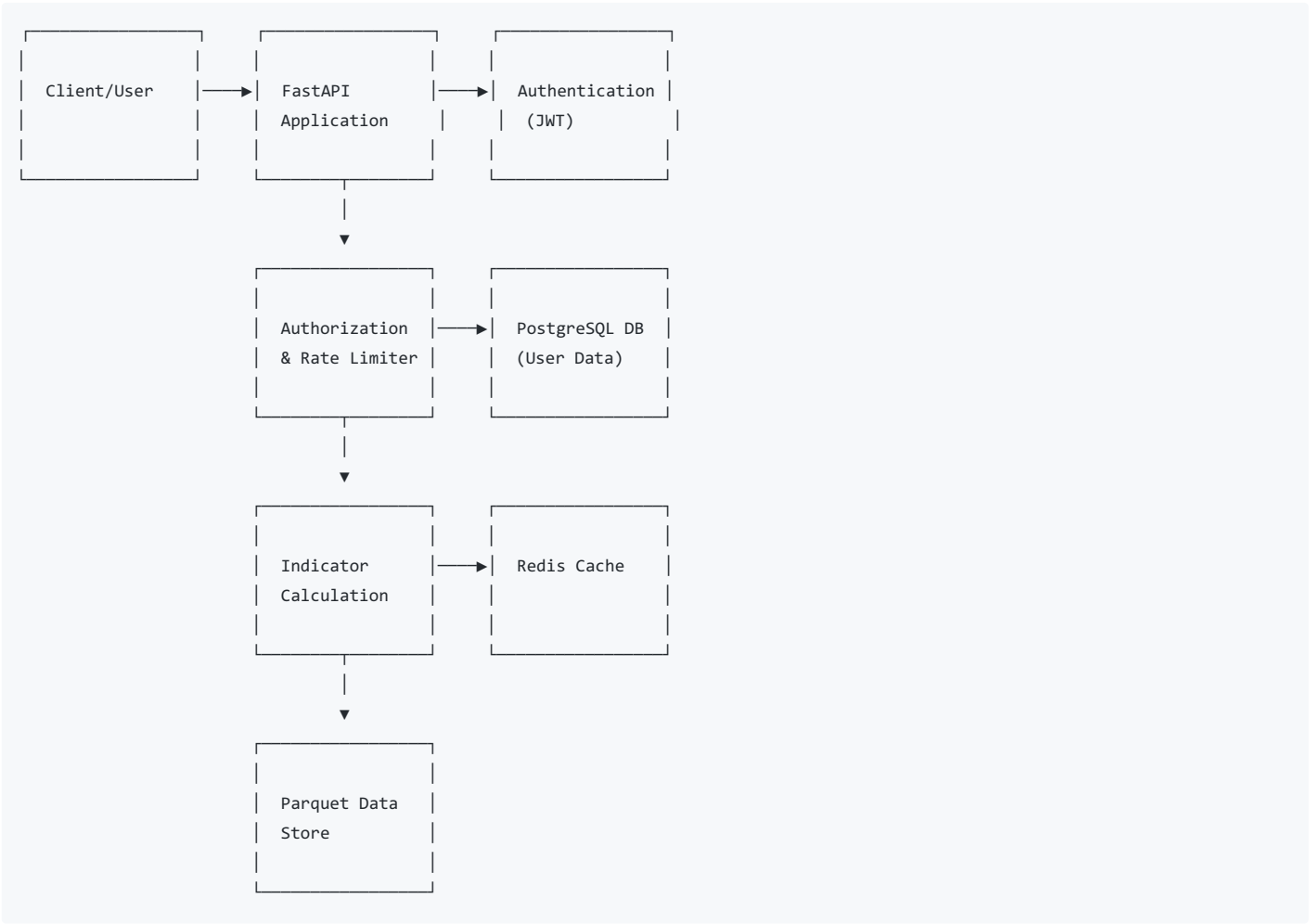
Kalpi Technical Indicator API: Architecture Design Document

1. System Architecture

1.1 High-Level Component Diagram



1.2 Component Descriptions

1. **FastAPI Application**: Core API framework handling HTTP requests, routing, and response serialization.
2. **Authentication (JWT)**: Validates user identity via JWT tokens.
3. **Authorization & Rate Limiter**: Enforces tier-based access controls and request limits.
4. **PostgreSQL Database**: Stores user data, subscription tiers, and request counts.
5. **Indicator Calculation**: Processes financial data using Pandas to compute technical indicators.
6. **Redis Cache**: Stores frequently accessed indicator results to minimize recalculation.
7. **Parquet Data Store**: Efficiently stores and provides access to OHLC stock data.

2. Data Loading Strategy

2.1 Parquet File Handling

I chose to load the entire Parquet file into memory at application startup for the following reasons:

1. **Performance**: The dataset (3 years of daily data for multiple stocks) is relatively small (~10-50MB) and fits comfortably in memory.
2. **Query Speed**: In-memory access provides sub-millisecond data retrieval compared to disk I/O operations.
3. **Simplicity**: Avoids complex data partitioning strategies for this scale of data.

2.2 Data Filtering Approach

When a request comes in:

1. Filter the in-memory DataFrame by symbol and date range.
2. Perform indicator calculations only on the filtered subset.
3. This minimizes computational overhead while maintaining responsiveness.

