API Integration Guide

Jupiter Swap DApp

Implementation Guide

Comprehensive API Integration Documentation

Jupiter API: v6 Integration
Helius RPC: Advanced Features
Alchemy: Backup & Scaling
Coingecko: Market Data
Sentry: Error Monitoring
Custom APIs: Internal Services
WebSocket: Real-time Updates
GraphQL: Advanced Queries

Integration Highlights

Jupiter API v6 Complete Integration
Multi-RPC Endpoint Management
Advanced Error Handling Patterns
Performance Optimization Strategies
Security Best Practices
Rate Limiting & Caching
Real-time Data Streaming
Comprehensive Testing Coverage

Author: Kamel (@treizeb__)
Company: DeAura.io
Updated: July 14, 2025

Contents

1 Jupiter API v6 Integration

1.1 Complete Jupiter Service Implementation

```
/**
 * Jupiter API v6 Service - Complete Implementation
 * Handles all Jupiter API interactions with advanced features
 */
export class JupiterService extends BaseService {
  private readonly apiBase: string;
  private readonly version: string;
  private readonly httpClient: HttpClient;
  private readonly rateLimiter: RateLimiter;
  private readonly cache: CacheManager;

  constructor(
    logger: Logger,
    errorHandler: ErrorHandler,
    httpClient: HttpClient,
    rateLimiter: RateLimiter,
    cache: CacheManager
  ) {
    super(logger, errorHandler);
    this.apiBase = this.validateRequired(
      process.env.NEXT_PUBLIC_JUPITER_API_BASE,
      'NEXT_PUBLIC_JUPITER_API_BASE'
    );
    this.version = process.env.NEXT_PUBLIC_JUPITER_API_VERSION || 'v6';
    this.httpClient = httpClient;
    this.rateLimiter = rateLimiter;
    this.cache = cache;
  }

  /**
   * Get optimized quote with advanced parameters
   * @param params - Enhanced quote parameters
   * @returns Promise<JupiterQuote>
   */
  async getQuote(params: EnhancedQuoteParams): Promise<JupiterQuote> {
    this.logger.info('Fetching Jupiter quote', { params });

    // Rate limiting
    await this.rateLimiter.waitForToken('jupiter-quote');

    // Cache key generation
    const cacheKey = this.generateQuoteCacheKey(params);

    // Check cache first
    const cachedQuote = await this.cache.get<JupiterQuote>(cacheKey);
    if (cachedQuote && !this.isQuoteStale(cachedQuote)) {
      this.logger.debug('Returning cached quote', { cacheKey });
      return cachedQuote;
    }

    return this.executeWithRetry(async () => {
      const queryParams = this.buildAdvancedQuoteParams(params);

      const response = await this.httpClient.get(
        `${this.apiBase}/${this.version}/quote`,
        {
          params: queryParams,
          timeout: 10000,
          headers: {
```

```
            'User-Agent': 'Jupiter-Swap-DApp/1.0',
            'Accept-Encoding': 'gzip, deflate',
          }
        }
      );

      const quote = this.validateAndEnhanceQuote(response.data, params);

      // Cache the result
      await this.cache.set(cacheKey, quote, 15000); // 15 seconds TTL

      return quote;
    }, 3, 1000);
  }

  /**
   * Get swap transaction with advanced options
   * @param quote - Jupiter quote
   * @param userPublicKey - User's wallet public key
   * @param options - Advanced swap options
   * @returns Promise<SwapTransaction>
   */
  async getSwapTransaction(
    quote: JupiterQuote,
    userPublicKey: PublicKey,
    options: AdvancedSwapOptions = {}
  ): Promise<SwapTransaction> {
    this.logger.info('Getting swap transaction', {
      quote: quote.inputMint + '->' + quote.outputMint,
      userPublicKey: userPublicKey.toString(),
      options
    });

    // Rate limiting
    await this.rateLimiter.waitForToken('jupiter-swap');

    return this.executeWithRetry(async () => {
      const swapRequest = this.buildAdvancedSwapRequest(quote, userPublicKey, options
    );

      const response = await this.httpClient.post(
        `${this.apiBase}/${this.version}/swap`,
        swapRequest,
        {
          timeout: 15000,
          headers: {
            'Content-Type': 'application/json',
            'User-Agent': 'Jupiter-Swap-DApp/1.0',
          }
        }
      );

      return this.validateSwapTransaction(response.data);
    }, 2, 2000);
  }

  /**
   * Get supported tokens with metadata
   * @returns Promise<TokenInfo[]>
   */
  async getSupportedTokens(): Promise<TokenInfo[]> {
    const cacheKey = 'jupiter-tokens';

