perplexity

Fi MCP Development Server - Ultimate Reference Manual and Agent Development Guide

Table of Contents

1. Introduction and Overview

What is Fi MCP Development Server

The Fi MCP Development Server (`fi-mcp-dev`) is a **hackathon-ready mock server** that simulates the production Fi MCP (Model Context Protocol) server[^1]. It provides a lightweight, safe environment for developers to experiment with India's first personal finance MCP without accessing real user data or production systems[^2].

Core Architecture Principles

The Fi MCP server operates on several key architectural foundations:

Model Context Protocol Implementation: Fi MCP is built on the standardized MCP specification[3][4], enabling seamless integration with AI assistants and agents. Unlike traditional APIs, MCP provides a **structured, query-ready live stream** of financial data specifically designed for AI consumption[^5].

Event-Driven Architecture: The server processes requests through JSON-RPC 2.0 messages[^3], providing real-time streaming capabilities essential for financial data applications where timeliness is critical.

Security-First Design: All financial data is served from static JSON files representing various user scenarios, ensuring no real financial information is exposed during development[^1].

Fi MCP vs Traditional Financial APIs

Traditional financial APIs require custom integrations for each data source and often provide fragmented access to different financial accounts. Fi MCP provides a **unified, standardized interface** that aggregates data from multiple sources:

- **Bank account balances and transactions**
- **Mutual fund holdings and performance analytics**
- **EPF (Employee Provident Fund) details**
- **Credit reports and scores**
- **Indian and US stock portfolios**
- **Insurance and loan information**

This comprehensive approach enables AI agents to provide **holistic financial insights** rather than fragmented, source-specific analysis.

2. Fi MCP Server Architecture and Implementation

Directory Structure and Components

The Fi MCP development server follows a clean, modular architecture[^1]:

```
fi-mcp-dev/
├── main.go                    # Server entrypoint and routing
├── middlewares/
│   └── auth.go               # Authentication and session management
├── test_data_dir/           # User scenarios and financial data
│   ├── 1111111111/         # Phone number directories
│   │   ├── fetch_net_worth.json
│   │   ├── fetch_credit_report.json
│   │   ├── fetch_epf_details.json
│   │   ├── fetch_mf_transactions.json
│   │   ├── fetch_bank_transactions.json
│   │   └── fetch_stock_transactions.json
│   └── [25 other user scenarios]
└── static/                  # HTML templates for authentication
    ├── login.html
    └── login-successful.html
```

Authentication Flow Architecture

The Fi MCP server implements a **three-step authentication process** designed to simulate real-world security while maintaining development simplicity:

Step 1: Session Initialization

```
// When any tool is called without authentication
if !isAuthenticated(sessionId) {
```

```
    return authenticationRequiredResponse(sessionId)
}
```

Step 2: Web-Based Login
The server redirects to `/mockWebPage?sessionId=...` where users enter their phone number. The phone number must correspond to a directory in `test_data_dir/`.

Step 3: Session Persistence
Once authenticated, the session is stored in memory for the server's lifetime, enabling multiple API calls without re-authentication.

Data Serving Architecture

The server uses a **file-based data serving pattern** where each user scenario (identified by phone number) has a dedicated directory containing six JSON files representing different financial data endpoints[^1]:

1. `fetch_net_worth.json` - Comprehensive net worth calculation
2. `fetch_credit_report.json` - Credit score and loan details
3. `fetch_epf_details.json` - Employee Provident Fund information
4. `fetch_mf_transactions.json` - Mutual fund transaction history
5. `fetch_bank_transactions.json` - Bank account transaction data
6. `fetch_stock_transactions.json` - Stock portfolio transactions

3. Data Structures and API Endpoints

Core API Endpoints and Their Purpose

The Fi MCP server exposes five primary tools through the MCP protocol, each serving specific financial analysis needs:

1. `fetch_net_worth` - Comprehensive Wealth Analysis

Purpose: Calculate and analyze comprehensive net worth using actual data from connected financial accounts.

Use Cases:

- Portfolio analysis and asset allocation insights
- Net worth tracking and trend visualization
- Financial health assessments
- Investment performance analysis
- Debt-to-asset ratio calculations

Data Structure Deep Dive:

```
{
  "netWorthResponse": {
    "assetValues": [
      {
        "netWorthAttribute": "ASSET_TYPE_MUTUAL_FUND",
        "value": {
          "currencyCode": "INR",
          "units": "84613"
        }
      }
    ],
    "liabilityValues": [
      {
        "netWorthAttribute": "LIABILITY_TYPE_VEHICLE_LOAN",
        "value": {
          "currencyCode": "INR",
          "units": "5000"
        }
      }
    ],
    "totalNetWorthValue": {
      "currencyCode": "INR",
      "units": "868721"
    }
  }
}
```


Asset Type Classifications:

- `ASSET_TYPE_EPF`: Employee Provident Fund balance
- `ASSET_TYPE_INDIAN_SECURITIES`: Indian stock holdings
- `ASSET_TYPE_SAVINGS_ACCOUNTS`: Bank account balances
- `ASSET_TYPE_MUTUAL_FUND`: Mutual fund portfolio value
- `ASSET_TYPE_US_SECURITIES`: US stock investments

Liability Type Classifications:

- `LIABILITY_TYPE_HOME_LOAN`: Home loan outstanding
- `LIABILITY_TYPE_VEHICLE_LOAN`: Vehicle loan balance
- `LIABILITY_TYPE_OTHER_LOAN`: Personal and other loans

2. `fetch_credit_report` - Credit Analysis and Risk Assessment

Purpose: Retrieve comprehensive credit report information for loan prioritization and credit risk analysis.

Financial Intelligence Applications:

- **Credit Score Monitoring**: Track bureau scores over time
- **Loan Prioritization**: Identify highest interest rate debts first

- **Credit Utilization Analysis**: Calculate credit card usage ratios
- **Payment History Evaluation**: Assess payment consistency patterns

Key Data Points:

```
{
  "creditReports": [{
    "creditReportData": {
      "score": {
        "bureauScore": "746",
        "bureauScoreConfidenceLevel": "H"
      },
      "creditAccount": {
        "creditAccountDetails": [{
          "subscriberName": "HDFC Bank",
          "accountType": "10",  // Credit Card
          "rateOfInterest": "11.5",
          "currentBalance": "5000",
          "amountPastDue": "1000",
          "paymentRating": "0"  // Current (no delays)
        }]
      }
    }
  }]
}
```

Credit Account Type Mapping:

- `"01"`: Home Loan
- `"03"`: Personal Loan
- `"04"`: Vehicle Loan
- `"10"`: Credit Card accounts
- `"53"`: Business Loan

Payment Rating Interpretation:

- `"0"`: Current (no delays)
- `"1"`: 30 days past due
- `"2"`: 60 days past due
- `"3"`: 90 days past due
- `"4"`: 120 days past due
- `"5"`: 150+ days past due

3. `fetch_epf_details` - Retirement Planning Data

Purpose: Access Employee Provident Fund account information for retirement planning and contribution tracking.

Strategic Applications:

- **Retirement Planning Calculations**: Project future EPF corpus
- **EPF Contribution Tracking**: Monitor employee vs employer contributions
- **Interest Earned Analysis**: Calculate EPF returns over time
- **Employer Contribution Verification**: Ensure proper employer compliance

Data Structure Analysis:


```
{
  "uanAccounts": [{
    "rawDetails": {
      "est_details": [{
        "est_name": "KARZA TECHNOLOGIES PRIVATE LIMITED",
        "doj_epf": "24-03-2021",
        "doe_epf": "02-01-2022",
        "pf_balance": {
          "net_balance": "200000",
          "employee_share": {
            "credit": "100000",
            "balance": "100000"
          },
          "employer_share": {
            "credit": "100000",
            "balance": "100000"
          }
        }
      }],
      "overall_pf_balance": {
        "current_pf_balance": "211111",
        "employee_share_total": {
          "balance": "11111"
        }
      }
    }
  }]
}
```


4. `fetch_mf_transactions` - Investment Performance Analysis

Purpose: Retrieve mutual fund transaction history for XIRR calculations and portfolio performance analysis.

Advanced Analytics Applications:

- **XIRR (Extended Internal Rate of Return) Calculations**: Measure annualized returns

- **Investment Pattern Analysis**: Identify SIP consistency and timing
- **Fund Performance Comparison**: Compare returns across different schemes
- **Portfolio Rebalancing Insights**: Suggest optimal allocation changes

Transaction Data Structure:

```
{
  "transactions": [{
    "isinNumber": "INF760K01FC4",
    "folioId": "55557777",
    "externalOrderType": "BUY",
    "transactionDate": "2022-12-31T18:30:00Z",
    "purchasePrice": {
      "currencyCode": "INR",
      "units": "66",
      "nanos": 554600000
    },
    "transactionAmount": {
      "currencyCode": "INR",
      "units": "6655",
      "nanos": 460000000
    },
    "transactionUnits": 100,
    "transactionMode": "N",
    "schemeName": "Canara Robeco Gilt Fund - Regular Plan"
  }]
}
```

Transaction Mode Classifications:

- `"N"`: Normal purchase/redemption
- `"S"`: SIP (Systematic Investment Plan)
- `"D"`: Dividend reinvestment
- `"B"`: Bonus units allocation

Currency Format and Precision Handling

All monetary values in Fi MCP use a **precision-aware format** that handles both large amounts and fractional values accurately:

```
{
  "currencyCode": "INR",
  "units": "string_value",
  "nanos": "numeric_fractional_value"
}
```

Precision Calculation:

```
// To get actual monetary value
const actualValue = parseFloat(units) + (nanos / 1000000000);
```

This format ensures financial calculations maintain precision for both small transactions (₹1.50) and large amounts (₹10,00,000), critical for accurate financial analysis.

4. Financial Archetype Classification System

Understanding User Financial Personas

The Fi MCP development server includes **26 distinct user scenarios** representing the full spectrum of Indian financial behaviors and circumstances. These scenarios enable developers to build robust agents that can handle diverse financial situations.

