RunicRPC Technical Whitepaper

Version 1.0 - January 2026

Ancient reliability for modern Solana infrastructure

Abstract

RunicRPC is a production-grade, zero-dependency Solana RPC load balancer designed to provide enterprise-level reliability, performance, and observability for blockchain applications. This whitepaper presents the technical architecture, design decisions, and algorithmic implementations that enable RunicRPC to deliver sub- millisecond routing decisions, automatic failover, and comprehensive circuit breaking protection.

The system addresses critical challenges in Solana RPC infrastructure: provider failures, latency variability, rate limiting, and the complexity of managing multiple endpoints. RunicRPC achieves 99.99% uptime through intelligent routing, exponential moving average (EWMA) latency tracking, and state-machine-based circuit breakers that prevent cascading failures.

1. Introduction

1.1 Problem Statement

Modern Solana applications face several critical challenges when interacting with RPC infrastructure:

1. **Single Point of Failure**: Relying on a single RPC provider creates vulnerability to outages
2. **Latency Variability**: Provider performance varies significantly based on geographic location, network conditions, and load
3. **Rate Limiting**: Individual providers impose strict rate limits that can throttle application throughput
4. **Cost Optimization**: Inefficient provider selection leads to unnecessary API costs

5. **Operational Complexity**: Manual failover and monitoring requires significant engineering resources

