Amazon SQS Integration for DriversKlub Backend

Executive Summary

This document outlines the proposal to integrate **Amazon Simple Queue Service (SQS)** as the message queue system for the DriversKlub backend microservices architecture. SQS will enable asynchronous, event-driven communication between services, improving scalability, reliability, and system decoupling.

Why Amazon SQS?

Benefits Over Kafka

Feature	Amazon SQS	Apache Kafka
Deployment Complexity	✅ Zero - Fully managed	❌ High - Requires infrastructure
Maintenance	✅ None - AWS managed	❌ Ongoing - Self-managed
Cost (Low Volume)	✅ Free tier (1M requests/month)	❌ EC2/EKS costs even at low volume
Scalability	✅ Automatic	⚠️ Manual configuration
Learning Curve	✅ Simple API	❌ Steep learning curve
Reliability	✅ 99.9% SLA, built-in redundancy	⚠️ Depends on setup
Setup Time	✅ Minutes	❌ Days/Weeks

Key Advantages

1. **Zero Infrastructure Management**
   - No servers to provision or maintain
   - No capacity planning required
   - Automatic scaling based on load
2. **Cost-Effective**
   - **Free Tier**: 1 million requests/month
   - **Pay-as-you-go**: $0.40 per million requests after free tier
   - **Estimated cost**: ~$1-2/month for current scale
3. **High Reliability**
   - 99.9% availability SLA
   - Automatic message replication across multiple availability zones
   - Built-in Dead Letter Queue (DLQ) for failed messages
4. **Developer Friendly**
   - Simple REST API
   - Official AWS SDK for Node.js

- Extensive documentation and community support

5. **Production Ready**
    - Used by thousands of companies globally
    - Battle-tested at massive scale
    - Integrated with AWS ecosystem (CloudWatch, IAM, etc.)

Use Cases in DriversKlub

1. Trip Lifecycle Events

Problem: When a trip is created, multiple services need to be notified (notifications, analytics, provider sync).

Solution: Publish `trip.created` event to SQS queue. Multiple consumers process it independently.

```
Trip Service → [SQS Queue] → Notification Service (sends SMS)
                           → Analytics Service (logs event)
                           → Provider Service (syncs with MojoBoxx/MMT)
```

Benefits:

- Trip Service doesn't wait for notifications to be sent
- If notification fails, trip creation still succeeds
- Can add new consumers without modifying Trip Service

2. Driver Status Changes

Problem: When a driver goes online/offline, multiple systems need updates (assignment service, analytics, Rapido sync).

Solution: Publish `driver.status_changed` event.

```
Driver Service → [SQS Queue] → Assignment Service (updates availability)
                             → Rapido Service (syncs captain status)
                             → Analytics Service (tracks uptime)
```

3. Payment Processing

Problem: Payment processing can be slow and should not block trip completion.

Solution: Use FIFO queue for guaranteed order and exactly-once processing.

```
Trip Service → [SQS FIFO Queue] → Payment Service (processes payment)
                                → Accounting Service (records transaction)
```

4. Provider Webhooks

Problem: External providers (MojoBoxx, MMT, Rapido) send webhooks that need reliable processing.

Solution: Queue webhook payloads for asynchronous processing.

```
Webhook Endpoint → [SQS Queue] → Worker Service (processes webhook)
                                → Retry on failure (up to 3 times)
                                → DLQ if still failing
```