2.3 Large Data File Considerations

For significantly larger datasets (e.g., tick data or 10+ years of data), I would implement:

1. **Partitioned Storage**: Organize data by symbol and time period.
2. **Lazy Loading**: Load only requested symbols/timeframes on demand.
3. **Dask Integration**: For distributed processing of very large datasets.
4. **Database Migration**: Move to a time-series database like InfluxDB or TimescaleDB.

3. Subscription Model Implementation

3.1 Tier Structure

Feature	Free Tier	Pro Tier	Premium Tier
Daily Request Limit	50	500	Unlimited
Available Indicators	SMA, EMA	SMA, EMA, RSI, MACD	All indicators
Data Range	Last 3 months	Last 1 year	Full 3 years
Price	Free	$X/month	$Y/month

3.2 Implementation Details

1. **User Table Schema**:

   ```
   User(id, username, hashed_password, tier, requests_today, last_request_reset)
   ```

2. **Request Counting**:
   - Increment `requests_today` with each successful API call.
   - Reset counter daily at midnight UTC.
   - Return 429 (Too Many Requests) when limit is exceeded.
3. **Date Range Validation**:
   - Calculate allowed date range based on user tier.
   - Return 403 (Forbidden) if requested range exceeds tier limits.
4. **Indicator Access Control**:
   - Validate requested indicator against allowed indicators for user tier.
   - Return 403 (Forbidden) if indicator is not available for tier.

4. Caching Strategy

4.1 Cache Key Design

Cache keys are constructed as:

```
{symbol}:{start_date}:{end_date}:{indicator_name}:{parameters}
```

Example: `RELIANCE:2023-01-01:2023-03-31:sma:14`

4.2 Cache Implementation

1. **Redis TTL**: Cache entries expire after 24 hours.
2. **Cache Invalidation**: No explicit invalidation needed as financial data is historical.
3. **Cache Hit Flow**:
   - Check cache before calculation.
   - If found, return cached result.
   - If not found, calculate, cache, then return.

4.3 Benefits

1. **Reduced Computation**: Frequently requested indicators (e.g., SMA(20) for popular stocks) are served from cache.
2. **Improved Response Time**: ~1-2ms from cache vs. ~50-100ms for calculation.
3. **Reduced Resource Usage**: Lower CPU utilization during peak loads.

5. Scalability Considerations

5.1 Horizontal Scaling

The architecture supports horizontal scaling through:

1. **Stateless API Design**: No server-side session state.
2. **Shared Redis Cache**: Multiple API instances can leverage the same cache.
3. **Database Connection Pooling**: Efficient handling of concurrent database connections.

5.2 Vertical Scaling

For increased computational demands:

1. **Pandas Optimization**: Using optimized operations and numba acceleration where applicable.
2. **Memory Management**: Careful control of DataFrame copies and memory usage.
3. **Async Processing**: FastAPI's async capabilities for concurrent request handling.

5.3 Load Balancing Strategy

1. **Round-Robin DNS**: Simple distribution of traffic across API instances.
2. **Health Checks**: Regular monitoring of API instance health.
3. **Autoscaling**: Add/remove instances based on CPU utilization and request volume.

6. Security Considerations

6.1 Authentication & Authorization

1. **JWT Implementation**: Short-lived tokens (30 minutes) with refresh token capability.
2. **Password Security**: Bcrypt hashing with appropriate work factor.
3. **Role-Based Access**: Tier-specific permissions enforced at the API level.

6.2 Data Protection

1. **Input Validation**: All request parameters are validated using Pydantic models.
2. **SQL Injection Prevention**: Using SQLAlchemy ORM with parameterized queries.
3. **Rate Limiting**: Prevents abuse and DoS attacks.

6.3 API Security

1. **CORS Configuration**: Restricting cross-origin requests to trusted domains.
2. **HTTP Security Headers**: Implementation of recommended security headers.
3. **TLS/SSL**: All communications encrypted (HTTPS only).

7. Error Handling Strategy

7.1 HTTP Status Codes

Scenario	Status Code	Response
Invalid credentials	401	{"detail": "Invalid authentication credentials"}
Exceeded tier limits	403	{"detail": "Resource not available for your tier"}
Rate limit exceeded	429	{"detail": "Rate limit exceeded. Try again tomorrow."}
Invalid parameters	422	Validation error details
Server error	500	{"detail": "Internal server error"}

7.2 Logging Strategy

1. **Request Logging**: Path, method, status code, response time.
2. **Error Logging**: Full exception details with stack trace.
3. **Audit Logging**: Authentication attempts and tier limit violations.

8. Future Enhancements

8.1 Technical Improvements

1. **GraphQL API**: For more flexible querying of multiple indicators.
2. **WebSocket Support**: Real-time indicator updates for streaming data.
3. **Data Versioning**: Track and serve different versions of indicator calculations.

8.2 Business Features

1. **Custom Indicator Builder**: Allow premium users to define custom indicators.
2. **Batch Processing**: Calculate multiple indicators in a single request.
3. **Export Functionality**: Download indicator data in CSV/Excel format.

9. Conclusion

This architecture balances performance, scalability, and security while implementing a robust tiered subscription model. The design choices prioritize response time and resource efficiency, making it suitable for both small-scale deployments and growth to larger user bases.

The system can handle the current requirements efficiently and has clear paths for scaling as demand increases. The subscription model is implemented with proper controls and validation at multiple levels, ensuring that users can only access features and data appropriate for their tier.