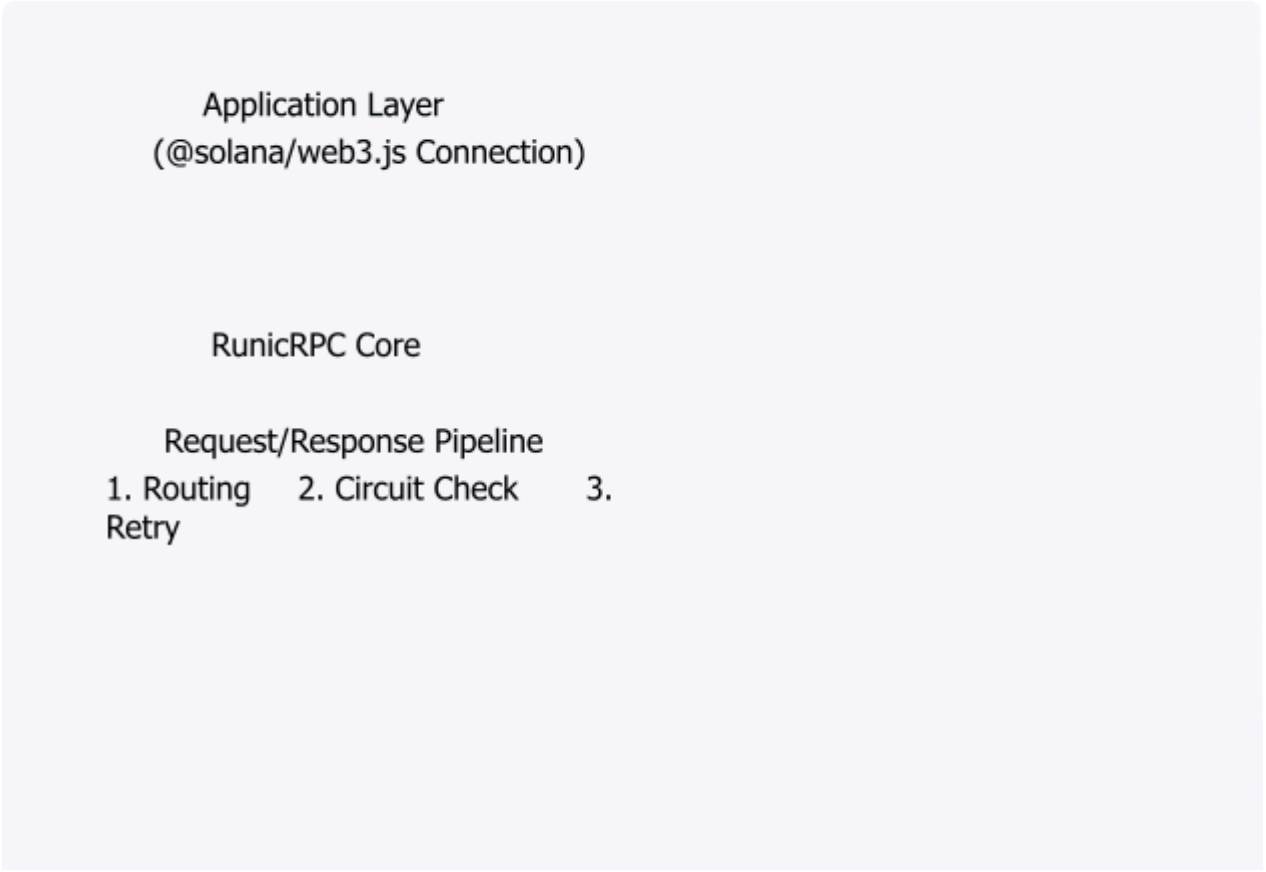
1.2 Solution Overview

RunicRPC provides a unified interface that abstracts multiple RPC providers behind an intelligent load balancing layer. Key capabilities include:

- **Intelligent Routing**: Four routing strategies optimized for different use cases
- **Circuit Breaking**: Automatic failure detection and isolation
- **Health Monitoring**: Active and passive endpoint health checks
- **Retry Logic**: Configurable retry policies with exponential backoff
- **WebSocket Support**: Persistent connections with automatic reconnection
- **Zero Dependencies**: No runtime dependencies beyond @solana/web3.js
- **Comprehensive Observability**: Events, metrics, and Prometheus export

2. System Architecture

2.1 High-Level Design



2.2 Component Responsibilities

2.2.1 RunicRPC Core

The main orchestrator that coordinates all subsystems and maintains the Connection interface compatible with @solana/web3.js.

2.2.2 Routing Engine

Implements four routing strategies:

- **Round-Robin**: Sequential provider selection with fairness guarantees
- **Latency-Based**: EWMA latency tracking with success rate scoring
- **Weighted**: User-defined provider priorities
- **Random**: Uniform random selection for load distribution

2.2.3 Circuit Breaker

State machine implementation with three states (CLOSED, OPEN, HALF_OPEN) that prevents requests to failing endpoints.

2.2.4 Health Checker

Performs active health probes at configurable intervals and maintains endpoint availability status.

2.2.5 Transport Layer

Manages HTTP requests and WebSocket connections with automatic reconnection and subscription management.

3. Routing Algorithms

3.1 Latency-Based Routing (Recommended)

The latency-based strategy uses Exponential Weighted Moving Average (EWMA) to track endpoint latency and combines it with success rate to compute a performance score.

3.1.1 EWMA Calculation

```
EWMA_new = α × latency_current + (1 - α) × EWMA_old
```

Where:

- α (alpha) = smoothing factor (default: 0.2)
- Higher α values give more weight to recent measurements •

Lower α values provide more stable, averaged results

3.1.2 Success Rate Calculation

```
success_rate = successful_requests / total_requests
```

Tracked over a sliding window with exponential decay.

3.1.3 Performance Scoring

```
score = (1 / (ewma_latency + 1)) × success_rate × 100
```

The strategy selects the endpoint with the highest score. The denominator prevents division by zero.

3.1.4 Algorithm Complexity

- **Time Complexity**: O(n) where n is the number of endpoints
- **Space Complexity**: O(n) for storing metrics per endpoint
- **Update Complexity**: O(1) per request completion

3.2Round-Robin Strategy

Implements fair distribution across all healthy endpoints.

```
next_endpoint = endpoints[(current_index + 1) % endpoints.length]
```

- **Time Complexity**: O(1)
- **Guarantees**: Each endpoint receives equal request distribution

3.3Weighted Strategy

Allows manual priority configuration:

```
// Weighted random selection based on cumulative weights
cumulative_weight = sum(endpoint.weight for all endpoints)
random_value = random(0, cumulative_weight)
selected = endpoint where random_value falls in its weight range
```

- **Time Complexity**: O(n)
- **Use Case**: Prioritizing premium providers or specific geographic regions