```

```
    // Check cache (longer TTL for token list)
    const cachedTokens = await this.cache.get<TokenInfo[]>(cacheKey);
    if (cachedTokens) {
      return cachedTokens;
    }

    return this.executeWithRetry(async () => {
      const response = await this.httpClient.get(
        `${this.apiBase}/${this.version}/tokens`,
        { timeout: 30000 }
      );

      const tokens = this.validateTokenList(response.data);

      // Cache for 1 hour
      await this.cache.set(cacheKey, tokens, 3600000);

      return tokens;
    });
  }

  /**
   * Get route map for advanced routing
   * @returns Promise<RouteMap>
   */
  async getRouteMap(): Promise<RouteMap> {
    const cacheKey = 'jupiter-route-map';

    const cachedRouteMap = await this.cache.get<RouteMap>(cacheKey);
    if (cachedRouteMap) {
      return cachedRouteMap;
    }

    return this.executeWithRetry(async () => {
      const response = await this.httpClient.get(
        `${this.apiBase}/${this.version}/route-map`,
        { timeout: 30000 }
      );

      const routeMap = this.validateRouteMap(response.data);

      // Cache for 30 minutes
      await this.cache.set(cacheKey, routeMap, 1800000);

      return routeMap;
    });
  }

  /**
   * Get price impact analysis
   * @param params - Price impact parameters
   * @returns Promise<PriceImpactAnalysis>
   */
  async getPriceImpactAnalysis(
    params: PriceImpactParams
  ): Promise<PriceImpactAnalysis> {
    // Get multiple quotes with different amounts to analyze price impact
    const amounts = [
      params.baseAmount * 0.1,
      params.baseAmount * 0.5,
      params.baseAmount,
      params.baseAmount * 2,
      params.baseAmount * 5,
```

```
    ];

    const quotes = await Promise.all(
      amounts.map(amount =>
        this.getQuote({
          ...params,
          amount: Math.floor(amount),
        }).catch(error => {
          this.logger.warn(`Failed to get quote for amount ${amount}:`, error);
          return null;
        })
      )
    );

    return this.analyzePriceImpact(quotes.filter(Boolean) as JupiterQuote[], params);
  }

  /**
   * Build advanced quote parameters
   */
  private buildAdvancedQuoteParams(params: EnhancedQuoteParams): Record<string,
    string> {
    const baseParams = {
      inputMint: params.inputMint,
      outputMint: params.outputMint,
      amount: params.amount.toString(),
      slippageBps: (params.slippageBps || 50).toString(),
      feeBps: (params.feeBps || 0).toString(),
      onlyDirectRoutes: (params.onlyDirectRoutes || false).toString(),
      asLegacyTransaction: 'false',
      platformFeeBps: '25', // 0.25% platform fee
      maxAccounts: '64',
    };

    // Add advanced parameters
    if (params.excludeDexes && params.excludeDexes.length > 0) {
      baseParams['excludeDexes'] = params.excludeDexes.join(',');
    }

    if (params.onlyDirectRoutes !== undefined) {
      baseParams['onlyDirectRoutes'] = params.onlyDirectRoutes.toString();
    }

    if (params.maxSplits !== undefined) {
      baseParams['maxSplits'] = params.maxSplits.toString();
    }

    if (params.minimizeSlippage) {
      baseParams['minimizeSlippage'] = 'true';
    }

    return baseParams;
  }

  /**
   * Build advanced swap request
   */
  private buildAdvancedSwapRequest(
    quote: JupiterQuote,
    userPublicKey: PublicKey,
    options: AdvancedSwapOptions
  ): SwapRequest {
    const baseRequest = {
```

```
      quoteResponse: quote,
      userPublicKey: userPublicKey.toString(),
      wrapAndUnwrapSol: true,
      useSharedAccounts: true,
      feeAccount: options.feeAccount,
      trackingAccount: options.trackingAccount,
      computeUnitPriceMicroLamports: options.priorityFee || 'auto',
      asLegacyTransaction: false,
      useTokenLedger: false,
      destinationTokenAccount: options.destinationTokenAccount,
    };

    // Add advanced options
    if (options.dynamicComputeUnitLimit) {
      baseRequest['dynamicComputeUnitLimit'] = true;
    }

    if (options.skipUserAccountsRpcCalls) {
      baseRequest['skipUserAccountsRpcCalls'] = true;
    }

    return baseRequest;
  }

  /**
   * Validate and enhance quote response
   */
  private validateAndEnhanceQuote(
    data: any,
    params: EnhancedQuoteParams
  ): JupiterQuote {
    if (!data.inputMint || !data.outputMint || !data.inAmount || !data.outAmount) {
      throw new ValidationError('Invalid quote response structure');
    }

    // Enhance quote with additional metadata
    const enhancedQuote: JupiterQuote = {
      ...data,
      timestamp: Date.now(),
      requestParams: params,
      priceImpact: this.calculatePriceImpact(data),
      estimatedGas: this.estimateGasCost(data),
      routeQuality: this.assessRouteQuality(data),
    };

    return enhancedQuote;
  }

  /**
   * Generate cache key for quote
   */
  private generateQuoteCacheKey(params: EnhancedQuoteParams): string {
    const keyData = {
      inputMint: params.inputMint,
      outputMint: params.outputMint,
      amount: params.amount,
      slippageBps: params.slippageBps || 50,
      excludeDexes: params.excludeDexes?.sort() || [],
    };

    return `jupiter-quote:${Buffer.from(JSON.stringify(keyData)).toString('base64')
    }`;
  }
```

```
309
310   /**
311    * Check if quote is stale
312    */
313   private isQuoteStale(quote: JupiterQuote): boolean {
314     const maxAge = 15000; // 15 seconds
315     return Date.now() - quote.timestamp > maxAge;
316   }
317
318   /**
319    * Calculate price impact
320    */
321   private calculatePriceImpact(quote: any): number {
322     // Implementation depends on quote structure
323     // This is a simplified calculation
324     const inputAmount = parseFloat(quote.inAmount);
325     const outputAmount = parseFloat(quote.outAmount);
326     const marketPrice = quote.marketPrice || (outputAmount / inputAmount);
327
328     const executionPrice = outputAmount / inputAmount;
329     const priceImpact = ((marketPrice - executionPrice) / marketPrice) * 100;
330
331     return Math.max(0, priceImpact);
332   }
333
334   /**
335    * Estimate gas cost
336    */
337   private estimateGasCost(quote: any): number {
338     // Base gas cost estimation based on route complexity
339     const baseGas = 5000; // Base transaction cost
340     const routeComplexity = quote.routePlan?.length || 1;
341     const complexityMultiplier = Math.min(routeComplexity * 1000, 10000);
342
343     return baseGas + complexityMultiplier;
344   }
345
346   /**
347    * Assess route quality
348    */
349   private assessRouteQuality(quote: any): 'excellent' | 'good' | 'fair' | 'poor' {
350     const priceImpact = this.calculatePriceImpact(quote);
351     const routeLength = quote.routePlan?.length || 1;
352
353     if (priceImpact < 0.1 && routeLength <= 2) return 'excellent';
354     if (priceImpact < 0.5 && routeLength <= 3) return 'good';
355     if (priceImpact < 1.0 && routeLength <= 4) return 'fair';
356     return 'poor';
357   }
358
359   /**
360    * Analyze price impact across different amounts
361    */
362   private analyzePriceImpact(
363     quotes: JupiterQuote[],
364     params: PriceImpactParams
365   ): PriceImpactAnalysis {
366     const analysis: PriceImpactAnalysis = {
367       baseAmount: params.baseAmount,
368       quotes: quotes.map(quote => ({
369         amount: parseFloat(quote.inAmount),
370         priceImpact: quote.priceImpact || 0,
371         outputAmount: parseFloat(quote.outAmount),
```

```
        rate: parseFloat(quote.outAmount) / parseFloat(quote.inAmount),
      })),
      recommendations: [],
    };

    // Generate recommendations based on analysis
    const impacts = analysis.quotes.map(q => q.priceImpact);
    const avgImpact = impacts.reduce((a, b) => a + b, 0) / impacts.length;

    if (avgImpact > 2.0) {
      analysis.recommendations.push('Consider splitting large trades into smaller chunks');
    }

    if (impacts[impacts.length - 1] > impacts[0] * 3) {
      analysis.recommendations.push('Price impact increases significantly with larger amounts');
    }

    return analysis;
  }
}
```