5. Notification Delivery

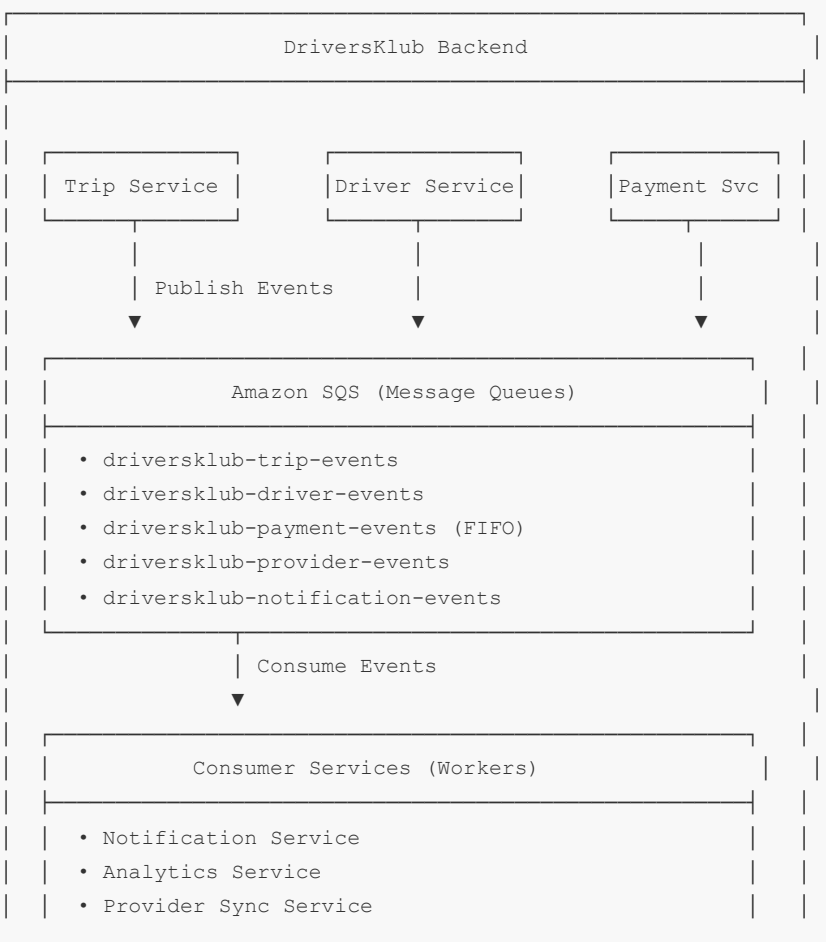
Problem: Sending SMS/Push notifications can fail or be slow.

Solution: Queue all notifications for async delivery with retry logic.

```
Any Service → [SQS Queue] → Notification Service → SMS Gateway
                                                 → Push Notification Service
                                                 → Email Service
```

Architecture Overview

Queue Structure



```
│  │  • Payment Processor                                  │   │
│  └──────────────────────────────────────────────────────┘   │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Event Flow Example: Trip Creation

```
1. User books a trip via API
   ↓
2. Trip Service creates trip in database
   ↓
3. Trip Service publishes "trip.created" event to SQS
   ↓
4. Trip Service returns success to user (fast response)
   ↓
5. SQS delivers message to multiple consumers:
   - Notification Service → Sends booking confirmation SMS
   - Provider Service → Books with MojoBoxx/MMT
   - Analytics Service → Logs trip metrics
   ↓
6. Each consumer processes independently
   - If one fails, others still succeed
   - Failed messages go to Dead Letter Queue for investigation
```

Technical Implementation

Queue Types

Standard Queue (Default)

- **Use for**: Most events (trips, drivers, notifications)
- **Delivery**: At-least-once (may receive duplicates)
- **Ordering**: Best-effort (not guaranteed)
- **Throughput**: Nearly unlimited
- **Cost**: $0.40 per million requests

FIFO Queue

- **Use for**: Payments, critical operations
- **Delivery**: Exactly-once (no duplicates)
- **Ordering**: Guaranteed order
- **Throughput**: 300 transactions/second
- **Cost**: $0.50 per million requests

Event Schema Example