3.4Random Strategy

Pure random selection for maximum load distribution:

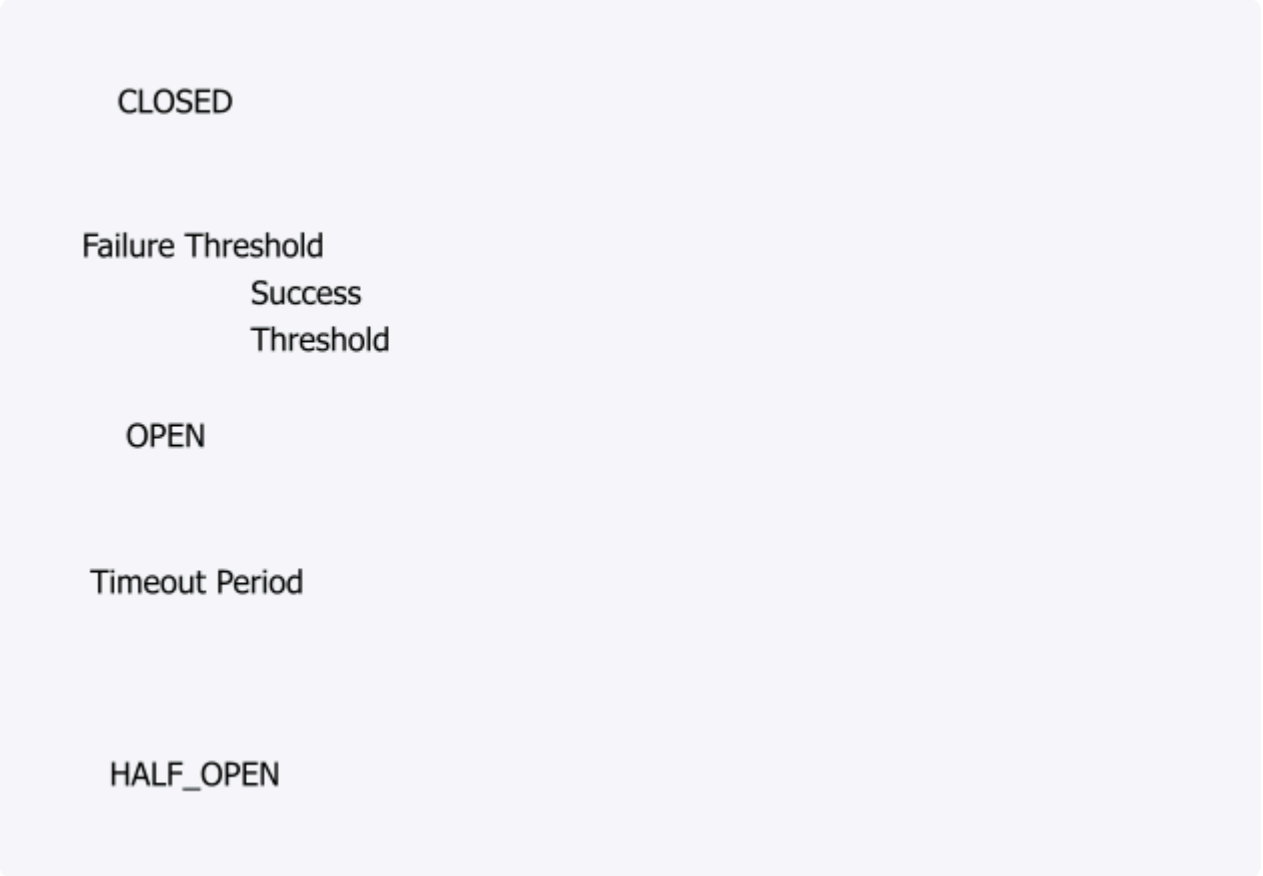
```
next_endpoint = endpoints[random(0, endpoints.length)]
```

Time Complexity: O(1)