Listing 1: Advanced Jupiter API v6 Service

2 Helius RPC Integration

2.1 Advanced Helius Features

```
/**
 * Helius RPC Service - Advanced Integration
 * Leverages Helius-specific features for enhanced performance
 */
export class HeliusService extends BaseService {
  private readonly apiKey: string;
  private readonly endpoint: string;
  private readonly httpClient: HttpClient;
  private readonly websocketManager: WebSocketManager;

  constructor(
    logger: Logger,
    errorHandler: ErrorHandler,
    httpClient: HttpClient,
    websocketManager: WebSocketManager
  ) {
    super(logger, errorHandler);
    this.apiKey = this.validateRequired(
      process.env.NEXT_PUBLIC_HELIUS_API_KEY,
      'NEXT_PUBLIC_HELIUS_API_KEY'
    );
    this.endpoint = `https://mainnet.helius-rpc.com/?api-key=${this.apiKey}`;
    this.httpClient = httpClient;
    this.websocketManager = websocketManager;
  }

  /**
   * Get enhanced account info with Helius metadata
   * @param publicKey - Account public key
   * @returns Promise<EnhancedAccountInfo>
   */
  async getEnhancedAccountInfo(publicKey: PublicKey): Promise<EnhancedAccountInfo> {
```

```
    return this.executeWithRetry(async () => {
      const response = await this.httpClient.post(this.endpoint, {
        jsonrpc: '2.0',
        id: 1,
        method: 'getAccountInfo',
        params: [
          publicKey.toString(),
          {
            encoding: 'jsonParsed',
            commitment: 'confirmed',
          },
        ],
      });

      if (response.data.error) {
        throw new RpcError(response.data.error.message, this.endpoint);
      }

      // Enhance with Helius-specific data
      const accountInfo = response.data.result;
      const enhancedInfo = await this.enhanceAccountInfo(accountInfo, publicKey);

      return enhancedInfo;
    });
  }

  /**
   * Get token accounts with enhanced metadata
   * @param owner - Owner public key
   * @returns Promise<EnhancedTokenAccount[]>
   */
  async getTokenAccountsByOwner(owner: PublicKey): Promise<EnhancedTokenAccount[]> {
    return this.executeWithRetry(async () => {
      const response = await this.httpClient.post(this.endpoint, {
        jsonrpc: '2.0',
        id: 1,
        method: 'getTokenAccountsByOwner',
        params: [
          owner.toString(),
          { programId: TOKEN_PROGRAM_ID.toString() },
          {
            encoding: 'jsonParsed',
            commitment: 'confirmed',
          },
        ],
      });

      if (response.data.error) {
        throw new RpcError(response.data.error.message, this.endpoint);
      }

      const accounts = response.data.result.value;

      // Enhance each account with metadata
      const enhancedAccounts = await Promise.all(
        accounts.map((account: any) => this.enhanceTokenAccount(account))
      );

      return enhancedAccounts;
    });
  }

  /**
```

```
   * Get transaction history with enhanced parsing
   * @param address - Address to get history for
   * @param options - History options
   * @returns Promise<EnhancedTransaction[]>
   */
  async getTransactionHistory(
    address: PublicKey,
    options: TransactionHistoryOptions = {}
  ): Promise<EnhancedTransaction[]> {
    return this.executeWithRetry(async () => {
      const response = await this.httpClient.post(this.endpoint, {
        jsonrpc: '2.0',
        id: 1,
        method: 'getSignaturesForAddress',
        params: [
          address.toString(),
          {
            limit: options.limit || 50,
            before: options.before,
            until: options.until,
            commitment: 'confirmed',
          },
        ],
      });

      if (response.data.error) {
        throw new RpcError(response.data.error.message, this.endpoint);
      }

      const signatures = response.data.result;

      // Get detailed transaction info for each signature
      const transactions = await this.getTransactionDetails(signatures);

      // Parse and enhance transactions
      const enhancedTransactions = await Promise.all(
        transactions.map((tx: any) => this.enhanceTransaction(tx))
      );

      return enhancedTransactions;
    });
  }

  /**
   * Subscribe to account changes via WebSocket
   * @param publicKey - Account to monitor
   * @param callback - Callback for updates
   * @returns Subscription ID
   */
  async subscribeToAccount(
    publicKey: PublicKey,
    callback: (accountInfo: EnhancedAccountInfo) => void
  ): Promise<string> {
    const wsEndpoint = this.endpoint.replace('https://', 'wss://').replace('http://',
     'ws://');

    const subscriptionId = await this.websocketManager.subscribe(
      wsEndpoint,
      'accountSubscribe',
      [
        publicKey.toString(),
        {
          encoding: 'jsonParsed',
```

```
          commitment: 'confirmed',
        },
      ],
      async (data: any) => {
        const enhancedInfo = await this.enhanceAccountInfo(data.result, publicKey);
        callback(enhancedInfo);
      }
    );

    this.logger.info('Subscribed to account updates', {
      publicKey: publicKey.toString(),
      subscriptionId
    });

    return subscriptionId;
  }

  /**
   * Get DAS (Digital Asset Standard) API data
   * @param assetId - Asset ID
   * @returns Promise<DASAsset>
   */
  async getDASAsset(assetId: string): Promise<DASAsset> {
    const dasEndpoint = `https://mainnet.helius-rpc.com/?api-key=${this.apiKey}`;

    return this.executeWithRetry(async () => {
      const response = await this.httpClient.post(dasEndpoint, {
        jsonrpc: '2.0',
        id: 1,
        method: 'getAsset',
        params: {
          id: assetId,
        },
      });

      if (response.data.error) {
        throw new RpcError(response.data.error.message, dasEndpoint);
      }

      return response.data.result;
    });
  }

  /**
   * Get assets by owner using DAS API
   * @param owner - Owner public key
   * @param options - Query options
   * @returns Promise<DASAsset[]>
   */
  async getAssetsByOwner(
    owner: PublicKey,
    options: AssetQueryOptions = {}
  ): Promise<DASAsset[]> {
    const dasEndpoint = `https://mainnet.helius-rpc.com/?api-key=${this.apiKey}`;

    return this.executeWithRetry(async () => {
      const response = await this.httpClient.post(dasEndpoint, {
        jsonrpc: '2.0',
        id: 1,
        method: 'getAssetsByOwner',
        params: {
          ownerAddress: owner.toString(),
          page: options.page || 1,
```

```
          limit: options.limit || 1000,
          displayOptions: {
            showFungible: options.showFungible || false,
            showNativeBalance: options.showNativeBalance || true,