Primary Financial Archetypes

Beginner Investors (Phone Numbers: 1111111111, 2020202020)

Characteristics:

- **No Assets Connected** (`1111111111`): Only basic savings account balance
- **Starter Saver** (`2020202020`): Recently started investing with low ticket sizes (₹500-₹1000 SIPs)

Agent Development Implications:

- Focus on **financial education** and **basic planning**
- Recommend **simple, low-risk investment options**
- Emphasize **building emergency funds** before investments
- Provide **step-by-step guidance** for account linking

Code Pattern for Detection:

```
def detect_beginner_investor(financial_data):
    net_worth = financial_data.get('fetch_net_worth', {})
    mf_transactions = financial_data.get('fetch_mf_transactions', {})

    # Check if only savings account exists
    assets = net_worth.get('netWorthResponse', {}).get('assetValues', [])
    savings_only = len([a for a in assets if a['netWorthAttribute'] == 'ASSET_TYPE_SAVING

    # Check for recent, small investments
    transactions = mf_transactions.get('transactions', [])
    recent_small_investments = any(
        float(t['transactionAmount']['units']) < 5000
        for t in transactions[-5:]  # Last 5 transactions
    )
```

```
    return savings_only or recent_small_investments
```

Conservative Investors (Phone Numbers: 9999999999, 1010101010, 2424242424)

Characteristics:

- **Fixed Income Fanatic** (`9999999999`): 80% allocation to debt mutual funds and FDs
- **Precious Metal Believer** (`1010101010`): ~50% allocation to gold MFs/ETFs
- **Mattress Money Mindset** (`2424242424`): 95% in FDs/savings, minimal market exposure

Investment Psychology:

- **Risk-averse** with preference for guaranteed returns
- **Capital preservation** over wealth creation
- **Inflation protection** concerns driving gold allocation
- **Liquidity preference** for immediate access to funds

Agent Strategies:

- **Gradual transition** to slightly higher-risk instruments
- **Education on inflation impact** on fixed deposits
- **Hybrid fund recommendations** for balanced exposure
- **Tax-efficient fixed income** options like PPF, ELSS

Aggressive Wealth Builders (Phone Numbers: 2222222222, 1313131313, 1616161616)

Characteristics:

- **Comprehensive Asset Coverage**: All major asset classes connected
- **High Equity Exposure**: 80-90% equity allocation for long-term growth
- **Systematic Approach**: Regular SIPs and disciplined investing
- **International Diversification**: 10-20% allocation to US/international funds

Financial Metrics:

- **High Savings Rate**: 30-50% of income invested
- **Long Investment Horizon**: 10+ year investment timelines
- **Multiple Account Integration**: EPF, MFs, stocks, US securities

Agent Development Focus:

- **Advanced portfolio optimization** algorithms
- **Tax-loss harvesting** strategies

- **Asset rebalancing** recommendations
- **Goal-based investment planning** (retirement, children's education)

```
def analyze_aggressive_wealth_builder(financial_data):
    net_worth = financial_data.get('fetch_net_worth', {})
    mf_analytics = net_worth.get('mfSchemeAnalytics', {}).get('schemeAnalytics', [])

    # Calculate equity exposure
    total_mf_value = 0
    equity_value = 0

    for scheme in mf_analytics:
        current_value = float(scheme['enrichedAnalytics']['analytics']['schemeDetails']['
        total_mf_value += current_value

        if scheme['schemeDetail']['assetClass'] == 'EQUITY':
            equity_value += current_value

    equity_percentage = (equity_value / total_mf_value) * 100 if total_mf_value > 0 else

    return {
        'equity_exposure': equity_percentage,
        'is_aggressive': equity_percentage > 70,
        'diversification_score': len(set(s['schemeDetail']['amc'] for s in mf_analytics))
    }
```

Debt-Stressed Individuals (Phone Number: 7777777777)

Critical Characteristics:

- **High Liability Load**: Multiple active loans and credit card debt
- **Poor Investment Performance**: XIRR < 5% on mutual fund investments
- **Credit Score Issues**: Score < 650 with high utilization
- **Cash Flow Problems**: Negative or minimal net worth

Agent Intervention Strategies:

- **Debt Consolidation Analysis**: Identify highest-interest debts for priority repayment
- **Cash Flow Optimization**: Track income vs. expenses to find savings opportunities
- **Credit Score Improvement**: Provide actionable steps to improve credit health
- **Emergency Fund Building**: Even ₹500/month emergency fund to prevent additional debt

```
def assess_debt_stress_level(financial_data):
    net_worth = financial_data.get('fetch_net_worth', {})
    credit_report = financial_data.get('fetch_credit_report', {})

    # Calculate debt-to-asset ratio
    assets = sum(float(a['value']['units']) for a in net_worth.get('netWorthResponse', {}
    liabilities = sum(float(l['value']['units']) for l in net_worth.get('netWorthResponse
```

```
    debt_to_asset_ratio = liabilities / assets if assets > 0 else float('inf')

    # Check credit score
    credit_score = 0
    if credit_report.get('creditReports'):
        credit_score = int(credit_report['creditReports'][^0]['creditReportData']['score'

    return {
        'debt_stress_level': 'HIGH' if debt_to_asset_ratio > 0.5 or credit_score < 650 el
        'debt_to_asset_ratio': debt_to_asset_ratio,
        'credit_score': credit_score,
        'immediate_action_required': debt_to_asset_ratio > 0.8
    }
```

Systematic Investors (Phone Numbers: 8888888888, 1919191919)

Characteristics:

- **SIP Samurai** (`8888888888`): 3-5 active SIPs with consistent 8-12% XIRR
- **Section 80C Strategist** (`1919191919`): Tax-optimized investing through ELSS, EPF, NPS

Investment Discipline Metrics:

- **SIP Consistency**: Zero missed SIP payments over 2+ years
- **Goal-Based Allocation**: Clear allocation between tax-saving and wealth creation
- **Performance Tracking**: Regular monitoring of fund performance and rebalancing

Agent Enhancement Opportunities:

- **SIP Optimization**: Suggest optimal SIP dates based on market volatility
- **Tax Planning**: Year-end tax-saving investment reminders
- **Performance Attribution**: Break down returns by fund category and time period

Special Circumstance Archetypes

NRI Investor (Phone Number: 2323232323)

Unique Challenges:

- **Large EPF Corpus** without current contributions
- **Bulk MF Transactions** instead of systematic investing
- **Currency Risk Management** between USD earnings and INR investments
- **Regulatory Compliance** for NRI investment rules

Agent Specialization Requirements:

- **NRI-Specific Investment Options**: FCNR, NRE account optimization
- **Tax Implications**: Double taxation avoidance strategies

- **Repatriation Rules**: Understanding of fund transfer regulations

Freelancer/Gig Worker (Phone Number: 2121212121)

Income Characteristics:

- **Irregular Cash Flow**: Freelance + salary income with UPI app credits
- **High Liquidity Needs**: Irregular income requires larger emergency funds
- **Business Expense Management**: Need to track business vs. personal expenses

Agent Adaptations:

- **Variable SIP Strategies**: Flexible SIP amounts based on income months
- **Tax Planning for Freelancers**: ITR-3 optimization and business deductions
- **Separate Business vs. Personal Analysis**: Track profitability metrics

5. Advanced ADK Integration Patterns

Architecting Production-Ready Financial Agents

Building sophisticated financial agents requires understanding both ADK's capabilities and the nuances of financial data analysis. The integration patterns below demonstrate how to create agents that provide genuine financial value.