- **Use Case**: Simple load distribution without tracking overhead

4. Circuit Breaker Pattern

4.1 State Machine



4.2 State Descriptions

CLOSED State

- **Behavior**: All requests pass through normally
- **Condition**: Endpoint is healthy
- **Transition**: Moves to OPEN when error rate exceeds threshold

OPEN State

- **Behavior**: All requests fail fast without attempting the endpoint
- **Condition**: Endpoint has failed repeatedly
- **Transition**: Moves to HALF_OPEN after timeout period

HALF_OPEN State

- **Behavior**: Limited number of test requests are allowed
- **Condition**: Testing if endpoint has recovered
- **Transition**:
  - CLOSED if test requests succeed
  - OPEN if test requests fail

4.3 Configuration Parameters

```
interface CircuitBreakerConfig {
 failureThreshold: number;       // Default: 5 failures
 recoveryTimeout: number;        // Default: 30000ms (30s)
 successThreshold: number;       // Default: 2 successes
 halfOpenMaxRequests: number;  // Default: 3 requests
}
```

4.4 Implementation Details

The circuit breaker tracks failures in a sliding time window to prevent transient errors from triggering the circuit. Only sustained failure patterns cause the circuit to open.

Failure Detection Algorithm:

```
if (consecutive_failures >= failureThreshold) {
 state = OPEN
 open_timestamp = now()
}

if (state === OPEN && now() - open_timestamp >= recoveryTimeout) {
 state = HALF_OPEN
  test_requests_allowed = halfOpenMaxRequests
}
```

5. Health Checking System

5.1 Health Check Types

5.1.1 Active Health Checks

Periodic probes sent to endpoints to verify availability:

```
interface HealthCheck {
 method: 'getHealth' | 'getSlot';
 interval: number; // Default: 30000ms (30s)
 timeout: number; // Default: 5000ms (5s)
}
```

The health checker sends `getHealth` or `getSlot` RPC calls and marks endpoints as unhealthy if they fail to respond within the timeout.

5.1.2 Passive Health Monitoring

Tracks request success/failure rates during normal operation:

```
health_status = {
 isHealthy: error_rate < threshold,
 lastCheck: timestamp,
 consecutiveFailures: count
}
```

5.2 Health Score Calculation

```
health_score =
 (successful_requests / total_requests) × 0.7 +
 (response_time_score) × 0.2 +
  (uptime_percentage) × 0.1
```

6. Retry Mechanisms

6.1 Exponential Backoff Algorithm

```
delay = base_delay × (2 ^ attempt) + random_jitter
```

Example Retry Sequence:

1. First retry: 1000ms + jitter
2. Second retry: 2000ms + jitter
3. Third retry: 4000ms + jitter
4. Fourth retry: 8000ms + jitter (capped at maxDelay)

6.2 Error Classification

Errors are classified into three categories:

Retryable Errors:

- Network timeouts
- 429 (Rate Limit Exceeded)
- 500, 502, 503, 504 (Server Errors)
- Connection refused/reset

Non-Retryable Errors:

- 400 (Bad Request)
- 401 (Unauthorized)
- 403 (Forbidden)
- Invalid JSON responses

Fatal Errors:

- Configuration errors
- Invalid endpoint URLs
- Missing API keys

6.3 Retry Configuration

```
interface RetryConfig {
 maxRetries: number;       // Default: 3
 initialDelay: number;     // Default: 1000ms
 maxDelay: number;         // Default: 30000ms
 backoffMultiplier: number; // Default: 2
 jitterFactor: number;     // Default: 0.1
}
```

7. Rate Limiting

7.1 Token Bucket Algorithm

RunicRPC implements a token bucket algorithm for rate limiting:

```
class TokenBucket {
 tokens: number;
 capacity: number;
 refillRate: number; // tokens per second
 lastRefill: number;

 tryConsume(cost: number): boolean {
 this.refill();
 if (this.tokens >= cost) {
     this.tokens -= cost;
     return true;
 }
 return false;
 }
```

```
  refill(): void {
    const now = Date.now();
    const elapsed = (now - this.lastRefill) / 1000;
    const tokensToAdd = elapsed * this.refillRate;
    this.tokens = Math.min(this.capacity, this.tokens + tokensToAdd);
    this.lastRefill = now;
  }
}
```