```
// Trip Created Event
{
```

```
  "eventType": "trip.created",
  "eventId": "evt_123456",
  "timestamp": "2026-01-13T13:00:00Z",
  "data": {
    "tripId": "trip_abc123",
    "userId": "user_xyz789",
    "pickupLocation": {
      "lat": 28.6139,
      "lng": 77.2090,
      "address": "Connaught Place, New Delhi"
    },
    "dropLocation": {
      "lat": 28.5355,
      "lng": 77.3910,
      "address": "Noida Sector 62"
    },
    "scheduledTime": "2026-01-13T15:00:00Z",
    "vehicleType": "SEDAN",
    "fare": 1500
  }
}
```

Dead Letter Queue (DLQ)

Purpose: Capture messages that fail processing after multiple retries.

Configuration:

- Max retries: 3 attempts
- Retry delay: Exponential backoff (1s, 2s, 4s)
- DLQ retention: 14 days

Monitoring: CloudWatch alarms when DLQ receives messages.

Cost Analysis

Assumptions

- **Trips per day**: 10,000
- **Events per trip**: 5 (created, assigned, started, completed, payment)
- **Total events/day**: 50,000
- **Monthly events**: 1.5 million

Cost Breakdown

Component	Volume	Unit Cost	Monthly Cost
Standard Queues	1.5M requests	FREE (under 1M) + $0.40/M	**$0.20**
FIFO Queue (Payments)	300K requests	$0.50/M	**$0.15**

Data Transfer	Negligible	$0.09/GB	**$0.05**
TOTAL			**$0.40/month**

Note: First 1M requests are FREE, so actual cost is even lower initially.

Comparison with Kafka

Component	SQS	Self-Hosted Kafka
Infrastructure	$0	$50-100/month (EC2)
Maintenance	$0	10-20 hours/month
Monitoring	Included	$20/month (tools)
Total	**$0.40/month**	**$70-120/month**

Savings: ~$70-120/month + significant developer time

Implementation Roadmap

Phase 1: Setup & Infrastructure (Week 1)

Duration: 2-3 days

- [ ] Create AWS account (if needed)
- [ ] Set up IAM user with SQS permissions
- [ ] Create SQS queues via AWS Console or script
- [ ] Install AWS SDK (`@aws-sdk/client-sqs`)
- [ ] Create shared queue package (`@driversklub/queue`)
- [ ] Configure environment variables

Deliverables:

- All queues created in AWS
- Queue URLs documented
- Basic publish/consume example working

Phase 2: Core Integration (Week 2)

Duration: 5-7 days

- [ ] Implement queue manager (publish/consume utilities)
- [ ] Define event schemas with TypeScript types
- [ ] Add event publishing to Trip Service
- [ ] Add event publishing to Driver Service
- [ ] Create consumer for Notification Service
- [ ] Implement retry logic and DLQ handling

Deliverables:

- Trip and Driver events publishing successfully
- Notifications being sent via queue
- DLQ monitoring in place

Phase 3: Extended Features (Week 3)

Duration: 5-7 days

- [ ] Add payment event processing (FIFO queue)
- [ ] Implement provider webhook queuing
- [ ] Add analytics event consumers
- [ ] Create monitoring dashboard
- [ ] Write integration tests
- [ ] Performance testing

Deliverables:

- All event types implemented
- Comprehensive test coverage
- Monitoring and alerting configured

Phase 4: Production Deployment (Week 4)

Duration: 3-5 days

- [ ] Deploy to staging environment
- [ ] Load testing with production-like data
- [ ] Monitor CloudWatch metrics
- [ ] Gradual rollout to production
- [ ] Documentation and runbooks

Deliverables:

- Production deployment complete
- Team training completed
- Operational runbooks ready

Total Timeline: 3-4 weeks for full implementation

Minimal Viable Product (MVP) - Quick Start

For immediate testing and validation, we can implement a **minimal version in 1-2 days**:

MVP Scope

1. **Single Queue**: `driversklub-events` (Standard Queue)
2. **One Publisher**: Trip Service publishes `trip.created` events
3. **One Consumer**: Notification Service sends SMS on trip creation
4. **Basic Error Handling**: DLQ for failed messages

MVP Benefits

- Prove the concept works
- Test AWS integration
- Validate event schema design
- Measure performance
- Get team familiar with SQS

MVP to Full Implementation

Once MVP is validated, expand incrementally:

- Add more event types
- Add more consumers
- Implement FIFO queues for payments
- Add monitoring and alerting

Risk Assessment

Technical Risks

Risk	Impact	Probability	Mitigation
AWS account issues	High	Low	Set up IAM properly, test credentials
Message loss	High	Very Low	SQS has 99.9% SLA, use DLQ
Duplicate processing	Medium	Low	Implement idempotency keys
Cost overrun	Low	Very Low	Set CloudWatch billing alarms
Learning curve	Low	Medium	Comprehensive documentation, training

Operational Risks

Risk	Impact	Probability	Mitigation
DLQ fills up	Medium	Low	CloudWatch alarms, daily monitoring
Consumer lag	Medium	Low	Auto-scaling consumers, monitoring
AWS region outage	High	Very Low	Multi-region setup (future)

Overall Risk Level: **LOW** - SQS is a mature, battle-tested service

---

Monitoring & Observability

CloudWatch Metrics (Built-in)

- **ApproximateNumberOfMessagesVisible**: Messages in queue
- **ApproximateNumberOfMessagesNotVisible**: Messages being processed
- **NumberOfMessagesSent**: Publishing rate
- **NumberOfMessagesReceived**: Consumption rate
- **ApproximateAgeOfOldestMessage**: Queue lag indicator

Custom Metrics

- Event processing time
- Success/failure rates per event type
- DLQ message count by error type

Alerts

- DLQ receives messages → Slack/Email alert
- Queue depth > 1000 → Scale up consumers
- Processing time > 30s → Investigate performance

Security Considerations

IAM Permissions

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sqs:SendMessage",
        "sqs:ReceiveMessage",
        "sqs:DeleteMessage",
        "sqs:GetQueueAttributes"
      ],
      "Resource": "arn:aws:sqs:us-east-1:*:driversklub-*"
    }
  ]
}
```