          },
        },
      });

      if (response.data.error) {
        throw new RpcError(response.data.error.message, dasEndpoint);
      }

      return response.data.result.items;
    });
  }

  /**
   * Enhance account info with metadata
   */
  private async enhanceAccountInfo(
    accountInfo: any,
    publicKey: PublicKey
  ): Promise<EnhancedAccountInfo> {
    const enhanced: EnhancedAccountInfo = {
      ...accountInfo,
      publicKey: publicKey.toString(),
      enhancedAt: Date.now(),
    };

    // Add token metadata if it's a token account
    if (accountInfo?.value?.data?.parsed?.type === 'account') {
      const tokenMint = accountInfo.value.data.parsed.info.mint;
      enhanced.tokenMetadata = await this.getTokenMetadata(tokenMint);
    }

    return enhanced;
  }

  /**
   * Enhance token account with metadata
   */
  private async enhanceTokenAccount(account: any): Promise<EnhancedTokenAccount> {
    const enhanced: EnhancedTokenAccount = {
      ...account,
      enhancedAt: Date.now(),
    };

    const tokenMint = account.account.data.parsed.info.mint;
    enhanced.tokenMetadata = await this.getTokenMetadata(tokenMint);

    return enhanced;
  }

  /**
   * Get token metadata
   */
  private async getTokenMetadata(mint: string): Promise<TokenMetadata | null> {
    try {
      const asset = await this.getDASAsset(mint);

      return {
        name: asset.content?.metadata?.name || 'Unknown Token',
        symbol: asset.content?.metadata?.symbol || 'UNKNOWN',
```

```
        decimals: asset.token_info?.decimals || 0,
        logoURI: asset.content?.files?.[0]?.uri,
        description: asset.content?.metadata?.description,
        tags: asset.grouping?.map(g => g.group_value) || [],
      };
    } catch (error) {
      this.logger.warn(`Failed to get metadata for token ${mint}:`, error);
      return null;
    }
  }

  /**
   * Get detailed transaction information
   */
  private async getTransactionDetails(signatures: any[]): Promise<any[]> {
    const batchSize = 10;
    const batches = [];

    for (let i = 0; i < signatures.length; i += batchSize) {
      const batch = signatures.slice(i, i + batchSize);
      batches.push(batch);
    }

    const allTransactions = [];

    for (const batch of batches) {
      const batchPromises = batch.map(sig =>
        this.httpClient.post(this.endpoint, {
          jsonrpc: '2.0',
          id: 1,
          method: 'getTransaction',
          params: [
            sig.signature,
            {
              encoding: 'jsonParsed',
              commitment: 'confirmed',
              maxSupportedTransactionVersion: 0,
            },
          ],
        })
      );

      const batchResults = await Promise.allSettled(batchPromises);
      const validResults = batchResults
        .filter(result => result.status === 'fulfilled')
        .map(result => (result as PromiseFulfilledResult<any>).value.data.result)
        .filter(result => result && !result.error);

      allTransactions.push(...validResults);
    }

    return allTransactions;
  }

  /**
   * Enhance transaction with parsed data
   */
  private async enhanceTransaction(transaction: any): Promise<EnhancedTransaction> {
    const enhanced: EnhancedTransaction = {
      signature: transaction.transaction.signatures[0],
      slot: transaction.slot,
      blockTime: transaction.blockTime,
      confirmationStatus: transaction.confirmationStatus,
```

```
      err: transaction.meta?.err,
      fee: transaction.meta?.fee,
      enhancedAt: Date.now(),
      parsedInstructions: [],
      tokenTransfers: [],
      solTransfers: [],
    };

    // Parse instructions
    if (transaction.transaction?.message?.instructions) {
      enhanced.parsedInstructions = await this.parseInstructions(
        transaction.transaction.message.instructions
      );
    }

    // Extract token transfers
    if (transaction.meta?.postTokenBalances && transaction.meta?.preTokenBalances) {
      enhanced.tokenTransfers = this.extractTokenTransfers(
        transaction.meta.preTokenBalances,
        transaction.meta.postTokenBalances
      );
    }

    // Extract SOL transfers
    if (transaction.meta?.postBalances && transaction.meta?.preBalances) {
      enhanced.solTransfers = this.extractSolTransfers(
        transaction.meta.preBalances,
        transaction.meta.postBalances,
        transaction.transaction.message.accountKeys
      );
    }

    return enhanced;
  }

  /**
   * Parse transaction instructions
   */
  private async parseInstructions(instructions: any[]): Promise<ParsedInstruction[]>
    {
    return instructions.map(instruction => ({
      programId: instruction.programId,
      parsed: instruction.parsed || null,
      program: instruction.program || 'unknown',
      type: instruction.parsed?.type || 'unknown',
      info: instruction.parsed?.info || {},
    }));
  }

  /**
   * Extract token transfers from balance changes
   */
  private extractTokenTransfers(
    preBalances: any[],
    postBalances: any[]
  ): TokenTransfer[] {
    const transfers: TokenTransfer[] = [];

    // Create maps for easier lookup
    const preMap = new Map(preBalances.map(b => [b.accountIndex, b]));
    const postMap = new Map(postBalances.map(b => [b.accountIndex, b]));

    // Find all account indices that had balance changes
```

```
    const allIndices = new Set([...preMap.keys(), ...postMap.keys()]);

    for (const index of allIndices) {
      const pre = preMap.get(index);
      const post = postMap.get(index);

      if (pre && post && pre.mint === post.mint) {
        const preAmount = parseFloat(pre.uiTokenAmount.amount);
        const postAmount = parseFloat(post.uiTokenAmount.amount);
        const change = postAmount - preAmount;

        if (change !== 0) {
          transfers.push({
            mint: pre.mint,
            amount: Math.abs(change),
            decimals: pre.uiTokenAmount.decimals,
            direction: change > 0 ? 'in' : 'out',
            accountIndex: index,
          });
        }
      }
    }

    return transfers;
  }

  /**
   * Extract SOL transfers from balance changes
   */
  private extractSolTransfers(
    preBalances: number[],
    postBalances: number[],
    accountKeys: string[]
  ): SolTransfer[] {
    const transfers: SolTransfer[] = [];

    for (let i = 0; i < preBalances.length; i++) {
      const change = postBalances[i] - preBalances[i];
      if (change !== 0) {
        transfers.push({
          account: accountKeys[i],
          amount: Math.abs(change),
          direction: change > 0 ? 'in' : 'out',
        });
      }
    }

    return transfers;
  }
}
```