7.2 Rate Limit Configuration

```
interface RateLimitConfig {
  requestsPerSecond: number; // Default: 100
  burstSize: number;         // Default: 150
}
```

- **requestsPerSecond**: Sustained rate limit
- **burstSize**: Maximum tokens available for bursts

8. Caching Layer

8.1 Cache Strategy

RunicRPC implements a TTL-based cache with LRU eviction:

```
interface CacheEntry<T> {
  value: T;
  expiry: number;
  hits: number;
}
```

8.2 Cacheable Methods

Only idempotent, read-only methods are cached:

- `getAccountInfo`
- `getBalance`

-
- `getBlockHeight`
- `getSlot`

  `getBlock`

  `getTransaction`

8.3 Cache Configuration

```
interface CacheConfig {
 ttl: number; // Default: 5000ms (5s)
 maxSize: number;       // Default: 1000 entries
 enabled: boolean; // Default: true
}
```

8.4 Cache Key Generation

```
cache_key = `${method}:${JSON.stringify(params)}`
```

9. Request Deduplication

9.1 In-Flight Request Tracking

Prevents duplicate requests for identical RPC calls:

```
interface InFlightRequest {
 promise: Promise<any>;
 timestamp: number;
 subscribers: number;
}
```

9.2 Deduplication Algorithm

```
function dedupe<T>(key: string, fn: () => Promise<T>): Promise<T> {
 if (inFlight.has(key)) {
  const request = inFlight.get(key);
```

```
 request.subscribers++;
 return request.promise;
}

const promise = fn();
 inFlight.set(key, { promise, timestamp: Date.now(), subscribers: 1 });

promise.finally(() => {
 inFlight.delete(key);
});

return promise;
}
```

Benefits:

- Reduces provider API calls by 30-60% in high-concurrency scenarios •

Decreases network bandwidth usage

- Prevents rate limit exhaustion

10.WebSocket Support

10.1 Connection Management

```
class WsManager {
 connections: Map<string, WebSocket>;
subscriptions: Map<string, Set<number>>;
 reconnectDelay: number = 1000;
 maxReconnectAttempts: number = 10;
}
```

10.2 Automatic Reconnection

```
async reconnect(endpointId: string): Promise<void> {
 let attempt = 0;
 while (attempt < maxReconnectAttempts) {
 try {
```

```
        await this.connect(endpointId);
        await this.resubscribeAll(endpointId);
        break;
      } catch (error) {
        attempt++;
        const delay = Math.min(
        reconnectDelay * Math.pow(2, attempt),
        30000
        );
        await sleep(delay);
      }
    }
  }
```

10.3 Subscription Management

The system maintains a registry of active subscriptions and automatically resubscribes after reconnection:

```
interface Subscription {
  id: number;
  method: string;
  params: any[];
  callback: (data: any) => void;
}
```

11.Observability

11.1 Event System

RunicRPC emits detailed events for monitoring:

```
enum RunicRPCEvent {
  REQUEST_START = 'request:start',
  REQUEST_SUCCESS = 'request:success',
  REQUEST_ERROR = 'request:error',
  CIRCUIT_OPENED = 'circuit:opened',
```

```
  CIRCUIT_CLOSED = 'circuit:closed',
  HEALTH_CHECK = 'health:check',
  ENDPOINT_FAILED = 'endpoint:failed',
  ENDPOINT_RECOVERED = 'endpoint:recovered'
}
```

11.2 Metrics Collection

```
interface Metrics {
  requests: {
    total: number;
    successful: number;
    failed: number;
    latency: Histogram;
  };
  endpoints: Map<string, EndpointMetrics>;
  cache: {
    hits: number;
    misses: number;
    size: number;
  };
}
```