Multi-Tool Orchestration Pattern

Financial analysis requires **coordinated data gathering** from multiple endpoints. Here's an advanced pattern for comprehensive financial health assessment:

```
from google.adk.agents import LlmAgent, SequentialAgent
from google.adk.tools import ToolContext
import asyncio
from typing import Dict, List, Any

class FinancialDataOrchestrator:
    """Orchestrates multiple Fi MCP calls for comprehensive analysis"""

    def __init__(self, mcp_client):
        self.mcp_client = mcp_client

    async def comprehensive_financial_analysis(self, phone_number: str, tool_context: Too
        """Fetch all financial data sources and perform cross-analysis"""

        # Parallel data fetching for performance
        data_tasks = {
            'net_worth': self.mcp_client.call_tool('fetch_net_worth'),
            'credit_report': self.mcp_client.call_tool('fetch_credit_report'),
            'epf_details': self.mcp_client.call_tool('fetch_epf_details'),
            'mf_transactions': self.mcp_client.call_tool('fetch_mf_transactions'),
            'bank_transactions': self.mcp_client.call_tool('fetch_bank_transactions'),
            'stock_transactions': self.mcp_client.call_tool('fetch_stock_transactions')
```

```
        }

        # Execute all calls concurrently
        results = {}
        for key, task in data_tasks.items():
            try:
                results[key] = await task
            except Exception as e:
                results[key] = {'error': str(e), 'available': False}
                print(f"Warning: Could not fetch {key}: {e}")

        # Store comprehensive data in session state
        tool_context.state[f"financial_data_{phone_number}"] = results
        tool_context.state["last_analysis_timestamp"] = time.time()

        # Perform cross-data analysis
        analysis = self._cross_analyze_financial_data(results)
        tool_context.state[f"financial_analysis_{phone_number}"] = analysis

        return {
            'data_sources': results,
            'analysis': analysis,
            'data_completeness': self._assess_data_completeness(results)
        }

    def _cross_analyze_financial_data(self, financial_data: Dict[str, Any]) -> Dict[str,
        """Perform sophisticated cross-data analysis"""
        analysis = {
            'financial_health_score': 0,
            'risk_factors': [],
            'opportunities': [],
            'alerts': []
        }

        # Net worth vs. age analysis
        net_worth_data = financial_data.get('net_worth', {})
        credit_data = financial_data.get('credit_report', {})

        if net_worth_data.get('netWorthResponse'):
            total_net_worth = float(net_worth_data['netWorthResponse']['totalNetWorthValu

            # Extract age from credit report
            age = self._calculate_age_from_credit_report(credit_data)
            if age:
                expected_net_worth = age * 50000  # Rule of thumb: Age × ₹50k
                net_worth_ratio = total_net_worth / expected_net_worth

                if net_worth_ratio < 0.5:
                    analysis['alerts'].append('Net worth significantly below age-appropri
                elif net_worth_ratio > 2.0:
                    analysis['opportunities'].append('Excellent wealth accumulation - cor

        # Debt-to-income analysis
        self._analyze_debt_burden(financial_data, analysis)

        # Investment diversification analysis
```

```
        self._analyze_portfolio_diversification(financial_data, analysis)

        # Credit health assessment
        self._analyze_credit_health(financial_data, analysis)

        return analysis

    def _analyze_debt_burden(self, financial_data: Dict[str, Any], analysis: Dict[str, An
        """Analyze debt burden and repayment capacity"""
        net_worth_data = financial_data.get('net_worth', {})
        bank_transactions = financial_data.get('bank_transactions', {})

        if not net_worth_data.get('netWorthResponse'):
            return

        liabilities = net_worth_data['netWorthResponse'].get('liabilityValues', [])
        total_debt = sum(float(l['value']['units']) for l in liabilities)

        if total_debt > 0:
            # Estimate monthly income from bank transactions
            monthly_income = self._estimate_monthly_income(bank_transactions)

            if monthly_income > 0:
                # Assume 40% of debt as annual EMI burden (rough estimate)
                estimated_annual_emi = total_debt * 0.4
                debt_to_income_ratio = estimated_annual_emi / (monthly_income * 12)

                if debt_to_income_ratio > 0.5:
                    analysis['risk_factors'].append(f'High debt burden: {debt_to_income_r
                    analysis['alerts'].append('Consider debt consolidation or aggressive

    def _analyze_portfolio_diversification(self, financial_data: Dict[str, Any], analysis
        """Analyze investment portfolio diversification"""
        net_worth_data = financial_data.get('net_worth', {})

        if not net_worth_data.get('mfSchemeAnalytics'):
            return

        schemes = net_worth_data['mfSchemeAnalytics'].get('schemeAnalytics', [])

        # Calculate asset class allocation
        asset_allocation = {'EQUITY': 0, 'DEBT': 0, 'HYBRID': 0, 'CASH': 0}
        total_mf_value = 0

        for scheme in schemes:
            current_value = float(scheme['enrichedAnalytics']['analytics']['schemeDetails
            asset_class = scheme['schemeDetail']['assetClass']

            asset_allocation[asset_class] += current_value
            total_mf_value += current_value

        if total_mf_value > 0:
            equity_percentage = (asset_allocation['EQUITY'] / total_mf_value) * 100

            if equity_percentage > 90:
                analysis['risk_factors'].append(f'Very high equity exposure: {equity_perc
```

```
            elif equity_percentage < 20:
                analysis['opportunities'].append('Consider increasing equity allocation f

            # Check fund house diversification
            fund_houses = set(s['schemeDetail']['amc'] for s in schemes)
            if len(fund_houses) < 3:
                analysis['opportunities'].append('Consider diversifying across more fund

def create_financial_orchestration_tool(mcp_client):
    """Factory function to create orchestration tool"""
    orchestrator = FinancialDataOrchestrator(mcp_client)

    async def orchestrate_financial_analysis(phone_number: str, tool_context: ToolContext
        """Main orchestration function to be used as ADK tool"""
        return await orchestrator.comprehensive_financial_analysis(phone_number, tool_cor

    return orchestrate_financial_analysis
```

State-Aware Financial Planning Agent

Financial planning requires **contextual memory** across multiple conversations. Here's how to build an agent that maintains financial context:

```
class FinancialPlanningAgent:
    """Advanced financial planning agent with persistent state management"""

    def __init__(self, mcp_client):
        self.mcp_client = mcp_client
        self.setup_agent()

    def setup_agent(self):
        """Configure ADK agent with sophisticated financial planning capabilities"""

        # Create financial analysis tools
        self.financial_tools = [
            self.create_goal_based_planning_tool(),
            self.create_tax_optimization_tool(),
            self.create_portfolio_rebalancing_tool(),
            self.create_debt_optimization_tool(),
            create_financial_orchestration_tool(self.mcp_client)
        ]

        self.agent = LlmAgent(
            model="gemini-2.0-flash",
            name="comprehensive_financial_planner",
            instruction="""
            You are an expert financial advisor with comprehensive access to a user's fir

            CORE CAPABILITIES:
            1. **Holistic Financial Analysis**: Access complete financial picture includi
            2. **Goal-Based Planning**: Create specific, measurable financial plans for r
            3. **Tax Optimization**: Identify tax-saving opportunities across 80C, 80D, a
            4. **Risk Assessment**: Evaluate financial health, debt burden, and investmen
            5. **Performance Tracking**: Monitor investment returns, SIP performance, and
```

```
            ANALYSIS APPROACH:
            - Always start with comprehensive data gathering using orchestrate_financial_
            - Cross-reference data from multiple sources for accuracy
            - Provide specific, actionable recommendations with rupee amounts and timelin
            - Consider user's age, risk tolerance, and life stage in recommendations
            - Identify both immediate actions and long-term strategies

            STATE MANAGEMENT:
            - Remember previous analyses and recommendations across conversations
            - Track progress toward financial goals over time
            - Maintain context about user's preferences and risk tolerance

            COMMUNICATION STYLE:
            - Use clear, jargon-free explanations
            - Provide specific numbers and calculations
            - Explain the 'why' behind recommendations
            - Offer alternative scenarios and what-if analysis

            When user provides their phone number, immediately analyze their complete fin
            and provide a comprehensive financial health assessment.
            """,
            tools=self.financial_tools,
            output_key="financial_plan_response"
        )

        self.session_service = InMemorySessionService()
        self.runner = Runner(
            agent=self.agent,
            app_name="comprehensive_financial_planner",
            session_service=self.session_service
        )

    def create_goal_based_planning_tool(self):
        """Create tool for goal-based financial planning"""

        def goal_based_planning(goal_type: str, target_amount: float, time_horizon: int,
            """
            Create goal-based investment plan

            Args:
                goal_type: Type of goal (retirement, home_purchase, child_education, emer
                target_amount: Target amount needed (in INR)
                time_horizon: Years to achieve the goal
                tool_context: ADK tool context for state access
            """

            # Get current financial position
            phone_number = tool_context.state.get("user:phone_number")
            financial_data = tool_context.state.get(f"financial_data_{phone_number}", {})

            current_investments = self._calculate_goal_relevant_investments(financial_dat

            # Calculate required monthly SIP
            expected_return = self._get_expected_return_for_goal(goal_type)
            required_monthly_sip = self._calculate_sip_amount(
```

```
                target_amount, time_horizon, expected_return, current_investments
            )

            # Asset allocation recommendation
            asset_allocation = self._recommend_asset_allocation(goal_type, time_horizon)

            # Specific fund recommendations
            fund_recommendations = self._recommend_funds_for_goal(goal_type, asset_alloca

            plan = {
                'goal_type': goal_type,
                'target_amount': target_amount,
                'time_horizon': time_horizon,
                'current_investments': current_investments,
                'required_monthly_sip': required_monthly_sip,
                'asset_allocation': asset_allocation,
                'fund_recommendations': fund_recommendations,
                'tax_implications': self._analyze_tax_implications(goal_type, required_mo
            }

            # Store plan in state for tracking
            goal_plans = tool_context.state.get("user:goal_plans", {})
            goal_plans[goal_type] = plan
            tool_context.state["user:goal_plans"] = goal_plans

            return plan

        return goal_based_planning

    def create_tax_optimization_tool(self):
        """Create comprehensive tax optimization tool"""

        def tax_optimization_analysis(assessment_year: str, tool_context: ToolContext) ->
            """
            Perform comprehensive tax optimization analysis

            Args:
                assessment_year: Assessment year (e.g., "2024-25")
                tool_context: ADK tool context
            """

            phone_number = tool_context.state.get("user:phone_number")
            financial_data = tool_context.state.get(f"financial_data_{phone_number}", {})

            # Current 80C investments
            current_80c = self._calculate_current_80c_investments(financial_data)

            # EPF analysis
            epf_data = financial_data.get('epf_details', {})
            epf_contribution = self._calculate_annual_epf_contribution(epf_data)

            # Calculate tax-saving opportunities
            remaining_80c_limit = max(0, 150000 - current_80c - epf_contribution)

            recommendations = {
                'current_80c_utilization': current_80c,
```

```
                'epf_contribution': epf_contribution,
                'remaining_80c_limit': remaining_80c_limit,
                'recommendations': []
            }

            if remaining_80c_limit > 0:
                recommendations['recommendations'].append({
                    'type': 'ELSS',
                    'amount': min(remaining_80c_limit, 50000),
                    'tax_saving': min(remaining_80c_limit, 50000) * 0.31,  # Assuming 31%
                    'rationale': 'ELSS provides tax saving with potential for equity retu
                })

            # 80D health insurance analysis
            recommendations['health_insurance_80d'] = self._analyze_health_insurance_dedu

            # HRA optimization if applicable
            recommendations['hra_optimization'] = self._analyze_hra_optimization(financia

            return recommendations

        return tax_optimization_analysis
```