Listing 2: Helius RPC Service with Advanced Features

3 Alchemy Integration

3.1 Alchemy as Backup and Scaling Solution

```
/**
 * Alchemy Service - Backup RPC and Enhanced Features
 * Provides backup RPC functionality and Alchemy-specific features
 */
```

```
export class AlchemyService extends BaseService {
  private readonly apiKey: string;
  private readonly endpoint: string;
  private readonly httpClient: HttpClient;

  constructor(
    logger: Logger,
    errorHandler: ErrorHandler,
    httpClient: HttpClient
  ) {
    super(logger, errorHandler);
    this.apiKey = this.validateRequired(
      process.env.NEXT_PUBLIC_ALCHEMY_API_KEY,
      'NEXT_PUBLIC_ALCHEMY_API_KEY'
    );
    this.endpoint = `https://solana-mainnet.g.alchemy.com/v2/${this.apiKey}`;
    this.httpClient = httpClient;
  }

  /**
   * Get enhanced token balances using Alchemy's optimized endpoints
   * @param owner - Owner public key
   * @returns Promise<AlchemyTokenBalance[]>
   */
  async getTokenBalances(owner: PublicKey): Promise<AlchemyTokenBalance[]> {
    return this.executeWithRetry(async () => {
      const response = await this.httpClient.post(this.endpoint, {
        jsonrpc: '2.0',
        id: 1,
        method: 'alchemy_getTokenBalances',
        params: [
          owner.toString(),
          {
            mint: 'all',
          },
        ],
      });

      if (response.data.error) {
        throw new RpcError(response.data.error.message, this.endpoint);
      }

      return response.data.result.tokenBalances.map((balance: any) => ({
        mint: balance.mint,
        amount: balance.amount,
        decimals: balance.decimals,
        uiAmount: balance.uiAmount,
        uiAmountString: balance.uiAmountString,
        enhancedAt: Date.now(),
      }));
    });
  }

  /**
   * Get token metadata using Alchemy's metadata service
   * @param mints - Array of token mints
   * @returns Promise<AlchemyTokenMetadata[]>
   */
  async getTokenMetadata(mints: string[]): Promise<AlchemyTokenMetadata[]> {
    return this.executeWithRetry(async () => {
      const response = await this.httpClient.post(this.endpoint, {
        jsonrpc: '2.0',
        id: 1,
```

```
        method: 'alchemy_getTokenMetadata',
        params: [
          {
            mints: mints,
          },
        ],
      });

      if (response.data.error) {
        throw new RpcError(response.data.error.message, this.endpoint);
      }

      return response.data.result.map((metadata: any) => ({
        mint: metadata.mint,
        name: metadata.name,
        symbol: metadata.symbol,
        decimals: metadata.decimals,
        logoURI: metadata.logoURI,
        tags: metadata.tags || [],
        description: metadata.description,
        enhancedAt: Date.now(),
      }));
    });
  }

  /**
   * Get NFTs owned by address
   * @param owner - Owner public key
   * @param options - Query options
   * @returns Promise<AlchemyNFT[]>
   */
  async getNFTsByOwner(
    owner: PublicKey,
    options: NFTQueryOptions = {}
  ): Promise<AlchemyNFT[]> {
    return this.executeWithRetry(async () => {
      const response = await this.httpClient.post(this.endpoint, {
        jsonrpc: '2.0',
        id: 1,
        method: 'alchemy_getNFTs',
        params: [
          owner.toString(),
          {
            pageKey: options.pageKey,
            limit: options.limit || 100,
            withMetadata: options.withMetadata !== false,
          },
        ],
      });

      if (response.data.error) {
        throw new RpcError(response.data.error.message, this.endpoint);
      }

      return response.data.result.nfts.map((nft: any) => ({
        mint: nft.mint,
        name: nft.name,
        symbol: nft.symbol,
        uri: nft.uri,
        metadata: nft.metadata,
        collection: nft.collection,
        enhancedAt: Date.now(),
      }));
```

```
    });
  }

  /**
   * Get transaction receipts with enhanced parsing
   * @param signatures - Transaction signatures
   * @returns Promise<AlchemyTransactionReceipt[]>
   */
  async getTransactionReceipts(
    signatures: string[]
  ): Promise<AlchemyTransactionReceipt[]> {
    const batchSize = 25; // Alchemy's batch limit
    const batches = [];

    for (let i = 0; i < signatures.length; i += batchSize) {
      const batch = signatures.slice(i, i + batchSize);
      batches.push(batch);
    }

    const allReceipts = [];

    for (const batch of batches) {
      const batchRequests = batch.map((signature, index) => ({
        jsonrpc: '2.0',
        id: index + 1,
        method: 'getTransaction',
        params: [
          signature,
          {
            encoding: 'jsonParsed',
            commitment: 'confirmed',
            maxSupportedTransactionVersion: 0,
          },
        ],
      }));

      const response = await this.httpClient.post(this.endpoint, batchRequests);

      if (Array.isArray(response.data)) {
        const validReceipts = response.data
          .filter(result => result.result && !result.error)
          .map(result => this.enhanceTransactionReceipt(result.result));

        allReceipts.push(...validReceipts);
      }
    }

    return allReceipts;
  }

  /**
   * Get account activity with Alchemy's enhanced parsing
   * @param address - Address to get activity for
   * @param options - Activity options
   * @returns Promise<AlchemyActivity[]>
   */
  async getAccountActivity(
    address: PublicKey,
    options: ActivityOptions = {}
  ): Promise<AlchemyActivity[]> {
    return this.executeWithRetry(async () => {
      const response = await this.httpClient.post(this.endpoint, {
        jsonrpc: '2.0',
```

```
        id: 1,
        method: 'alchemy_getAccountActivity',
        params: [
          address.toString(),
          {
            limit: options.limit || 50,
            before: options.before,
            after: options.after,
            activityTypes: options.activityTypes || ['TRANSFER', 'SWAP', 'BURN', 'MINT'],
          },
        ],
      });

      if (response.data.error) {
        throw new RpcError(response.data.error.message, this.endpoint);
      }

      return response.data.result.activities.map((activity: any) => ({
        signature: activity.signature,
        type: activity.type,
        source: activity.source,
        timestamp: activity.timestamp,
        nativeTransfers: activity.nativeTransfers || [],
        tokenTransfers: activity.tokenTransfers || [],
        description: activity.description,
        enhancedAt: Date.now(),
      }));
    });
  }

  /**
   * Enhance transaction receipt with additional parsing
   */
  private enhanceTransactionReceipt(transaction: any): AlchemyTransactionReceipt {
    return {
      signature: transaction.transaction.signatures[0],
      slot: transaction.slot,
      blockTime: transaction.blockTime,
      confirmationStatus: transaction.confirmationStatus,
      err: transaction.meta?.err,
      fee: transaction.meta?.fee,
      computeUnitsConsumed: transaction.meta?.computeUnitsConsumed,
      logMessages: transaction.meta?.logMessages || [],
      preBalances: transaction.meta?.preBalances || [],
      postBalances: transaction.meta?.postBalances || [],
      preTokenBalances: transaction.meta?.preTokenBalances || [],
      postTokenBalances: transaction.meta?.postTokenBalances || [],
      innerInstructions: transaction.meta?.innerInstructions || [],
      enhancedAt: Date.now(),
    };
  }
}
```

