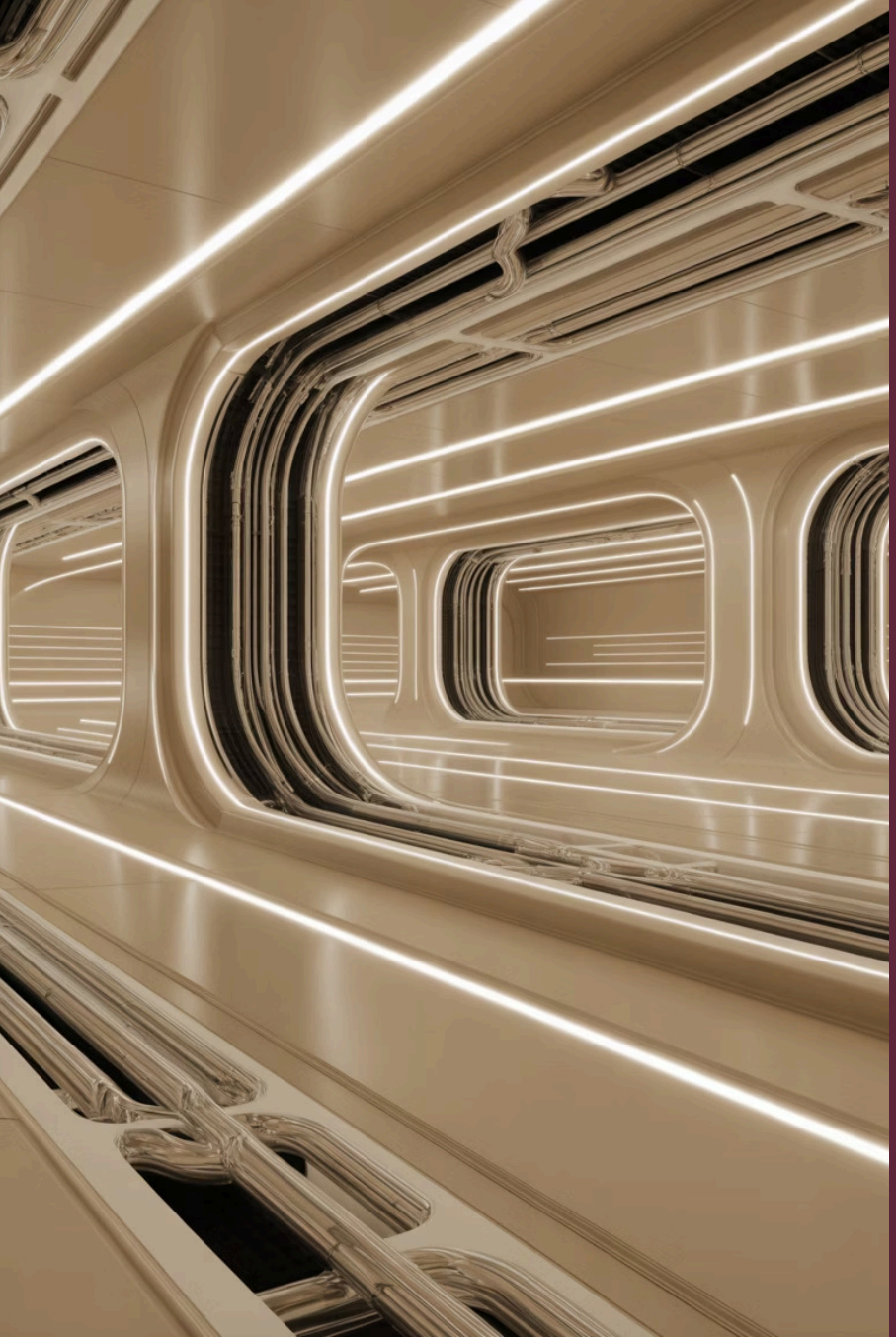


Building Real-Time Feature Pipelines: JavaScript's Role in Modern Data-Driven Applications

Discover how JavaScript powers the next generation of intelligent, data-driven web applications through real-time feature engineering and machine learning integration.

By Venkata Chandra Sekhar Sastry Chilkuri

Indiana University

Conf42.com JavaScript 2025

The Evolution of Web Applications

Traditional Architecture

Static content delivery with minimal personalisation. Request-response patterns dominated, with limited real-time capabilities and basic user interactions.

- Server-rendered pages
- Batch data processing
- Delayed feature updates
- Generic user experiences

Modern Data-Driven Apps

Dynamic, personalised experiences powered by real-time data streams and machine learning models that adapt instantly to user behaviour.

- Real-time feature computation
- Streaming data pipelines
- Instant model inference
- Hyper-personalised content

The Feature Pipeline Challenge

Feature pipelines transform raw data into meaningful inputs for machine learning models and application logic. Modern applications require features that are computed in real-time, cached efficiently, and consistent across environments from servers to edge nodes to browsers.