Real-Time Portfolio Monitoring System

Create agents that provide **continuous portfolio monitoring** with automated alerts and rebalancing suggestions:

```
class PortfolioMonitoringAgent:
    """Real-time portfolio monitoring with automated insights"""

    def __init__(self, mcp_client):
        self.mcp_client = mcp_client
        self.setup_monitoring_agent()

    def setup_monitoring_agent(self):
        """Setup agent with portfolio monitoring capabilities"""

        monitoring_tools = [
            self.create_performance_tracker(),
            self.create_rebalancing_analyzer(),
            self.create_risk_monitor(),
            self.create_sip_optimizer()
        ]

        self.agent = LlmAgent(
            model="gemini-2.0-flash",
            name="portfolio_monitor",
            instruction="""
            You are a portfolio monitoring specialist focused on investment performance t

            KEY RESPONSIBILITIES:
            1. **Performance Analysis**: Calculate XIRR, absolute returns, and compare ag
            2. **Risk Monitoring**: Track portfolio volatility, concentration risk, and a
```

```
            3. **Rebalancing**: Identify when portfolio allocation deviates from target a
            4. **SIP Optimization**: Analyze SIP performance and suggest timing/amount op
            5. **Alert Generation**: Proactively identify issues requiring immediate atte

            ANALYSIS METHODOLOGY:
            - Use actual transaction data to calculate precise performance metrics
            - Compare performance against relevant benchmarks (Nifty 50, debt indices)
            - Consider tax implications in all recommendations
            - Factor in transaction costs and exit loads

            COMMUNICATION:
            - Provide clear performance attribution (what contributed to gains/losses)
            - Suggest specific actions with expected impact
            - Explain market context behind performance
            - Prioritize recommendations by potential impact
            """,
            tools=monitoring_tools,
            output_key="portfolio_analysis"
        )

    def create_performance_tracker(self):
        """Create comprehensive performance tracking tool"""

        def track_portfolio_performance(analysis_period: str, tool_context: ToolContext)
            """
            Track portfolio performance across multiple dimensions

            Args:
                analysis_period: Period for analysis (1Y, 3Y, 5Y, inception)
                tool_context: ADK tool context
            """

            phone_number = tool_context.state.get("user:phone_number")
            financial_data = tool_context.state.get(f"financial_data_{phone_number}", {})

            mf_data = financial_data.get('net_worth', {}).get('mfSchemeAnalytics', {})
            transaction_data = financial_data.get('mf_transactions', {})

            performance_analysis = {
                'overall_portfolio': {},
                'scheme_wise_performance': [],
                'asset_class_performance': {},
                'benchmark_comparison': {}
            }

            # Calculate overall portfolio XIRR
            overall_xirr = self._calculate_portfolio_xirr(transaction_data, mf_data)
            performance_analysis['overall_portfolio']['xirr'] = overall_xirr

            # Scheme-wise performance breakdown
            for scheme in mf_data.get('schemeAnalytics', []):
                scheme_performance = self._analyze_scheme_performance(scheme, transaction
                performance_analysis['scheme_wise_performance'].append(scheme_performance

            # Asset class performance
            asset_classes = ['EQUITY', 'DEBT', 'HYBRID', 'CASH']
```

```
            for asset_class in asset_classes:
                class_performance = self._calculate_asset_class_performance(mf_data, asse
                performance_analysis['asset_class_performance'][asset_class] = class_perf

            # Benchmark comparison
            performance_analysis['benchmark_comparison'] = self._compare_against_benchmar
                performance_analysis['asset_class_performance']
            )

            # Generate performance insights
            insights = self._generate_performance_insights(performance_analysis)
            performance_analysis['insights'] = insights

            return performance_analysis

        return track_portfolio_performance

    def _calculate_portfolio_xirr(self, transaction_data: dict, mf_data: dict) -> float:
        """Calculate portfolio-level XIRR using cash flow method"""

        transactions = transaction_data.get('transactions', [])
        current_values = {}

        # Get current values for each scheme
        for scheme in mf_data.get('schemeAnalytics', []):
            isin = scheme['schemeDetail']['isinNumber']
            current_value = float(scheme['enrichedAnalytics']['analytics']['schemeDetails
            current_values[isin] = current_value

        # Build cash flow series
        cash_flows = []
        dates = []

        for transaction in transactions:
            isin = transaction['isinNumber']
            date = datetime.strptime(transaction['transactionDate'], '%Y-%m-%dT%H:%M:%SZ'
            amount = float(transaction['transactionAmount']['units'])

            # Negative for investments (cash outflow), positive for redemptions
            if transaction['externalOrderType'] == 'BUY':
                cash_flows.append(-amount)
            else:
                cash_flows.append(amount)
            dates.append(date)

        # Add current values as final cash flow
        total_current_value = sum(current_values.values())
        cash_flows.append(total_current_value)
        dates.append(datetime.now())

        # Calculate XIRR using financial formula
        return self._xirr_calculation(dates, cash_flows)
```

6. Complete Code Examples and Best Practices

Production-Ready Fi MCP Client Implementation

Here's a robust, production-ready implementation that handles all edge cases and provides comprehensive error handling:

```
import asyncio
import aiohttp
import json
import uuid
import time
import logging
from typing import Dict, Any, Optional, List
from dataclasses import dataclass
from enum import Enum

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class AuthenticationState(Enum):
    """Authentication state enumeration"""
    NOT_AUTHENTICATED = "not_authenticated"
    AUTHENTICATING = "authenticating"
    AUTHENTICATED = "authenticated"
    AUTHENTICATION_FAILED = "authentication_failed"

@dataclass
class MCPResponse:
    """Structured MCP response with metadata"""
    success: bool
    data: Dict[str, Any]
    error_message: Optional[str] = None
    response_time_ms: int = 0
    cached: bool = False

class ProductionFiMCPClient:
    """Production-ready Fi MCP client with comprehensive error handling"""

    def __init__(self, base_url: str = "http://localhost:8080", timeout: int = 30):
        self.base_url = base_url
        self.timeout = timeout
        self.session_id = None
        self.auth_state = AuthenticationState.NOT_AUTHENTICATED
        self.response_cache = {}
        self.rate_limit_tracker = {}

        # Performance metrics
        self.metrics = {
            'total_requests': 0,
            'successful_requests': 0,
            'failed_requests': 0,
            'average_response_time': 0,
            'cache_hits': 0
```

```
        }

    async def authenticate_with_retry(self, phone_number: str, max_retries: int = 3) -> b
        """Authenticate with automatic retry logic"""

        for attempt in range(max_retries):
            try:
                self.auth_state = AuthenticationState.AUTHENTICATING
                success = await self._perform_authentication(phone_number)

                if success:
                    self.auth_state = AuthenticationState.AUTHENTICATED
                    logger.info(f"Authentication successful for {phone_number}")
                    return True
                else:
                    logger.warning(f"Authentication attempt {attempt + 1} failed for {pho

            except Exception as e:
                logger.error(f"Authentication error on attempt {attempt + 1}: {e}")

            if attempt < max_retries - 1:
                await asyncio.sleep(2 ** attempt)  # Exponential backoff

        self.auth_state = AuthenticationState.AUTHENTICATION_FAILED
        return False

    async def _perform_authentication(self, phone_number: str) -> bool:
        """Perform actual authentication steps"""

        # Generate unique session ID
        self.session_id = f"mcp-session-{uuid.uuid4()}"

        async with aiohttp.ClientSession(timeout=aiohttp.ClientTimeout(total=self.timeout
            # Step 1: Initiate authentication
            headers = {
                "Content-Type": "application/json",
                "Mcp-Session-Id": self.session_id
            }

            payload = {
                "jsonrpc": "2.0",
                "id": 1,
                "method": "tools/call",
                "params": {
                    "name": "fetch_net_worth",
                    "arguments": {}
                }
            }

            async with session.post(f"{self.base_url}/mcp/stream", headers=headers, json=
                if response.status != 200:
                    logger.error(f"Authentication initiation failed: HTTP {response.statu
                    return False

                result = await response.json()
                content = result.get("result", {}).get("content", [{}])[^0]
```

```
                login_data = json.loads(content.get("text", "{}"))

                if login_data.get("status") != "login_required":
                    logger.error("Unexpected authentication flow response")
                    return False

            # Step 2: Submit phone number
            login_data = {
                "sessionId": self.session_id,
                "phoneNumber": phone_number
            }

            async with session.post(
                f"{self.base_url}/login",
                data=login_data,
                headers={"Content-Type": "application/x-www-form-urlencoded"}
            ) as response:
                return response.status == 200

    async def call_tool_with_caching(self, tool_name: str, arguments: Optional[Dict] = No
                                     cache_ttl: int = 300) -> MCPResponse:
        """Call MCP tool with response caching and performance monitoring"""

        start_time = time.time()
        self.metrics['total_requests'] += 1

        # Check authentication
        if self.auth_state != AuthenticationState.AUTHENTICATED:
            return MCPResponse(
                success=False,
                data={},
                error_message="Not authenticated. Call authenticate_with_retry() first."
            )

        # Check cache
        cache_key = f"{tool_name}:{json.dumps(arguments or {}, sort_keys=True)}"
        if cache_key in self.response_cache:
            cache_entry = self.response_cache[cache_key]
            if time.time() - cache_entry['timestamp'] < cache_ttl:
                self.metrics['cache_hits'] += 1
                return MCPResponse(
                    success=True,
                    data=cache_entry['data'],
                    response_time_ms=int((time.time() - start_time) * 1000),
                    cached=True
                )

        # Rate limiting check
        if self._is_rate_limited(tool_name):
            return MCPResponse(
                success=False,
                data={},
                error_message="Rate limit exceeded. Please wait before retrying."
            )

        try:
```

```
            # Make actual API call
            response_data = await self._make_api_call(tool_name, arguments or {})

            # Cache successful response
            self.response_cache[cache_key] = {
                'data': response_data,
                'timestamp': time.time()
            }

            # Update metrics
            self.metrics['successful_requests'] += 1
            response_time = int((time.time() - start_time) * 1000)
            self._update_response_time_metric(response_time)

            return MCPResponse(
                success=True,
                data=response_data,
                response_time_ms=response_time
            )

        except Exception as e:
            self.metrics['failed_requests'] += 1
            logger.error(f"Tool call failed for {tool_name}: {e}")

            return MCPResponse(
                success=False,
                data={},
                error_message=str(e),
                response_time_ms=int((time.time() - start_time) * 1000)
            )

    async def _make_api_call(self, tool_name: str, arguments: Dict) -> Dict[str, Any]:
        """Make actual API call to Fi MCP server"""

        payload = {
            "jsonrpc": "2.0",
            "id": 1,
            "method": "tools/call",
            "params": {
                "name": tool_name,
                "arguments": arguments
            }
        }

        headers = {
            "Content-Type": "application/json",
            "Mcp-Session-Id": self.session_id
        }

        async with aiohttp.ClientSession(timeout=aiohttp.ClientTimeout(total=self.timeout
            async with session.post(f"{self.base_url}/mcp/stream", headers=headers, json=
                if response.status != 200:
                    raise Exception(f"HTTP {response.status}: {await response.text()}")

                result = await response.json()
```

```
                # Extract data from JSON-RPC response
                content = result.get("result", {}).get("content", [{}])[^0]
                return json.loads(content.get("text", "{}"))

    def _is_rate_limited(self, tool_name: str) -> bool:
        """Check if tool is rate limited"""
        current_time = time.time()

        if tool_name not in self.rate_limit_tracker:
            self.rate_limit_tracker[tool_name] = []

        # Clean old entries (beyond 1 minute)
        self.rate_limit_tracker[tool_name] = [
            timestamp for timestamp in self.rate_limit_tracker[tool_name]
            if current_time - timestamp < 60
        ]

        # Check if we've exceeded 10 requests per minute
        if len(self.rate_limit_tracker[tool_name]) >= 10:
            return True

        # Add current request
        self.rate_limit_tracker[tool_name].append(current_time)
        return False

    def _update_response_time_metric(self, response_time: int):
        """Update average response time metric"""
        current_avg = self.metrics['average_response_time']
        successful_requests = self.metrics['successful_requests']

        self.metrics['average_response_time'] = (
            (current_avg * (successful_requests - 1) + response_time) / successful_reques
        )

    def get_performance_metrics(self) -> Dict[str, Any]:
        """Get client performance metrics"""
        success_rate = (
            self.metrics['successful_requests'] / self.metrics['total_requests'] * 100
            if self.metrics['total_requests'] > 0 else 0
        )

        cache_hit_rate = (
            self.metrics['cache_hits'] / self.metrics['total_requests'] * 100
            if self.metrics['total_requests'] > 0 else 0
        )

        return {
            **self.metrics,
            'success_rate_percentage': round(success_rate, 2),
            'cache_hit_rate_percentage': round(cache_hit_rate, 2),
            'authentication_status': self.auth_state.value
        }

    async def health_check(self) -> bool:
        """Perform health check on Fi MCP server"""
        try:
```

```
            async with aiohttp.ClientSession(timeout=aiohttp.ClientTimeout(total=5)) as s
                async with session.get(f"{self.base_url}/health") as response:
                    return response.status == 200
        except:
            return False
```

