Push Notifications & Real-Time Communication

Visual Guide to Modern Event-Driven Architecture

1. The Fundamental Problem: Polling vs Push

Traditional Polling Approach (The Hard Way)

Client → Server/Database

Request 1: "Any new orders?" → "No new orders"

Request 2: "Any new orders?" → "No new orders"

Request 3: "Any new orders?" → "Yes! Order #123"

Timeline: *Wasted requests every 30 seconds*
Order created at 2:15pm, discovered at 2:16pm
59 seconds of delay

Modern Push Approach (The Smart Way)

Database → Application Server → Multiple Services

Email Service | Mobile App | Analytics System | Warehouse System

Timeline: *Order created at 2:15:00pm*
All systems notified by 2:15:01pm
1 second total delay

2. Database Change Detection Methods

Method 1: Application-Level

HTTP POST /orders
↓
Flask App
↓
Save to Database
↓
Immediate Notification

Method 2: Background Polling

Background Process
↓
Query Recent Changes
↓
Check Every 10s
↓
Send Notifications

Method 3: Database Triggers

Database INSERT/UPDATE
↓
Database Trigger
↓
NOTIFY Channel
↓
Send Notifications

3. Webhook Architecture

Order Created → Your Server → HTTP POST https://partner-api.com → Partner Server → Process Data

HTTP 200 Response: {'status': 'success', 'processed_at': '2025-08-11T14:30:00Z'}

Network Path: *London → Internet → Azure West Europe*
Latency: *~50-200ms*
Protocol: *HTTPS over TCP/IP*

4. Network Level Understanding

Your AWS Server (London)
Order Created → Webhook Handler

Internet Infrastructure
London ISP → Internet Backbone → European Network → Azure Network

Azure SaaS (West Europe)
Load Balancer → App Service → Webhook Endpoint → Database

Network Details:
- *Protocol: HTTPS (Port 443)*
- *Latency: 50-200ms*
- *Same as browsing a website*
- *No persistent connection*
- *Each webhook = new TCP connection*

5. WebSockets for Real-Time Chat

WebSocket Handshake Process

Browser
Sends HTTP upgrade request

Handshake
GET /chat HTTP/1.1
Connection: Upgrade
Upgrade: websocket

Server
HTTP 101 Switching Protocols
Connection established

Persistent Bidirectional Communication

Browser "Hello!" →→ Server ←← All Users Receive Message

6. High-Frequency Data Solutions

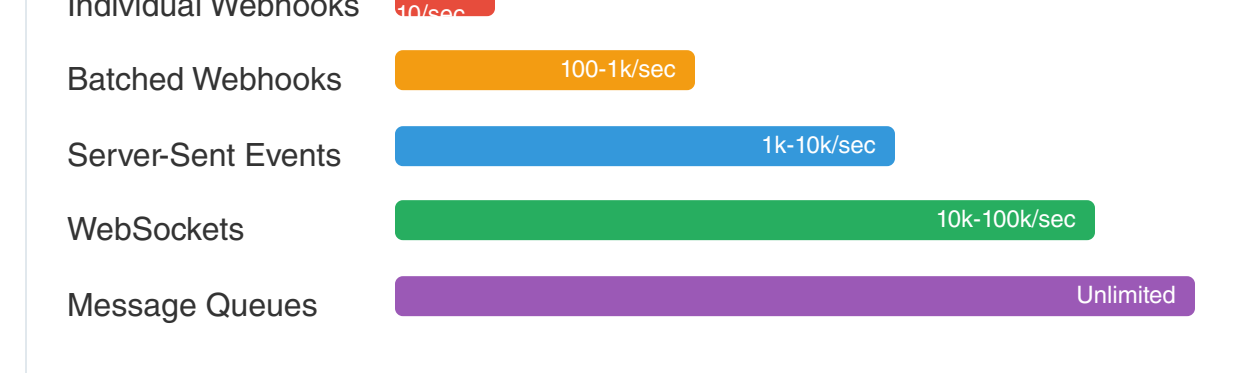
When Webhooks Break Down

Individual Webhooks
1000 TCP connections/second
❌ Server exhausted

Network Overhead per Call
DNS Lookup: ~5ms
TCP Handshake: ~50ms
TLS Handshake: ~100ms
HTTP Request/Response: ~100ms
Total: ~265ms per webhook

Batched Webhooks
20 connections/second
✅ 50x reduction in overhead

Solution Comparison



7. Pattern Comparison Matrix

Pattern	Latency	Reliability	Complexity	Scalability	Use Case
Individual Webhooks	Low (50-200ms)	Medium	Low	Low	< 10/second
Batched Webhooks	Medium (1-2s)	Medium	Low	Medium	10-500/second
Server-Sent Events	Low (50-200ms)	Medium	Medium	High	100-10k/second
WebSockets	Very Low (<50ms)	Medium	High	High	Real-time apps
Message Queues	Medium (1-5s)	Very High	Medium	Very High	Any volume

8. AWS Serverless Architecture

Data Layer

DynamoDB Table
Orders
↓
DynamoDB Stream
Change Events

Event Processing

Lambda Function
Order Processor
↓
SNS Topic
Order Events

Multiple Delivery Channels

Email via SES | SMS via SNS
HTTP Webhook | SQS Queue

Serverless Benefits:
- *No servers to manage*
- *Automatic scaling*
- *Pay per execution*
- *High availability*
- *Event-driven architecture*

9. Decision Tree: Choosing the Right Pattern

What type of communication do you need?