Listing 3: Alchemy Service Implementation

4 CoinGecko Market Data Integration

4.1 Market Data Service

```
/**
```

```
 * CoinGecko Service - Market Data Integration
 * Provides comprehensive market data for optimization algorithms
 */
export class CoingeckoService extends BaseService {
  private readonly apiBase: string;
  private readonly apiKey?: string;
  private readonly httpClient: HttpClient;
  private readonly cache: CacheManager;
  private readonly rateLimiter: RateLimiter;

  constructor(
    logger: Logger,
    errorHandler: ErrorHandler,
    httpClient: HttpClient,
    cache: CacheManager,
    rateLimiter: RateLimiter
  ) {
    super(logger, errorHandler);
    this.apiBase = 'https://api.coingecko.com/api/v3';
    this.apiKey = process.env.NEXT_PUBLIC_COINGECKO_API_KEY;
    this.httpClient = httpClient;
    this.cache = cache;
    this.rateLimiter = rateLimiter;
  }

  /**
   * Get token market data
   * @param tokenId - CoinGecko token ID
   * @returns Promise<TokenMarketData>
   */
  async getTokenMarketData(tokenId: string): Promise<TokenMarketData> {
    const cacheKey = `coingecko-market-${tokenId}`;

    // Check cache first (5 minutes TTL)
    const cached = await this.cache.get<TokenMarketData>(cacheKey);
    if (cached) {
      return cached;
    }

    // Rate limiting
    await this.rateLimiter.waitForToken('coingecko-api');

    return this.executeWithRetry(async () => {
      const headers: Record<string, string> = {};
      if (this.apiKey) {
        headers['X-CG-Pro-API-Key'] = this.apiKey;
      }

      const response = await this.httpClient.get(
        `${this.apiBase}/coins/${tokenId}`,
        {
          params: {
            localization: false,
            tickers: false,
            market_data: true,
            community_data: false,
            developer_data: false,
            sparkline: true,
          },
          headers,
          timeout: 10000,
        }
      );
```

```
65
66        const marketData = this.parseMarketData(response.data);
67
68        // Cache for 5 minutes
69        await this.cache.set(cacheKey, marketData, 300000);
70
71        return marketData;
72      });
73    }
74
75    /**
76     * Get multiple tokens market data
77     * @param tokenIds - Array of CoinGecko token IDs
78     * @returns Promise<TokenMarketData[]>
79     */
80    async getMultipleTokensMarketData(tokenIds: string[]): Promise<TokenMarketData[]> {
81      // Rate limiting
82      await this.rateLimiter.waitForToken('coingecko-api');
83
84      return this.executeWithRetry(async () => {
85        const headers: Record<string, string> = {};
86        if (this.apiKey) {
87          headers['X-CG-Pro-API-Key'] = this.apiKey;
88        }
89
90        const response = await this.httpClient.get(
91          `${this.apiBase}/coins/markets`,
92          {
93            params: {
94              vs_currency: 'usd',
95              ids: tokenIds.join(','),
96              order: 'market_cap_desc',
97              per_page: 250,
98              page: 1,
99              sparkline: true,
100             price_change_percentage: '1h,24h,7d,30d',
101           },
102           headers,
103           timeout: 15000,
104         }
105       );
106
107       return response.data.map((coin: any) => this.parseMarketDataFromList(coin));
108     });
109   }
110
111   /**
112    * Get token price history
113    * @param tokenId - CoinGecko token ID
114    * @param days - Number of days of history
115    * @returns Promise<PriceHistory>
116    */
117   async getTokenPriceHistory(tokenId: string, days: number = 7): Promise<PriceHistory
       > {
118     const cacheKey = `coingecko-history-${tokenId}-${days}`;
119
120     // Check cache (longer TTL for historical data)
121     const cached = await this.cache.get<PriceHistory>(cacheKey);
122     if (cached) {
123       return cached;
124     }
125
126     // Rate limiting
```

```
    await this.rateLimiter.waitForToken('coingecko-api');

    return this.executeWithRetry(async () => {
      const headers: Record<string, string> = {};
      if (this.apiKey) {
        headers['X-CG-Pro-API-Key'] = this.apiKey;
      }

      const response = await this.httpClient.get(
        `${this.apiBase}/coins/${tokenId}/market_chart`,
        {
          params: {
            vs_currency: 'usd',
            days: days.toString(),
            interval: days <= 1 ? 'hourly' : 'daily',
          },
          headers,
          timeout: 15000,
        }
      );

      const history = this.parsePriceHistory(response.data);

      // Cache for 1 hour
      await this.cache.set(cacheKey, history, 3600000);

      return history;
    });
  }

  /**
   * Search for tokens
   * @param query - Search query
   * @returns Promise<TokenSearchResult[]>
   */
  async searchTokens(query: string): Promise<TokenSearchResult[]> {
    if (query.length < 2) {
      return [];
    }

    const cacheKey = `coingecko-search-${query.toLowerCase()}`;

    // Check cache (10 minutes TTL)
    const cached = await this.cache.get<TokenSearchResult[]>(cacheKey);
    if (cached) {
      return cached;
    }

    // Rate limiting
    await this.rateLimiter.waitForToken('coingecko-api');

    return this.executeWithRetry(async () => {
      const headers: Record<string, string> = {};
      if (this.apiKey) {
        headers['X-CG-Pro-API-Key'] = this.apiKey;
      }

      const response = await this.httpClient.get(
        `${this.apiBase}/search`,
        {
          params: { query },
          headers,
          timeout: 10000,
```

```
        }
      );

      const results = response.data.coins.slice(0, 20).map((coin: any) => ({
        id: coin.id,
        name: coin.name,
        symbol: coin.symbol,
        thumb: coin.thumb,
        large: coin.large,
        market_cap_rank: coin.market_cap_rank,
      }));

      // Cache for 10 minutes
      await this.cache.set(cacheKey, results, 600000);

      return results;
    });
  }

  /**
   * Get global market data
   * @returns Promise<GlobalMarketData>
   */
  async getGlobalMarketData(): Promise<GlobalMarketData> {
    const cacheKey = 'coingecko-global';

    // Check cache (10 minutes TTL)
    const cached = await this.cache.get<GlobalMarketData>(cacheKey);
    if (cached) {
      return cached;
    }

    // Rate limiting
    await this.rateLimiter.waitForToken('coingecko-api');

    return this.executeWithRetry(async () => {
      const headers: Record<string, string> = {};
      if (this.apiKey) {
        headers['X-CG-Pro-API-Key'] = this.apiKey;
      }

      const response = await this.httpClient.get(
        `${this.apiBase}/global`,
        { headers, timeout: 10000 }
      );

      const globalData = this.parseGlobalData(response.data.data);

      // Cache for 10 minutes
      await this.cache.set(cacheKey, globalData, 600000);

      return globalData;
    });
  }

  /**
   * Parse market data from detailed coin response
   */
  private parseMarketData(data: any): TokenMarketData {
    const marketData = data.market_data;

    return {
      id: data.id,
```

```
      symbol: data.symbol,
      name: data.name,
      image: data.image?.large,
      current_price: marketData.current_price?.usd || 0,
      market_cap: marketData.market_cap?.usd || 0,
      market_cap_rank: data.market_cap_rank,
      fully_diluted_valuation: marketData.fully_diluted_valuation?.usd,
      total_volume: marketData.total_volume?.usd || 0,
      high_24h: marketData.high_24h?.usd || 0,
      low_24h: marketData.low_24h?.usd || 0,
      price_change_24h: marketData.price_change_24h || 0,
      price_change_percentage_24h: marketData.price_change_percentage_24h || 0,
      price_change_percentage_7d: marketData.price_change_percentage_7d || 0,
      price_change_percentage_30d: marketData.price_change_percentage_30d || 0,
      circulating_supply: marketData.circulating_supply || 0,
      total_supply: marketData.total_supply || 0,
      max_supply: marketData.max_supply,
      ath: marketData.ath?.usd || 0,
      ath_change_percentage: marketData.ath_change_percentage?.usd || 0,
      ath_date: marketData.ath_date?.usd,
      atl: marketData.atl?.usd || 0,
      atl_change_percentage: marketData.atl_change_percentage?.usd || 0,
      atl_date: marketData.atl_date?.usd,
      last_updated: data.last_updated,
      sparkline_in_7d: marketData.sparkline_7d?.price || [],
    };
  }

  /**
   * Parse market data from markets list response
   */
  private parseMarketDataFromList(data: any): TokenMarketData {
    return {
      id: data.id,
      symbol: data.symbol,
      name: data.name,
      image: data.image,
      current_price: data.current_price || 0,
      market_cap: data.market_cap || 0,
      market_cap_rank: data.market_cap_rank,
      fully_diluted_valuation: data.fully_diluted_valuation,
      total_volume: data.total_volume || 0,
      high_24h: data.high_24h || 0,
      low_24h: data.low_24h || 0,
      price_change_24h: data.price_change_24h || 0,
      price_change_percentage_24h: data.price_change_percentage_24h || 0,
      price_change_percentage_7d: data.price_change_percentage_7d || 0,
      price_change_percentage_30d: data.price_change_percentage_30d || 0,
      circulating_supply: data.circulating_supply || 0,
      total_supply: data.total_supply || 0,
      max_supply: data.max_supply,
      ath: data.ath || 0,
      ath_change_percentage: data.ath_change_percentage || 0,
      ath_date: data.ath_date,
      atl: data.atl || 0,
      atl_change_percentage: data.atl_change_percentage || 0,
      atl_date: data.atl_date,
      last_updated: data.last_updated,
      sparkline_in_7d: data.sparkline_in_7d?.price || [],
    };
  }

  /**
```

```
   * Parse price history data
   */
  private parsePriceHistory(data: any): PriceHistory {
    return {
      prices: data.prices.map(([timestamp, price]: [number, number]) => ({
        timestamp,
        price,
        date: new Date(timestamp),
      })),
      market_caps: data.market_caps.map(([timestamp, cap]: [number, number]) => ({
        timestamp,
        market_cap: cap,
        date: new Date(timestamp),
      })),
      total_volumes: data.total_volumes.map(([timestamp, volume]: [number, number])
    => ({
        timestamp,
        volume,
        date: new Date(timestamp),
      })),
    };
  }

  /**
   * Parse global market data
   */
  private parseGlobalData(data: any): GlobalMarketData {
    return {
      active_cryptocurrencies: data.active_cryptocurrencies,
      upcoming_icos: data.upcoming_icos,
      ongoing_icos: data.ongoing_icos,
      ended_icos: data.ended_icos,
      markets: data.markets,
      total_market_cap: data.total_market_cap?.usd || 0,
      total_volume: data.total_volume?.usd || 0,
      market_cap_percentage: data.market_cap_percentage,
      market_cap_change_percentage_24h_usd: data.market_cap_change_percentage_24h_usd
     || 0,
      updated_at: data.updated_at,
    };
  }
}
```

