned evolution from REST to message-based communication provides a clear upgrade path that maintains system functionality while improving capabilities over time.