One-way (AWS → Azure)
How frequent are updates?
- < 10/sec: Individual Webhooks
- 10-500/sec: Batched Webhooks
- 500-10k/sec: Server-Sent Events
- > 10k/sec: Message Queues

Two-way (Bidirectional)
Real-time requirements?
- < 100ms latency: WebSockets
- 1-5 seconds OK: Webhook + Callback

10. Security Architecture Patterns

Webhook Security Layers

Request Origin → IP Whitelisting → HTTPS/TLS → API Key Auth → Signature Verify → Body Validation → Rate Limiting → Process Request

11. Error Handling and Retry Patterns

Webhook Request Sent
↓
Check Response Code

2xx Success ↓ ✅ Complete
4xx Client Error ↓ ❌ Don't Retry Alert Team
5xx Server Error ↓ Retry with Backoff

Retry Schedule Example:
- *Attempt 1: Immediate*
- *Attempt 2: 2 seconds later*
- *Attempt 3: 4 seconds later*
- *Attempt 4: 8 seconds later*
- *Failure: Dead letter queue*

12. Cost Optimization Strategies

AWS Cost Optimization
- Use Reserved Instances
  40-60% cost reduction
- Auto Scaling Groups
  Scale down during off-hours
- SQS Batching
  Reduce API call costs
- CloudWatch Log Retention
  30 days instead of forever

Azure Cost Optimization
- App Service Plans
  Shared during development
- Azure SQL Elastic Pools
  Share resources across DBs
- Function Apps
  Consumption plan for sporadic
- Storage Lifecycle Policies
  Archive old webhook logs

Architecture Cost Comparison
(1000 webhooks/day)

Basic Webhooks
£25/month
EC2 + App Service always on

Serverless
£8/month
Lambda + Azure Functions

Hybrid
£15/month
Best of both approaches

13. Common Anti-Patterns to Avoid

❌ What NOT to Do

Synchronous Processing
Block webhook response while doing heavy work

No Retry Logic
Give up after first failure without attempting retry

Infinite Retries
Keep retrying forever without backoff or limit

No Authentication
Accept webhooks from any source without verification

Large Payloads
Send entire database records instead of just changes

✅ Best Practices Instead

Async Processing
Return 200 immediately, process in background

Exponential Backoff
Smart retry with increasing delays

Circuit Breaker
Stop retrying after reasonable attempts

Strong Authentication
Verify signatures and API keys properly

Minimal Payloads
Send only essential data, provide API for details

14. Evolution Path: From Simple to Enterprise

Phase 1: MVP (Month 1) Simple Webhooks, Basic error logging → Phase 2: Production (Month 3) Add Authentication, Retry logic, Basic monitoring → Phase 3: Scale (Month 6) Batched webhooks, Load balancing, DB optimization → Phase 4: Enterprise (Month 12) Message queues, Multi-region, Advanced analytics

Phase 1: Get it working
1-10 webhooks/day

Phase 2: Make it reliable
10-100 webhooks/day

Phase 3: Handle growth
100-10k webhooks/day

Phase 4: Enterprise scale
10k+ webhooks/day

15. Production Architecture Example

AWS Production Environment

Application Load Balancer
↓
EC2 Auto Scaling Group
↓
Flask Applications
RDS PostgreSQL Multi-AZ | Redis Cluster Session/Cache
SQS Queue Dead Letter Queue

Monitoring & Security
- CloudWatch Logs
- CloudWatch Metrics
- AWS WAF
- VPC Security Groups

Security Features:
- Webhook signature verification
- API key authentication
- IP whitelisting
- HTTPS encryption

Azure Partner Environment

Azure Application Gateway
↓
App Service Multiple Instances
Azure SQL Database Geo-Redundant | Service Bus Queue
Application Insights

16. Monitoring and Observability

Webhook Success Rate
Target: > 99.9%

Average Response Time
Target: < 200ms

Queue Depth
Alert if > 1000

Error Rate by Type
4xx vs 5xx patterns

Alerting Thresholds

🚨 **Critical**
Success rate < 95%
Page on-call team

Warning
Response time > 500ms
Send email alert

📊 **Info**
New partner onboarded
Update dashboard

17. Code Examples

Robust Webhook Implementation

```
import requests from tenacity import retry, stop_after_attempt, wait_exponential class RobustNotifier: def __init__(self): self.primary_webhook = "https://azure-saas.com/webhook" self.fallback_queue = boto3.client('sqs') @retry( stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1, min=4, max=10) ) def send_notification(self, order_data): try: response = requests.post( self.primary_webhook, json=order_data, timeout=30, headers={'X-Signature': self.generate_signature(order_data)} ) response.raise_for_status() except requests.exceptions.RequestException: # Fallback to queue self.fallback_queue.send_message( QueueUrl='https://sqs.region.amazonaws.com/fallback', MessageBody=json.dumps(order_data) ) raise # Re-raise for retry logic
```

WebSocket Chat Server

```
from flask_socketio import SocketIO, emit, join_room app = Flask(__name__) socketio = SocketIO(app) @socketio.on('join_chat') def handle_join(data): username = data['username'] room = data['room'] join_room(room) # Tell everyone in room that user joined emit('user_joined', { 'message': f'{username} joined the chat' }, room=room) @socketio.on('send_message') def handle_message(data): # Send to everyone in room instantly emit('new_message', { 'username': data['username'], 'message': data['message'], 'timestamp': datetime.utcnow().isoformat() }, room=data['room'])
```

Key Takeaways

Start Simple
Begin with webhooks for low-frequency notifications

Scale Smartly
Move to batching, SSE, or queues as volume increases

Choose Based on Requirements
Consider latency, reliability, and complexity needs

Plan for Failure
Implement retry logic, monitoring, and fallback mechanisms

Security First
Always verify webhook signatures and use HTTPS

Monitor Everything
Track success rates, latency, and error patterns

This guide demonstrates the evolution from simple request-response to sophisticated event-driven architectures.
Each pattern has its place - the key is choosing the right tool for your specific requirements.