Listing 4: CoinGecko Service for Market Data

5 Rate Limiting Caching

5.1 Advanced Rate Limiting Implementation

```
/**
 * Rate Limiter - Token Bucket Algorithm Implementation
 * Provides sophisticated rate limiting for API calls
 */
export class RateLimiter {
  private buckets = new Map<string, TokenBucket>();
  private readonly logger: Logger;

  constructor(logger: Logger) {
    this.logger = logger;
  }

```

```
  /**
   * Configure rate limit for a specific endpoint
   * @param key - Unique identifier for the endpoint
   * @param config - Rate limit configuration
   */
  configure(key: string, config: RateLimitConfig): void {
    this.buckets.set(key, new TokenBucket(config, this.logger));
    this.logger.info('Rate limiter configured', { key, config });
  }

  /**
   * Wait for available token
   * @param key - Endpoint identifier
   * @returns Promise<void>
   */
  async waitForToken(key: string): Promise<void> {
    const bucket = this.buckets.get(key);
    if (!bucket) {
      // No rate limit configured, allow immediately
      return;
    }

    await bucket.consume();
  }

  /**
   * Check if request would be allowed without consuming token
   * @param key - Endpoint identifier
   * @returns boolean
   */
  canMakeRequest(key: string): boolean {
    const bucket = this.buckets.get(key);
    if (!bucket) {
      return true;
    }

    return bucket.hasTokens();
  }

  /**
   * Get current status of rate limiter
   * @param key - Endpoint identifier
   * @returns RateLimitStatus
   */
  getStatus(key: string): RateLimitStatus {
    const bucket = this.buckets.get(key);
    if (!bucket) {
      return {
        configured: false,
        available: Infinity,
        resetTime: null,
      };
    }

    return bucket.getStatus();
  }

  /**
   * Reset rate limiter for specific endpoint
   * @param key - Endpoint identifier
   */
  reset(key: string): void {
    const bucket = this.buckets.get(key);
```

```
    if (bucket) {
      bucket.reset();
      this.logger.info('Rate limiter reset', { key });
    }
  }
}

/**
 * Token Bucket Implementation
 */
class TokenBucket {
  private tokens: number;
  private lastRefill: number;
  private readonly config: RateLimitConfig;
  private readonly logger: Logger;

  constructor(config: RateLimitConfig, logger: Logger) {
    this.config = config;
    this.logger = logger;
    this.tokens = config.maxTokens;
    this.lastRefill = Date.now();
  }

  /**
   * Consume a token, waiting if necessary
   */
  async consume(): Promise<void> {
    this.refill();

    if (this.tokens >= 1) {
      this.tokens -= 1;
      return;
    }

    // Calculate wait time
    const tokensNeeded = 1 - this.tokens;
    const waitTime = (tokensNeeded / this.config.refillRate) * 1000;

    this.logger.debug('Rate limit hit, waiting', {
      waitTime: Math.round(waitTime),
      endpoint: this.config.name
    });

    await new Promise(resolve => setTimeout(resolve, waitTime));

    // Try again after waiting
    return this.consume();
  }

  /**
   * Check if tokens are available
   */
  hasTokens(): boolean {
    this.refill();
    return this.tokens >= 1;
  }

  /**
   * Get current status
   */
  getStatus(): RateLimitStatus {
    this.refill();