Feature Computation

Processing raw data into useful features at scale whilst maintaining low latency and high throughput.

Caching Strategy

Balancing freshness with performance through intelligent storage and retrieval mechanisms.

Cross-Environment Consistency

Ensuring identical logic execution across Node.js servers, edge workers, and browser clients.

Why JavaScript for Feature Pipelines?

Universal Language

Write feature computation logic once and deploy it across your entire stack backend services, edge functions, and frontend applications use the same codebase, eliminating translation errors and reducing maintenance overhead.

Rich Ecosystem

Leverage mature libraries for data processing, streaming, caching, and ML inference. From Apache Kafka clients to TensorFlow.js, the JavaScript ecosystem provides production-ready tools for every pipeline stage.

Native Async Support

Built-in async/await patterns and event-driven architecture make JavaScript naturally suited for real-time data streams, concurrent processing, and non-blocking I/O operations essential for low-latency pipelines.

Edge Computing Ready

Deploy feature computation close to users with edge runtimes like Cloudflare Workers and Vercel Edge Functions, reducing latency whilst maintaining full JavaScript compatibility.

Streaming Data Processing with Kafka

Apache Kafka powers real-time data processing. With KafkaJS, JavaScript developers build robust streaming pipelines in Node.js for ingesting events, computing features, and publishing results.

1

Event Ingestion

Consume real-time data from Kafka topics (user interactions, system events).

2

Feature Computation

Transform raw events into features via windowing, aggregation, and enrichment.

3

Feature Publishing

Publish computed features to downstream systems (Redis, feature stores, other topics).

```
const kafka = new Kafka({ clientId: 'feature-pipeline' });
const consumer = kafka.consumer({ groupId: 'features' });

await consumer.subscribe({ topic: 'user-events' });
await consumer.run({
 eachMessage: async ({ message }) => {
 const features = computeFeatures(message.value);
 await publishFeatures(features);
 }
});
```

Real-Time Feature Serving Architecture

01

Feature Computation Service

Node.js microservices process incoming data streams and compute features using business logic and statistical transformations.

02

Redis Feature Store

Computed features are cached in Redis with configurable TTL values, enabling sub-millisecond retrieval for online inference.

03

WebSocket Distribution

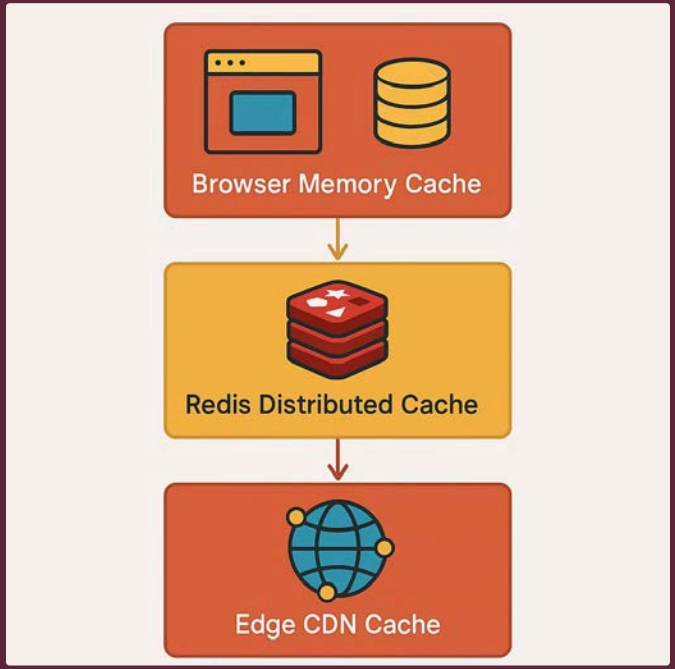
Features stream to connected clients via WebSockets, ensuring browsers receive updates instantly without polling overhead.

04

Client-Side Inference

Browser applications use cached features with TensorFlow.js or ONNX.js to run ML models locally, eliminating server round-trips.

Implementing Efficient Caching Strategies



Multi-Layer Cache Design

Effective feature pipelines employ multiple caching layers, each optimised for specific access patterns and latency requirements. JavaScript's flexible architecture supports seamless integration across all layers.

- **Browser Memory Cache**

  In-memory storage for frequently accessed features using Map or LRU cache implementations with sub-microsecond access times.

- **Redis Distributed Cache**

  Centralised feature store accessible across services with millisecond latency, supporting TTL-based expiration and pub/sub updates.

- **Edge CDN Cache**

  Geographically distributed caching at edge locations for globally consistent feature access with minimal latency.

Client-Side ML Inference with JavaScript

Browsers now act as powerful ML platforms. TensorFlow.js and ONNX.js allow sophisticated, privacy-preserving model inference directly in the browser, reducing server dependencies.