Comprehensive Financial Agent with Error Recovery

```
class RobustFinancialAgent:
    """Production-ready financial agent with comprehensive error handling"""

    def __init__(self, mcp_base_url: str = "http://localhost:8080"):
        self.mcp_client = ProductionFiMCPClient(mcp_base_url)
        self.setup_agent()
        self.conversation_history = {}

    def setup_agent(self):
        """Setup ADK agent with robust error handling"""

        financial_tools = [
            self.create_resilient_data_fetcher(),
            self.create_smart_financial_analyzer(),
            self.create_personalized_advisor(),
            self.create_goal_tracker()
        ]

        self.agent = LlmAgent(
            model="gemini-2.0-flash",
            name="robust_financial_advisor",
            instruction="""
            You are a professional financial advisor with access to comprehensive Indian
            You provide personalized, actionable financial advice based on real user data

            CORE PRINCIPLES:
            1. **Data-Driven Decisions**: Base all recommendations on actual user financi
            2. **Risk Assessment**: Consider user's age, income, dependents, and risk tol
            3. **Goal-Oriented Planning**: Align advice with specific financial goals
            4. **Tax Efficiency**: Optimize for tax implications in all recommendations
            5. **Behavioral Guidance**: Provide practical steps user can take immediately

            ERROR HANDLING:
            - If data is incomplete, clearly state what information is missing
            - Provide general guidance when specific data isn't available
            - Always acknowledge data limitations in recommendations

            COMMUNICATION STYLE:
            - Use simple, clear language avoiding financial jargon
            - Provide specific numbers and calculations when possible
            - Explain the reasoning behind each recommendation
            - Offer alternatives for different risk tolerances
            """,
            tools=financial_tools,
            before_agent_callback=self.log_agent_interactions,
            after_agent_callback=self.process_agent_response
        )
```

```
        self.session_service = InMemorySessionService()
        self.runner = Runner(
            agent=self.agent,
            app_name="robust_financial_advisor",
            session_service=self.session_service
        )

    def create_resilient_data_fetcher(self):
        """Create data fetcher with automatic retry and graceful degradation"""

        async def fetch_financial_data_resilient(phone_number: str, tool_context: ToolCon
            """
            Fetch financial data with resilience patterns
            """

            # Store user phone number in state
            tool_context.state["user:phone_number"] = phone_number

            # Check if we need to authenticate
            if self.mcp_client.auth_state != AuthenticationState.AUTHENTICATED:
                auth_success = await self.mcp_client.authenticate_with_retry(phone_numbe
                if not auth_success:
                    return {
                        'status': 'authentication_failed',
                        'message': 'Could not authenticate with Fi MCP server',
                        'available_data': {}
                    }

            # Define data sources with priority levels
            data_sources = [
                ('fetch_net_worth', 'critical', 'Net worth and asset information'),
                ('fetch_credit_report', 'important', 'Credit score and loan details'),
                ('fetch_epf_details', 'important', 'EPF and retirement planning data'),
                ('fetch_mf_transactions', 'helpful', 'Mutual fund transaction history'),
                ('fetch_bank_transactions', 'helpful', 'Bank transaction patterns'),
                ('fetch_stock_transactions', 'optional', 'Stock trading history')
            ]

            fetched_data = {}
            fetch_summary = {
                'successful_fetches': 0,
                'failed_fetches': 0,
                'critical_data_available': True,
                'warnings': []
            }

            # Fetch data with error handling
            for tool_name, priority, description in data_sources:
                try:
                    response = await self.mcp_client.call_tool_with_caching(tool_name, ca

                    if response.success:
                        fetched_data[tool_name] = response.data
                        fetch_summary['successful_fetches'] += 1
```

```
                        if response.cached:
                            fetch_summary['warnings'].append(f"{description} loaded from
                    else:
                        fetch_summary['failed_fetches'] += 1
                        if priority == 'critical':
                            fetch_summary['critical_data_available'] = False

                        fetch_summary['warnings'].append(f"Could not fetch {description}:

                except Exception as e:
                    fetch_summary['failed_fetches'] += 1
                    fetch_summary['warnings'].append(f"Error fetching {description}: {str

            # Store fetched data in session state
            tool_context.state[f"financial_data_{phone_number}"] = fetched_data
            tool_context.state["data_fetch_summary"] = fetch_summary
            tool_context.state["last_data_fetch"] = time.time()

            return {
                'status': 'success' if fetch_summary['critical_data_available'] else 'par
                'data': fetched_data,
                'summary': fetch_summary
            }

        return fetch_financial_data_resilient

    def create_smart_financial_analyzer(self):
        """Create intelligent financial analyzer that works with partial data"""

        def analyze_financial_health(tool_context: ToolContext) -> dict:
            """
            Perform financial health analysis with graceful degradation
            """

            phone_number = tool_context.state.get("user:phone_number")
            financial_data = tool_context.state.get(f"financial_data_{phone_number}", {})

            if not financial_data:
                return {
                    'error': 'No financial data available. Please fetch data first.',
                    'analysis_possible': False
                }

            analysis = {
                'overall_score': 0,
                'strengths': [],
                'areas_for_improvement': [],
                'immediate_actions': [],
                'long_term_strategies': [],
                'data_completeness': self._assess_data_completeness(financial_data)
            }

            # Net worth analysis (if available)
            if 'fetch_net_worth' in financial_data:
                net_worth_analysis = self._analyze_net_worth(financial_data['fetch_net_wo
                analysis.update(net_worth_analysis)
```

```
                analysis['overall_score'] += 30
            else:
                analysis['areas_for_improvement'].append('Connect accounts to track net w

            # Credit health analysis
            if 'fetch_credit_report' in financial_data:
                credit_analysis = self._analyze_credit_health(financial_data['fetch_credi
                analysis.update(credit_analysis)
                analysis['overall_score'] += 25
            else:
                analysis['immediate_actions'].append('Check and monitor credit score')

            # Investment analysis
            if 'fetch_mf_transactions' in financial_data:
                investment_analysis = self._analyze_investment_performance(financial_data
                analysis.update(investment_analysis)
                analysis['overall_score'] += 25

            # EPF analysis
            if 'fetch_epf_details' in financial_data:
                epf_analysis = self._analyze_retirement_readiness(financial_data['fetch_e
                analysis.update(epf_analysis)
                analysis['overall_score'] += 20

            # Normalize score to 100
            analysis['overall_score'] = min(100, analysis['overall_score'])

            # Add contextual recommendations
            analysis['personalized_recommendations'] = self._generate_personalized_recomm
                financial_data, analysis
            )

            return analysis

        return analyze_financial_health

    def log_agent_interactions(self, callback_context, new_message):
        """Log all agent interactions for debugging"""
        user_id = callback_context.state.get("user:phone_number", "unknown")
        message_text = new_message.parts[^0].text[:100] + "..." if len(new_message.parts[

        logger.info(f"Agent interaction - User: {user_id}, Message: {message_text}")

        # Store conversation history
        if user_id not in self.conversation_history:
            self.conversation_history[user_id] = []

        self.conversation_history[user_id].append({
            'timestamp': time.time(),
            'type': 'user_message',
            'content': new_message.parts[^0].text
        })

        return None  # Proceed with normal processing

    def process_agent_response(self, callback_context, agent_response):
```

```
        """Process and log agent responses"""
        user_id = callback_context.state.get("user:phone_number", "unknown")

        if user_id in self.conversation_history:
            self.conversation_history[user_id].append({
                'timestamp': time.time(),
                'type': 'agent_response',
                'content': agent_response.parts[^0].text if agent_response.parts else "No
            })

        # Add performance metrics to response
        metrics = self.mcp_client.get_performance_metrics()
        callback_context.state["system:performance_metrics"] = metrics

        return agent_response

    async def get_comprehensive_financial_advice(self, phone_number: str, user_query: str
        """Main entry point for comprehensive financial advice"""

        try:
            # Create or get existing session
            session = await self.session_service.create_session(
                app_name="robust_financial_advisor",
                user_id=phone_number
            )

            # Prepare comprehensive query
            enhanced_query = f"""
            Phone Number: {phone_number}
            User Query: {user_query}

            Please provide comprehensive financial advice based on this user's complete f
            Start by fetching their financial data, then analyze their financial health,
            actionable recommendations.
            """

            content = types.Content(role="user", parts=[types.Part(text=enhanced_query)])
            response_text = ""

            # Process through ADK agent
            async for event in self.runner.run_async(
                user_id=phone_number,
                session_id=session.id,
                new_message=content
            ):
                if hasattr(event, 'is_final_response') and event.is_final_response():
                    if hasattr(event, 'content') and event.content is not None:
                        if hasattr(event.content, 'parts') and event.content.parts:
                            response_text = ''.join([
                                part.text for part in event.content.parts
                                if hasattr(part, 'text') and part.text
                            ])
                            break

            # Add performance summary
            metrics = self.mcp_client.get_performance_metrics()
```

```
            response_text += f"\n\n---\nSystem Performance: {metrics['success_rate_percen

            return response_text

        except Exception as e:
            logger.error(f"Error in comprehensive financial advice: {e}")
            return f"I apologize, but I encountered an error while analyzing your financi

# Usage example
async def main():
    """Example usage of robust financial agent"""

    agent = RobustFinancialAgent()

    # Example comprehensive financial consultation
    phone_number = "2222222222"  # Comprehensive portfolio user
    user_query = "I want to plan for early retirement by age 40. Can you analyze my curre

    advice = await agent.get_comprehensive_financial_advice(phone_number, user_query)
    print("="*80)
    print("COMPREHENSIVE FINANCIAL ADVICE")
    print("="*80)
    print(advice)
    print("="*80)

    # Show performance metrics
    metrics = agent.mcp_client.get_performance_metrics()
    print("\nSystem Performance Metrics:")
    for key, value in metrics.items():
        print(f"  {key}: {value}")

if __name__ == "__main__":
    asyncio.run(main())
```