```

```
    const nextTokenTime = this.tokens < this.config.maxTokens
      ? Date.now() + ((1 - (this.tokens % 1)) / this.config.refillRate) * 1000
      : null;

    return {
      configured: true,
      available: Math.floor(this.tokens),
      resetTime: nextTokenTime,
      maxTokens: this.config.maxTokens,
      refillRate: this.config.refillRate,
    };
  }

  /**
   * Reset bucket to full capacity
   */
  reset(): void {
    this.tokens = this.config.maxTokens;
    this.lastRefill = Date.now();
  }

  /**
   * Refill tokens based on elapsed time
   */
  private refill(): void {
    const now = Date.now();
    const elapsed = (now - this.lastRefill) / 1000; // Convert to seconds
    const tokensToAdd = elapsed * this.config.refillRate;

    this.tokens = Math.min(this.config.maxTokens, this.tokens + tokensToAdd);
    this.lastRefill = now;
  }
}

/**
 * Cache Manager - Multi-layer Caching System
 */
export class CacheManager {
  private memoryCache = new Map<string, CacheEntry>();
  private readonly logger: Logger;
  private cleanupInterval: NodeJS.Timeout;

  constructor(logger: Logger) {
    this.logger = logger;

    // Cleanup expired entries every 5 minutes
    this.cleanupInterval = setInterval(() => {
      this.cleanup();
    }, 300000);
  }

  /**
   * Get value from cache
   * @param key - Cache key
   * @returns Promise<T | null>
   */
  async get<T>(key: string): Promise<T | null> {
    const entry = this.memoryCache.get(key);

    if (!entry) {
      return null;
    }

```

```
    if (this.isExpired(entry)) {
      this.memoryCache.delete(key);
      return null;
    }

    // Update access time for LRU
    entry.lastAccessed = Date.now();

    this.logger.debug('Cache hit', { key });
    return entry.value as T;
  }

  /**
   * Set value in cache
   * @param key - Cache key
   * @param value - Value to cache
   * @param ttl - Time to live in milliseconds
   */
  async set<T>(key: string, value: T, ttl: number): Promise<void> {
    const entry: CacheEntry = {
      value,
      expiresAt: Date.now() + ttl,
      createdAt: Date.now(),
      lastAccessed: Date.now(),
      size: this.estimateSize(value),
    };

    this.memoryCache.set(key, entry);

    this.logger.debug('Cache set', { key, ttl, size: entry.size });

    // Check if we need to evict entries
    this.evictIfNeeded();
  }

  /**
   * Delete value from cache
   * @param key - Cache key
   */
  async delete(key: string): Promise<void> {
    const deleted = this.memoryCache.delete(key);
    if (deleted) {
      this.logger.debug('Cache delete', { key });
    }
  }

  /**
   * Clear all cache entries
   */
  async clear(): Promise<void> {
    const size = this.memoryCache.size;
    this.memoryCache.clear();
    this.logger.info('Cache cleared', { entriesRemoved: size });
  }

  /**
   * Get cache statistics
   */
  getStats(): CacheStats {
    let totalSize = 0;
    let expiredCount = 0;
    const now = Date.now();

```

```
    for (const entry of this.memoryCache.values()) {
      totalSize += entry.size;
      if (this.isExpired(entry)) {
        expiredCount++;
      }
    }

    return {
      totalEntries: this.memoryCache.size,
      totalSize,
      expiredEntries: expiredCount,
      hitRate: 0, // Would need to track hits/misses for this
    };
  }

  /**
   * Cleanup expired entries
   */
  private cleanup(): void {
    const before = this.memoryCache.size;
    const now = Date.now();

    for (const [key, entry] of this.memoryCache.entries()) {
      if (this.isExpired(entry)) {
        this.memoryCache.delete(key);
      }
    }

    const removed = before - this.memoryCache.size;
    if (removed > 0) {
      this.logger.debug('Cache cleanup completed', { entriesRemoved: removed });
    }
  }

  /**
   * Evict entries if cache is too large
   */
  private evictIfNeeded(): void {
    const maxEntries = 1000; // Maximum number of entries
    const maxSize = 50 * 1024 * 1024; // 50MB

    if (this.memoryCache.size <= maxEntries) {
      return;
    }

    // Convert to array and sort by last accessed time (LRU)
    const entries = Array.from(this.memoryCache.entries())
      .sort(([, a], [, b]) => a.lastAccessed - b.lastAccessed);

    // Remove oldest entries until we're under the limit
    const toRemove = this.memoryCache.size - maxEntries;
    for (let i = 0; i < toRemove; i++) {
      const [key] = entries[i];
      this.memoryCache.delete(key);
    }

    this.logger.debug('Cache eviction completed', { entriesRemoved: toRemove });
  }

  /**
   * Check if cache entry is expired
   */
  private isExpired(entry: CacheEntry): boolean {
```

```
    return Date.now() > entry.expiresAt;
  }

  /**
   * Estimate size of cached value
   */
  private estimateSize(value: any): number {
    try {
      return JSON.stringify(value).length * 2; // Rough estimate (UTF-16)
    } catch {
      return 1000; // Default size if serialization fails
    }
  }

  /**
   * Cleanup on destruction
   */
  destroy(): void {
    if (this.cleanupInterval) {
      clearInterval(this.cleanupInterval);
    }
    this.memoryCache.clear();
  }
}
```

Listing 5: Sophisticated Rate Limiting System

6 Conclusion

This comprehensive API integration guide provides detailed implementation patterns for all external services used in the Jupiter Swap DApp. The integration follows best practices for reliability, performance, and maintainability.

6.1 Integration Summary

Key Integration Features:

- **Jupiter API v6:** Complete integration with advanced features
- **Multi-RPC Strategy:** Helius primary, Alchemy backup
- **Market Data:** CoinGecko integration for optimization
- **Rate Limiting:** Sophisticated token bucket implementation
- **Caching:** Multi-layer caching with LRU eviction
- **Error Handling:** Comprehensive retry and fallback logic
- **Performance:** Optimized for speed and reliability
- **Monitoring:** Detailed logging and metrics