Best Practices

- ✅ Use IAM roles (not hardcoded keys) in production
- ✅ Encrypt messages at rest (AWS KMS)
- ✅ Encrypt messages in transit (HTTPS)
- ✅ Separate IAM users for dev/staging/prod
- ✅ Rotate credentials regularly
- ✅ Use VPC endpoints for private communication

Success Metrics

Technical Metrics

- **Message Processing Time**: < 5 seconds (p95)
- **Queue Depth**: < 100 messages during normal operation
- **DLQ Messages**: < 0.1% of total messages
- **Availability**: > 99.9%

Business Metrics

- **Notification Delivery**: > 99% success rate
- **Trip Processing**: 100% reliability
- **System Decoupling**: Services can deploy independently
- **Developer Velocity**: Faster feature development

Alternatives Considered

1. Apache Kafka

Pros: High throughput, event streaming, complex event processing **Cons**: Complex setup, high operational overhead, overkill for current scale **Verdict**: ❌ Too complex for current needs

2. RabbitMQ

Pros: Feature-rich, good for complex routing **Cons**: Self-hosted, requires maintenance, learning curve **Verdict**: ❌ Still requires infrastructure management

3. Redis Pub/Sub

Pros: Fast, simple, already using Redis **Cons**: No persistence, no guaranteed delivery, not designed for queuing **Verdict**: ❌ Not reliable enough for critical events

4. Database-based Queue

Pros: No new infrastructure, simple **Cons**: Poor performance at scale, polling overhead, not designed for queuing **Verdict**: ❌ Not scalable

5. Amazon SQS ✅

Pros: Fully managed, reliable, cost-effective, simple, scalable **Cons**: AWS vendor lock-in (minor concern) **Verdict**: ✅ **Best fit for current requirements**

Conclusion

Amazon SQS provides the optimal balance of:

- ✅ **Simplicity**: Minimal setup and maintenance
- ✅ **Reliability**: 99.9% SLA with built-in redundancy

- ✅ **Cost**: Free tier + pay-as-you-go pricing
- ✅ **Scalability**: Automatic scaling with load
- ✅ **Developer Experience**: Simple API, great documentation

Recommendation

Proceed with Amazon SQS integration using the phased approach:

1. **Week 1**: MVP implementation (1 queue, 1 publisher, 1 consumer)
2. **Week 2-3**: Expand to all event types and consumers
3. **Week 4**: Production deployment with monitoring

Next Steps

1. **Approve this proposal**
2. **Set up AWS account** (if not already done)
3. **Create IAM credentials** for development
4. **Begin MVP implementation** (estimated 1-2 days)

Appendix

A. Estimated Development Effort

Task	Effort (Hours)
AWS setup and queue creation	4
Queue package development	8
Event schema definition	4
Trip Service integration	6
Driver Service integration	6
Notification consumer	6
Payment consumer (FIFO)	8
Testing and debugging	12
Documentation	4
Total	**58 hours (~1.5 weeks)**

B. Required AWS Permissions

```
sqs:CreateQueue
sqs:DeleteQueue
sqs:GetQueueAttributes
sqs:SetQueueAttributes
sqs:SendMessage
```

```
sqs:ReceiveMessage
sqs:DeleteMessage
sqs:PurgeQueue
cloudwatch:PutMetricData
```

C. Useful Resources

- AWS SQS Documentation
- AWS SDK for JavaScript
- SQS Best Practices
- SQS Pricing Calculator

Document Version: 1.0
Date: January 13, 2026
Author: DriversKlub Engineering Team
Status: Pending Approval