7. Production-Ready Agent Implementations

Specialized Financial Agent Architectures

Different financial use cases require specialized agent architectures. Here are production-ready implementations for specific scenarios:

Debt Management Specialist Agent

```
class DebtManagementAgent:
    """Specialized agent for debt optimization and credit improvement"""

    def __init__(self, mcp_client):
        self.mcp_client = mcp_client
        self.setup_debt_specialist()

    def setup_debt_specialist(self):
        """Configure agent specialized in debt management"""
```

```
        debt_tools = [
            self.create_debt_consolidation_analyzer(),
            self.create_credit_score_optimizer(),
            self.create_payment_strategy_planner(),
            self.create_emergency_fund_calculator()
        ]

        self.agent = LlmAgent(
            model="gemini-2.0-flash",
            name="debt_management_specialist",
            instruction="""
            You are a debt management and credit improvement specialist. Your expertise f

            1. **Debt Consolidation**: Analyze multiple debts and suggest optimal consoli
            2. **Credit Score Improvement**: Provide actionable steps to improve credit s
            3. **Payment Prioritization**: Use debt avalanche/snowball methods based on u
            4. **Emergency Fund Planning**: Build emergency funds even while paying off d

            DEBT ANALYSIS METHODOLOGY:
            - Calculate total debt burden and debt-to-income ratios
            - Identify highest interest rate debts for priority repayment
            - Assess minimum payment requirements vs. available cash flow
            - Consider balance transfer and consolidation opportunities

            COMMUNICATION APPROACH:
            - Be empathetic about debt stress while staying solution-focused
            - Provide step-by-step action plans with specific amounts and timelines
            - Explain the psychological benefits of different debt repayment strategies
            - Celebrate small wins and progress milestones
            """,
            tools=debt_tools
        )

    def create_debt_consolidation_analyzer(self):
        """Analyze debt consolidation opportunities"""

        def analyze_debt_consolidation(tool_context: ToolContext) -> dict:
            """
            Analyze opportunities for debt consolidation
            """

            phone_number = tool_context.state.get("user:phone_number")
            financial_data = tool_context.state.get(f"financial_data_{phone_number}", {})

            credit_data = financial_data.get('fetch_credit_report', {})
            net_worth_data = financial_data.get('fetch_net_worth', {})

            if not credit_data or not net_worth_data:
                return {'error': 'Insufficient data for debt consolidation analysis'}

            # Extract all debts from credit report and net worth
            debts = []

            # From credit report (detailed loan information)
            if credit_data.get('creditReports'):
                credit_accounts = credit_data['creditReports'][^0]['creditReportData']['c
```

```
            for account in credit_accounts:
                debt_info = {
                    'lender': account['subscriberName'],
                    'type': self._get_loan_type(account['accountType']),
                    'balance': float(account['currentBalance']),
                    'interest_rate': float(account.get('rateOfInterest', 0)),
                    'past_due': float(account.get('amountPastDue', 0)),
                    'payment_status': account.get('paymentRating', '0')
                }

                if debt_info['balance'] > 0:
                    debts.append(debt_info)

        # From net worth (liability summary)
        if net_worth_data.get('netWorthResponse', {}).get('liabilityValues'):
            for liability in net_worth_data['netWorthResponse']['liabilityValues']:
                liability_type = liability['netWorthAttribute']
                balance = float(liability['value']['units'])

                # Check if already included from credit report
                if not any(d['balance'] == balance for d in debts):
                    debts.append({
                        'type': liability_type,
                        'balance': balance,
                        'interest_rate': self._estimate_interest_rate(liability_type)
                        'lender': 'Unknown'
                    })

        # Calculate consolidation scenarios
        total_debt = sum(d['balance'] for d in debts)
        weighted_avg_interest = sum(d['balance'] * d['interest_rate'] for d in debts)

        # Debt prioritization strategies
        debt_avalanche = sorted(debts, key=lambda x: x['interest_rate'], reverse=True
        debt_snowball = sorted(debts, key=lambda x: x['balance'])

        # Consolidation recommendations
        consolidation_options = []

        # Personal loan consolidation
        if len(debts) > 2:
            estimated_personal_loan_rate = 11.0  # Average personal loan rate
            if estimated_personal_loan_rate < weighted_avg_interest:
                monthly_savings = (weighted_avg_interest - estimated_personal_loan_ra
                consolidation_options.append({
                    'type': 'personal_loan',
                    'description': 'Consolidate all debts into single personal loan',
                    'estimated_rate': estimated_personal_loan_rate,
                    'monthly_savings': monthly_savings,
                    'pros': ['Single payment', 'Lower average interest', 'Fixed repay
                    'cons': ['Requires good credit score', 'May have processing fees'
                })

        # Balance transfer for credit cards
        credit_card_debts = [d for d in debts if d['type'] == 'credit_card']
```

```
            if credit_card_debts:
                consolidation_options.append({
                    'type': 'balance_transfer',
                    'description': 'Transfer credit card balances to low-interest card',
                    'estimated_rate': 6.0,  # Promotional rate
                    'monthly_savings': sum(d['balance'] * (d['interest_rate'] - 6.0) / 12
                    'pros': ['0% promotional periods available', 'Simplifies payments'],
                    'cons': ['Promotional rates are temporary', 'Balance transfer fees']
                })

            return {
                'total_debt': total_debt,
                'number_of_debts': len(debts),
                'weighted_average_interest': weighted_avg_interest,
                'debt_breakdown': debts,
                'prioritization_strategies': {
                    'debt_avalanche': debt_avalanche,
                    'debt_snowball': debt_snowball
                },
                'consolidation_options': consolidation_options,
                'recommendation': self._generate_debt_recommendation(debts, consolidation
            }

        return analyze_debt_consolidation

    def create_credit_score_optimizer(self):
        """Create credit score improvement strategy"""

        def optimize_credit_score(target_score: int, tool_context: ToolContext) -> dict:
            """
            Create personalized credit score improvement plan
            """

            phone_number = tool_context.state.get("user:phone_number")
            financial_data = tool_context.state.get(f"financial_data_{phone_number}", {})

            credit_data = financial_data.get('fetch_credit_report', {})

            if not credit_data or not credit_data.get('creditReports'):
                return {'error': 'Credit report data not available'}

            credit_report = credit_data['creditReports'][^0]['creditReportData']
            current_score = int(credit_report['score']['bureauScore'])

            improvement_plan = {
                'current_score': current_score,
                'target_score': target_score,
                'improvement_needed': target_score - current_score,
                'action_items': [],
                'timeline': '6-12 months',
                'monthly_tasks': []
            }

            # Analyze credit utilization
            credit_accounts = credit_report['creditAccount']['creditAccountDetails']
            credit_cards = [acc for acc in credit_accounts if acc['accountType'] == '10']
```

```
        if credit_cards:
            total_limit = sum(float(acc.get('creditLimitAmount', 0)) for acc in credi
            total_balance = sum(float(acc['currentBalance']) for acc in credit_cards)
            utilization_ratio = (total_balance / total_limit * 100) if total_limit >

            if utilization_ratio > 30:
                improvement_plan['action_items'].append({
                    'priority': 'HIGH',
                    'action': f'Reduce credit utilization from {utilization_ratio:.1f
                    'impact': '+50 to +100 points',
                    'method': 'Pay down balances or request credit limit increases'
                })

        # Analyze payment history
        accounts_with_late_payments = [
            acc for acc in credit_accounts
            if acc.get('paymentRating', '0') != '0'
        ]

        if accounts_with_late_payments:
            improvement_plan['action_items'].append({
                'priority': 'CRITICAL',
                'action': 'Ensure all future payments are made on time',
                'impact': '+35 to +50 points over 6 months',
                'method': 'Set up automatic payments for at least minimum amounts'
            })

        # Analyze credit mix
        loan_types = set(acc['accountType'] for acc in credit_accounts)
        if len(loan_types) < 3:
            improvement_plan['action_items'].append({
                'priority': 'LOW',
                'action': 'Consider diversifying credit mix',
                'impact': '+10 to +20 points',
                'method': 'Add different types of credit (installment loan, secured c
            })

        # Generate monthly task schedule
        improvement_plan['monthly_tasks'] = [
            'Check credit report for errors and dispute if found',
            'Pay all bills on time, especially credit accounts',
            'Keep credit utilization below 30% (ideally below 10%)',
            'Avoid applying for new credit unless necessary',
            'Monitor credit score changes monthly'
        ]

        return improvement_plan

    return optimize_credit_score
```