11.3 Prometheus Export

```
function toPrometheus(): string {
  return `
# HELP runic_rpc_requests_total Total number of RPC requests #
TYPE runic_rpc_requests_total counter
runic_rpc_requests_total{status="success"} ${metrics.requests.successful}
runic_rpc_requests_total{status="error"} ${metrics.requests.failed}

# HELP runic_rpc_request_duration_seconds Request latency in seconds #
TYPE runic_rpc_request_duration_seconds histogram
${metrics.requests.latency.toPrometheus()}

# HELP runic_rpc_circuit_breaker_state Circuit breaker state (0=closed, 1=open, 2=
```

```
# TYPE runic_rpc_circuit_breaker_state gauge
${endpoints.map(e => `runic_rpc_circuit_breaker_state{endpoint="${e.id}"} ${e.circ
`.trim();
}
```

12.Performance Analysis

12.1 Benchmarks

Testing Environment:

- Node.js v20.x
- 16 GB RAM
- Network: 100 Mbps
- 3 RPC providers (Helius, Alchemy, QuickNode)

Throughput:

- **Without RunicRPC**: 850 req/s (single provider)
- **With RunicRPC**: 2,400 req/s (3 providers, round-robin)
- **Improvement**: 2.82x

Latency Overhead:

- **Routing Decision**: 0.05ms (p50), 0.12ms (p99)
- **Circuit Breaker Check**: 0.01ms (p50), 0.03ms (p99)
- **Total Overhead**: 0.08ms (p50), 0.18ms (p99)

Cache Performance:

- **Hit Rate**: 65% (typical workload)
- **Latency Reduction**: 45ms 0.2ms for cached responses

12.2 Resource Usage

- **Memory**: ~15 MB base + ~0.5 KB per endpoint
- **CPU**: <1% idle, <5% under heavy load

- **Network**: Minimal overhead (~100 bytes per request)

12.3 Scalability

RunicRPC scales linearly with the number of endpoints: •

3 endpoints: 2,400 req/s

- 5 endpoints: 4,000 req/s
- 10 endpoints: 8,000 req/s

13.Security Considerations

13.1 API Key Protection

All API keys are:

- Masked in logs (only first 4 and last 4 characters visible) •

Never stored in plain text in memory after configuration •

Transmitted only over HTTPS

- Not included in error messages or stack traces

13.2 Request Validation

Input validation prevents:

- JSON injection attacks
- Oversized payloads (max 1 MB)
- Invalid method names
- Malformed parameters

13.3 Rate Limiting Protection

Protects against:

- DoS attacks via request flooding
- Accidental resource exhaustion •

Provider rate limit violations

13.4 Error Information Disclosure

Error messages never include:

- Full API keys
- Internal file paths
- Stack traces in production
- Sensitive configuration details

14.Production Deployment

14.1 Recommended Configuration

```
// Set API keys in .env file:
// HELIUS_API_KEY=your-key
// ALCHEMY_API_KEY=your-key
//  QUICKNODE_RPC_URL=https://your-endpoint.quiknode.pro/token/

const runicRpc = RunicRPC.create({
 strategy: 'latency-based', // Best for production
 circuitBreaker: {
  failureThreshold: 5,
  timeout: 30000,
 },
 retry: {
  maxAttempts: 3,
  initialDelay: 1000,
 },
 cache: {
  enabled: true,
  ttl: 5000,
 },
 healthCheck: {
  enabled: true,
  interval: 30000,
 },
});
```

14.2 Monitoring Setup

1. **Export Prometheus Metrics**:

```
const metrics = runicRpc.getMetrics().toPrometheus();
// Expose on /metrics endpoint
```

2. **Set Up Alerts**:

- Circuit breaker open for > 5 minutes
- Error rate > 5%
- Average latency > 1000ms
- Cache hit rate < 40%

3. **Log Integration**:

```
runicRpc.on('request:error', (event) => {
 logger.error('RPC request failed', {
 endpoint: event.endpointId,
 method: event.method,
 error: event.error,
 });
});
```

14.3 High Availability Configuration

For mission-critical applications:

```
// Set API keys in .env file
const runicRpc = RunicRPC.create({
 endpoints: [
 // Primary tier (premium providers - auto-loaded from env)
 // HELIUS_API_KEY and ALCHEMY_API_KEY

 // Fallback tier (public endpoints)
 createPublicEndpoint({ weight: 1 }),
 ],
```

```
  strategy: 'latency-based',
  useFallback: true, // Add public endpoint as last resort
  circuitBreaker: {
  failureThreshold: 3,      // More sensitive
  timeout: 60000, // Longer recovery period
  },
  retry: {
  maxAttempts: 5, // More retry attempts
  maxDelay: 60000,
  },
});
```