TensorFlow.js

Full ML framework for browser-based training and inference, leveraging WebGL for GPU acceleration.

ONNX.js

Optimized runtime for pre-trained models (PyTorch, scikit-learn). Delivers high performance for inference workloads.

Edge Inference

Deploy models to edge workers for server-side inference, combining browser ML with edge computing.

```
import * as tf from '@tensorflow/tfjs';

const model = await tf.loadLayersModel('/model.json');
const features = tf.tensor2d([computedFeatures]);
const prediction = model.predict(features);
const result = await prediction.data();
```

Ensuring Server-Client Logic Consistency

One of the biggest challenges in feature pipelines is maintaining identical behaviour across environments. Inconsistent feature computation leads to training-serving skew, degraded model performance, and unpredictable user experiences.

1

Shared Code Modules

Package feature computation logic as isomorphic npm modules that run identically in Node.js and browsers.

2

Comprehensive Testing

Unit tests verify identical outputs across environments using the same test fixtures and assertions.

3

Schema Validation

Define feature schemas with TypeScript or JSON Schema to catch inconsistencies at compile time.

4

Monitoring & Alerts

Track feature distributions and detect drift between server and client computations in production.

Monitoring Pipeline Performance

Production feature pipelines require comprehensive observability to maintain reliability and performance. JavaScript applications can leverage modern monitoring tools to track critical metrics across the entire pipeline.

- **Feature Retrieval**

  Target latency for cached feature access from Redis or memory stores

- **Inference Time**

  Client-side model prediction latency including feature preparation

- **Pipeline Uptime**

  Availability target for real-time feature computation services

- **End-to-End Latency**

  Total time from event ingestion to feature availability for inference

Key Metrics to Track

Throughput Metrics

Events processed per second, features computed per minute, and cache hit rates across all pipeline stages.

Latency Distributions

P50, P95, and P99 latencies for feature computation, retrieval, and inference operations to identify bottlenecks.

Error Rates

Failed computations, cache misses, model inference errors, and their impact on user-facing functionality.

Resource Utilisation

Memory consumption, CPU usage, network bandwidth, and Redis memory for capacity planning and optimisation.

Feature Freshness

Age of cached features and staleness indicators to balance performance with data currency requirements.

Practical Application Patterns

Recommendation Engines

Build personalized product or content recommendations by computing user preference features in real-time and running collaborative filtering models in the browser.

- User interaction history aggregation
- Item similarity computations
- Real-time ranking adjustments
- A/B testing different algorithms

Personalisation Features

Tailor user experiences based on behaviour, context, and preferences using features computed from streaming events and historical data.

- Dynamic content filtering
- Contextual UI adaptations
- Predictive prefetching
- Adaptive notification timing

Real-Time Analytics Dashboards

Display live metrics and insights by streaming computed features directly to browser visualisations. WebSockets deliver updates as new data arrives, whilst client-side aggregations handle drill-downs without server queries.

Architectural Best Practices

Decouple Computation from Serving

Separate feature computation services from serving APIs. This allows independent scaling, versioning, and deployment whilst maintaining clear boundaries between batch processing and online inference workloads.

Design for Failure

Implement graceful degradation when features are unavailable. Cache features with reasonable TTLs, provide fallback values, and ensure applications remain functional even when ML models or feature stores are temporarily offline.

Version Your Features

Treat features as APIs with explicit versioning. This enables safe schema evolution, gradual rollouts of new features, and rollback capabilities when issues arise in production environments.

Optimise for Common Paths

Profile your pipeline to identify hot paths and optimise aggressively. Cache frequently accessed features, precompute expensive transformations, and batch operations where latency requirements allow.

Getting Started: Implementation Roadmap

Start with Offline Features

Begin by computing features in batch jobs and serving them from a simple cache. This establishes patterns before adding streaming complexity.

Add Real-Time Computation

Introduce event-driven feature updates using Kafka or message queues, incrementally replacing batch features with streaming equivalents.

Enable Client-Side Inference

Deploy ML models to browsers using TensorFlow.js, starting with simple models before tackling complex neural networks.

Scale and Optimise

Monitor performance, identify bottlenecks, and incrementally optimise your pipeline based on real production metrics and user impact.

This incremental approach reduces risk whilst building team expertise. Each stage delivers value independently, and you can pause at any point with a working system.

Key Takeaways

JavaScript Enables Full-Stack Features

Use the same language across your entire pipeline from backend processing to edge computing to browser inference eliminating integration friction and maintaining consistency.

Streaming + Caching = Real-Time

Combine event-driven feature computation with multi-layer caching strategies to deliver low-latency, fresh features that power personalised experiences at scale.

Client-Side ML Is Production-Ready

Modern JavaScript ML frameworks enable sophisticated inference in browsers and edge environments, reducing server costs whilst improving privacy and latency.

Thank You!