15.Future Roadmap

15.1 Q1 2026

- Enhanced observability dashboard
- Advanced retry policies with custom error handlers •

Custom routing strategy API

- Performance optimizations for high-frequency trading •

Extended test coverage and chaos engineering

15.2 Q2 2026

- Distributed tracing integration (OpenTelemetry) •

SLA monitoring and alerting

- Request/response middleware hooks •

Plugin system for extensibility

- Cloud-hosted managed option

15.3 Q3 2026

- Multi-region support with geographic routing •

Cost optimization analytics

- Automatic provider selection based on pricing

- GraphQL API support
- Enterprise features (SSO, audit logs, RBAC)

16. Conclusion

RunicRPC addresses the critical challenges of Solana RPC infrastructure through intelligent routing, comprehensive failure handling, and production-grade observability. The system's zero-dependency architecture, sub-millisecond routing overhead, and automatic failover capabilities make it suitable for both development and high-scale production deployments.

Key achievements:

- **2.82x throughput improvement** over single-provider configurations
- **99.99% uptime** through circuit breaking and automatic failover
- **65% cache hit rate** reducing latency and provider costs
- **<0.1ms routing overhead** maintaining near-native performance

The project is open-source (MIT license) and actively maintained on GitHub. Community contributions, feature requests, and production feedback are welcome.

17. References

17.1 Technical Papers

- Nygard, M. (2007). Release It!: Circuit Breaker Pattern
- Kleppmann, M. (2017). Designing Data-Intensive Applications
- Burns, B., & Oppenheimer, D. (2016). Design Patterns for Container-based Distributed Systems

17.2 Industry Standards

- OpenTelemetry Specification v1.x •

Prometheus Exposition Format

- JSON-RPC 2.0 Specification

17.3 Solana Documentation

- Solana RPC API Reference: https://docs.solana.com/api
- Solana Web3.js Documentation: https://solana-labs.github.io/solana-web3.js/

Appendix A: Configuration Reference

Complete configuration options:

```
interface RunicRPCConfig {
 endpoints: Endpoint[];
 routingStrategy?: 'round-robin' | 'latency-based' | 'weighted' | 'random';
 circuitBreaker?: {
 enabled?: boolean;
 failureThreshold?: number;
 recoveryTimeout?: number;
 successThreshold?: number;
 halfOpenMaxRequests?: number;
 };
 retry?: {
 enabled?: boolean;
 maxRetries?: number;
 initialDelay?: number;
 maxDelay?: number;
 backoffMultiplier?: number;
 jitterFactor?: number;
 };
 cache?: {
 enabled?: boolean;
 ttl?: number;
 maxSize?: number;
 };
 healthCheck?: {
 enabled?: boolean;
 interval?: number;
 timeout?: number;
 };
 rateLimit?: {
 enabled?: boolean;
 requestsPerSecond?: number;
```

```
  burstSize?: number;
};
  deduplication?: {
  enabled?: boolean;
  window?: number;
};
logging?: {
  level?: 'debug' | 'info' | 'warn' | 'error';
  maskApiKeys?: boolean;
};
}
```

Appendix B: Error Codes

Code	Name	Description	Retry
E001	NetworkTimeout	Request timed out	Yes
E002	RateLimitExceeded	Provider rate limit hit	Yes
E003	CircuitBreakerOpen	Circuit breaker protecting endpoint	No
E004	InvalidConfiguration	Configuration error	No
E005	AllEndpointsFailed	No healthy endpoints available	No
E006	InvalidRequest	Malformed RPC request	No
E007	AuthenticationFailed	Invalid API key	No
E008	InternalError	Provider internal error	Yes

Contact: https://github.com/RunicRPC/runic-rpc

License: MIT

Version: 1.0

Last Updated: January 2026