Investment Advisory Agent

```
class InvestmentAdvisoryAgent:
    """Specialized agent for investment analysis and portfolio optimization"""

    def __init__(self, mcp_client):
        self.mcp_client = mcp_client
        self.setup_investment_advisor()

    def setup_investment_advisor(self):
        """Configure investment specialist agent"""

        investment_tools = [
            self.create_portfolio_analyzer(),
            self.create_asset_allocation_optimizer(),
            self.create_sip_strategy_planner(),
            self.create_tax_efficient_rebalancer()
        ]

        self.agent = LlmAgent(
            model="gemini-2.0-flash",
            name="investment_advisory_specialist",
            instruction="""
            You are an investment advisory specialist with expertise in Indian markets ar

            CORE COMPETENCIES:
            1. **Portfolio Analysis**: Calculate returns, risk metrics, and performance a
            2. **Asset Allocation**: Optimize allocation based on age, goals, and risk to
            3. **Fund Selection**: Recommend specific funds based on category, performanc
            4. **Tax Optimization**: Suggest tax-efficient investment strategies

            INVESTMENT PHILOSOPHY:
            - Long-term wealth creation through disciplined investing
            - Diversification across asset classes and fund houses
            - Cost consciousness (expense ratios, exit loads, taxes)
            - Regular monitoring and rebalancing

            ANALYSIS FRAMEWORK:
            - Use actual XIRR calculations from transaction data
            - Compare performance against relevant benchmarks
            - Consider risk-adjusted returns (Sharpe ratio when possible)
            - Factor in tax implications for all recommendations
            """,
            tools=investment_tools
        )

    def create_portfolio_analyzer(self):
        """Comprehensive portfolio analysis tool"""

        def analyze_investment_portfolio(tool_context: ToolContext) -> dict:
            """
            Perform comprehensive portfolio analysis
            """

            phone_number = tool_context.state.get("user:phone_number")
            financial_data = tool_context.state.get(f"financial_data_{phone_number}", {})
```

```
        net_worth_data = financial_data.get('fetch_net_worth', {})
        transaction_data = financial_data.get('fetch_mf_transactions', {})

        if not net_worth_data or not net_worth_data.get('mfSchemeAnalytics'):
            return {'error': 'No mutual fund portfolio data available'}

        mf_analytics = net_worth_data['mfSchemeAnalytics']['schemeAnalytics']

        portfolio_analysis = {
            'summary': {},
            'asset_allocation': {},
            'performance_metrics': {},
            'fund_analysis': [],
            'risk_assessment': {},
            'recommendations': []
        }

        # Portfolio summary
        total_current_value = 0
        total_invested_value = 0

        for scheme in mf_analytics:
            analytics = scheme['enrichedAnalytics']['analytics']['schemeDetails']
            current_value = float(analytics['currentValue']['units'])
            invested_value = float(analytics.get('investedValue', {}).get('units', 0)

            total_current_value += current_value
            total_invested_value += invested_value

        portfolio_analysis['summary'] = {
            'total_current_value': total_current_value,
            'total_invested_value': total_invested_value,
            'absolute_returns': total_current_value - total_invested_value,
            'returns_percentage': ((total_current_value - total_invested_value) / tot
            'number_of_funds': len(mf_analytics)
        }

        # Asset allocation analysis
        asset_allocation = {'EQUITY': 0, 'DEBT': 0, 'HYBRID': 0, 'CASH': 0}
        fund_house_allocation = {}

        for scheme in mf_analytics:
            analytics = scheme['enrichedAnalytics']['analytics']['schemeDetails']
            current_value = float(analytics['currentValue']['units'])
            asset_class = scheme['schemeDetail']['assetClass']
            fund_house = scheme['schemeDetail']['amc']

            asset_allocation[asset_class] += current_value
            fund_house_allocation[fund_house] = fund_house_allocation.get(fund_house,

        # Convert to percentages
        for asset_class in asset_allocation:
            asset_allocation[asset_class] = (asset_allocation[asset_class] / total_cu

        portfolio_analysis['asset_allocation'] = {
```

```
                'by_asset_class': asset_allocation,
                'by_fund_house': {fh: (value / total_current_value * 100) for fh, value i
            }

            # Individual fund analysis
            for scheme in mf_analytics:
                fund_analysis = self._analyze_individual_fund(scheme, transaction_data)
                portfolio_analysis['fund_analysis'].append(fund_analysis)

            # Performance metrics
            portfolio_analysis['performance_metrics'] = self._calculate_portfolio_perform

            # Risk assessment
            portfolio_analysis['risk_assessment'] = self._assess_portfolio_risk(mf_analyt

            # Generate recommendations
            portfolio_analysis['recommendations'] = self._generate_portfolio_recommendati

            return portfolio_analysis

        return analyze_investment_portfolio

    def create_asset_allocation_optimizer(self):
        """Optimize asset allocation based on user profile"""

        def optimize_asset_allocation(age: int, risk_tolerance: str, investment_horizon:
                                      monthly_investment: float, tool_context: ToolContext)
            """
            Generate optimal asset allocation recommendation

            Args:
                age: User's current age
                risk_tolerance: conservative, moderate, aggressive
                investment_horizon: Investment timeline in years
                monthly_investment: Monthly SIP amount
            """

            # Age-based equity allocation (100 minus age rule as starting point)
            base_equity_allocation = max(30, min(90, 100 - age))

            # Adjust based on risk tolerance
            risk_adjustments = {
                'conservative': -20,
                'moderate': 0,
                'aggressive': +15
            }

            equity_allocation = max(20, min(90, base_equity_allocation + risk_adjustments

            # Adjust based on investment horizon
            if investment_horizon > 15:
                equity_allocation = min(90, equity_allocation + 10)
            elif investment_horizon < 5:
                equity_allocation = max(30, equity_allocation - 15)

            debt_allocation = 100 - equity_allocation
```

```
            # Generate specific fund recommendations
            recommendations = {
                'target_allocation': {
                    'equity': equity_allocation,
                    'debt': debt_allocation
                },
                'fund_recommendations': {
                    'equity_funds': self._recommend_equity_funds(equity_allocation, month
                    'debt_funds': self._recommend_debt_funds(debt_allocation, monthly_inv
                },
                'implementation_strategy': self._create_implementation_strategy(
                    equity_allocation, debt_allocation, monthly_investment
                ),
                'rebalancing_schedule': 'Review quarterly, rebalance if allocation differ
                'tax_considerations': self._analyze_tax_implications(equity_allocation, d
            }

            # Store optimal allocation in state for future reference
            tool_context.state["user:optimal_allocation"] = recommendations['target_alloc
            tool_context.state["user:risk_profile"] = {
                'age': age,
                'risk_tolerance': risk_tolerance,
                'investment_horizon': investment_horizon
            }

            return recommendations

        return optimize_asset_allocation

    def _recommend_equity_funds(self, equity_allocation: float, monthly_investment: float
        """Recommend specific equity funds based on allocation"""

        equity_amount = monthly_investment * equity_allocation / 100

        recommendations = []

        if equity_amount >= 1000:
            # Large cap + Mid cap + Small cap diversification
            recommendations.extend([
                {
                    'category': 'Large Cap Index Fund',
                    'suggested_funds': ['ICICI Prudential Nifty 50 Index Fund', 'UTI Nift
                    'allocation_percentage': 40,
                    'monthly_sip': equity_amount * 0.4,
                    'rationale': 'Stable large cap exposure with low cost'
                },
                {
                    'category': 'Flexi Cap Fund',
                    'suggested_funds': ['Parag Parikh Flexi Cap Fund', 'PGIM India Flexi
                    'allocation_percentage': 35,
                    'monthly_sip': equity_amount * 0.35,
                    'rationale': 'Flexibility to invest across market caps'
                },
                {
                    'category': 'Mid Cap Fund',
```

```
                'suggested_funds': ['Axis Midcap Fund', 'DSP Midcap Fund'],
                'allocation_percentage': 25,
                'monthly_sip': equity_amount * 0.25,
                'rationale': 'Higher growth potential from mid-cap companies'
            }
        ])
    else:
        # Single fund recommendation for smaller amounts
        recommendations.append({
            'category': 'Hybrid Aggressive Fund',
            'suggested_funds': ['ICICI Prudential Equity & Debt Fund', 'SBI Equity Hy
            'allocation_percentage': 100,
            'monthly_sip': equity_amount,
            'rationale': 'Single fund providing equity exposure with some debt stabil
        })

    return recommendations

def _recommend_debt_funds(self, debt_allocation: float, monthly_investment: float) ->
    """Recommend specific debt funds"""

    debt_amount = monthly_investment * debt_allocation / 100

    if debt_amount < 500:
        return [{
            'category': 'Hybrid Conservative Fund',
            'suggested_funds': ['ICICI Prudential Balanced Advantage Fund'],
            'allocation_percentage': 100,
            'monthly_sip': debt_amount,
            'rationale': 'Conservative hybrid for small debt allocation'
        }]

    return [
        {
            'category': 'Short Duration Fund',
            'suggested_funds': ['HDFC Short Term Debt Fund', 'ICICI Prudential Short
            'allocation_percentage': 60,
            'monthly_sip': debt_amount * 0.6,
            'rationale': 'Low interest rate risk with decent returns'
        },
        {
            'category': 'Corporate Bond Fund',
            'suggested_funds': ['HDFC Corporate Bond Fund', 'ICICI Prudential Corpora
            'allocation_percentage': 40,
            'monthly_sip': debt_amount * 0.4,
            'rationale': 'Higher yield from corporate bonds'
        }
    ]
```

8. Error Handling and Edge Cases

Comprehensive Error Handling Patterns

Financial applications require robust error handling due to the sensitive nature of financial data and the potential impact of incorrect information. Here are comprehensive error handling patterns for Fi MCP agents:

Data Validation and Sanitization

```
class FinancialDataValidator:
    """Comprehensive data validation for financial information"""

    @staticmethod
    def validate_currency_amount(amount_data: dict) -> tuple[bool, float, str]:
        """
        Validate and parse currency amount from Fi MCP format

        Returns:
            (is_valid, parsed_amount, error_message)
        """

        if not isinstance(amount_data, dict):
            return False, 0.0, "Amount data must be a dictionary"

        if 'currencyCode' not in amount_data:
            return False, 0.0, "Currency code missing"

        if amount_data['currencyCode'] != 'INR':
            return False, 0.0, f"Unsupported currency: {amount_data['currencyCode']}"

        if 'units' not in amount_data:
            return False, 0.0, "Units field missing"

        try:
            units = float(amount_data['units'])
            nanos = amount_data.get('nanos', 0)

            if isinstance(nanos, str):
                nanos = float(nanos)

            total_amount = units + (nanos / 1_000_000_000)

            if total_amount < 0:
                return False, 0.0, "Negative amounts not allowed in this context"

            if total_amount > 100_000_000_000:  # 100 crore limit
                return False, 0.0, "Amount exceeds maximum limit"

            return True, total_amount, ""

        except (ValueError, TypeError) as e:
            return False, 0.0, f"Invalid amount format: {str(e)}"
```

```python
    @staticmethod
    def validate_credit_score(score_data: dict) -> tuple[bool, int, str]:
        """Validate credit score data"""

        if not isinstance(score_data, dict):
            return False, 0, "Credit score data must be a dictionary"

        if 'bureauScore' not in score_data:
            return False, 0, "Bureau score missing"

        try:
            score = int(score_data['bureauScore'])

            if score < 300 or score > 900:
                return False, 0, f"Credit score {score} outside valid range (300-900)"

            confidence = score_data.get('bureauScoreConfidenceLevel', 'L')
            if confidence not in ['H', 'M', 'L']:
                return False, 0, f"Invalid confidence level: {confidence}"

            return True, score, ""

        except (ValueError, TypeError):
            return False, 0, "Invalid credit score format"

    @staticmethod
    def validate_date_format(date_string: str, expected_format: str = None) -> tuple[bool
        """Validate and parse date strings from Fi MCP"""

        if not isinstance(date_string, str):
            return False, None, "Date must be a string"

        date_formats = [
            '%Y-%m-%dT%H:%M:%SZ',  # ISO format
            '%Y%m%d',              # Credit report format
            '%d-%m-%Y',            # EPF format
        ]

        if expected_format:
            date_formats.insert(0, expected_format)

        for fmt in date_formats:
            try:
                parsed_date = datetime.strptime(date_string, fmt)

                # Validate date range (not too far in past or future)
                current_date = datetime.now()
                min_date = current_date - timedelta(days=365 * 100)  # 100 years ago
                max_date = current_date + timedelta(days=365 * 10)   # 10 years ahead

                if parsed_date < min_date or parsed_date > max_date:
                    return False, None, f"Date {date_string} outside valid range"

                return True, parsed_date, ""

            except ValueError:
```

```
                continue

        return False, None, f"Unable to parse date: {date_string}"

class RobustFinancialDataProcessor:
    """Robust processor for Fi MCP data with comprehensive error handling"""

    def __init__(self):
        self.validator = FinancialDataValidator()
        self.processing_errors = []

    def process_net_worth_data(self, net_worth_response: dict) -> dict:
        """Process net worth data with validation and error recovery"""

        self.processing_errors = []
        processed_data = {
            'total_assets': 0.0,
            'total_liabilities': 0.0,
            'net_worth': 0.0,
            'asset_breakdown': {},
            'liability_breakdown': {},
            'data_quality': 'good',
            'warnings': []
        }

        if not isinstance(net_worth_response, dict):
            processed_data['data_quality'] = 'error'
            self.processing_errors.append('Net worth response is not a dictionary')
            return processed_data

        net_worth_data = net_worth_response.get('netWorthResponse', {})

        # Process assets
        assets = net_worth_data.get('assetValues', [])
        if not isinstance(assets, list):
            self.processing_errors.append('Asset values is not a list')
        else:
            for asset in assets:
                self._process_asset_item(asset, processed_data)

        # Process liabilities
        liabilities = net_worth_data.get('liabilityValues', [])
        if not isinstance(liabilities, list):
            self.processing_errors.append('Liability values is not a list')
        else:
            for liability in liabilities:
                self._process_liability_item(liability, processed_data)

        # Calculate net worth
        processed_data['net_worth'] = processed_data['total_assets'] - processed_data['to

        # Validate against reported total (if available)
        if 'totalNetWorthValue' in net_worth_data:
            is_valid, reported_total, error = self.validator.validate_currency_amount(
                net_worth_data['totalNetWorthValue']
            )
```

```
            if is_valid:
                difference = abs(processed_data['net_worth'] - reported_total)
                if difference > 1000:  # More than ₹1000 difference
                    processed_data['warnings'].append(
                        f"Calculated net worth (₹{processed_data['net_worth']:,.0f}) dif
                        f"reported total (₹{reported_total:,.0f}) by ₹{difference:,.0f}"
                    )

        # Assess data quality
        if len(self.processing_errors) > 0:
            processed_data['data_quality'] = 'error'
        elif len(processed_data['warnings']) > 2:
            processed_data['data_quality'] = 'warning'
        elif processed_data['total_assets'] == 0:
            processed_data['data_quality'] = 'incomplete'
            processed_data['warnings'].append('No asset data found')

        processed_data['processing_errors'] = self.processing_errors

        return processed_data

    def _process_asset_item(self, asset: dict, processed_data: dict):
        """Process individual asset item with error handling"""

        if not isinstance(asset, dict):
            self.processing_errors.append(f'Asset item is not a dictionary: {asset}')
            return

        asset_type = asset.get('netWorthAttribute')
        if not asset_type:
            self.processing_errors.append('Asset missing netWorthAttribute')
            return

        value_data = asset.get('value', {})
        is_valid, amount, error = self.validator.validate_currency_amount(value_data)

        if is_valid:
            processed_data['total_assets'] += amount
            processed_data['asset_breakdown'][asset_type] = amount
        else:
            self.processing_errors.append(f'Invalid asset value for {asset_type}: {error}

    def _process_liability_item(self, liability: dict, processed_data: dict):
        """Process individual liability item with error handling"""

        if not isinstance(liability, dict):
            self.processing_errors.append(f'Liability item is not a dictionary: {liabilit
            return

        liability_type = liability.get('netWorthAttribute')
        if not liability_type:
            self.processing_errors.append('Liability missing netWorthAttribute')
            return

        value_data = liability.get('value', {})
```

```
        is_valid, amount, error = self.validator.validate_currency_amount(value_data)

        if is_valid:
            processed_data['total_liabilities'] += amount
            processed_data['liability_breakdown'][liability_type] = amount
        else:
            self.processing_errors.append(f'Invalid liability value for {liability_type}:

    def process_credit_report_data(self, credit_response: dict) -> dict:
        """Process credit report data with comprehensive validation"""

        self.processing_errors = []
        processed_data = {
            'credit_score': 0,
            'confidence_level': 'L',
            'total_accounts': 0,
            'active_accounts': 0,
            'total_outstanding': 0.0,
            'accounts': [],
            'recent_inquiries': 0,
            'data_quality': 'good',
            'warnings': []
        }

        if not isinstance(credit_response, dict):
            processed_data['data_quality'] = 'error'
            self.processing_errors.append('Credit response is not a dictionary')
            return processed_data

        credit_reports = credit_response.get('creditReports', [])
        if not credit_reports:
            processed_data['data_quality'] = 'incomplete'
            processed_data['warnings'].append('No credit reports found')
            return processed_data

        credit_data = credit_reports[^0].get('creditReportData', {})

        # Process credit score
        score_data = credit_data.get('score', {})
        is_valid, score, error = self.validator.validate_credit_score(score_data)

        if is_valid:
            processed_data['credit_score'] = score
            processed_data['confidence_level'] = score_data.get('bureauScoreConfidenceLev
        else:
            self.processing_errors.append(f'Invalid credit score: {error}')

        # Process credit accounts
        credit_account_data = credit_data.get('creditAccount', {})
        account_details = credit_account_data.get('creditAccountDetails', [])

        for account in account_details:
            processed_account = self._process_credit_account(account)
            if processed_account:
                processed_data['accounts'].append(processed_account)
```

```
        processed_data['total_accounts'] = len(processed_data['accounts'])
        processed_data['active_accounts'] = len([a for a in processed_data['accounts'] if
        processed_data['total_outstanding'] = sum(a['current_balance'] for a in processed

        # Process inquiries
        caps_data = credit_data.get('caps', {})
        caps_summary = caps_data.get('capsSummary', {})
        processed_data['recent_inquiries'] = int(caps_summary.get('capsLast90Days', 0))

        # Assess data quality
        if len(self.processing_errors) > 0:
            processed_data['data_quality'] = 'error'
        elif processed_data['credit_score'] == 0:
            processed_data['data_quality'] = 'incomplete'

        processed_data['processing_errors'] = self.processing_errors

        return processed_data

    def _process_credit_account(self, account: dict) -> dict:
        """Process individual credit account"""

        if not isinstance(account, dict):
            return None

        try:
            processed_account = {
                'lender': account.get('subscriberName', 'Unknown'),
                'account_type': account.get('accountType', ''),
                'current_balance': float(account.get('currentBalance', 0)),
                'credit_limit': float(account.get('creditLimitAmount', 0)),
                'interest_rate': float(account.get('rateOfInterest', 0)),
                'payment_status': account.get('paymentRating', '0'),
                'account_status': account.get('accountStatus', ''),
                'is_active': account.get('accountStatus') in ['11', '21', '71'],  # Activ
                'past_due_amount': float(account.get('amountPastDue', 0))
            }

            # Calculate utilization for credit cards
            if processed_account['account_type'] == '10' and processed_account['credit_li
                processed_account['utilization_ratio'] = (
                    processed_account['current_balance'] / processed_account['credit_limi
                )

            return processed_account

        except (ValueError, TypeError) as e:
            self.processing_errors.append(f'Error processing credit account: {str(e)}')
            return None
```

API Circuit Breaker and Retry Logic

```
import asyncio
from typing import Callable, Any
from enum import Enum
import time
import random

class CircuitState(Enum):
    CLOSED = "closed"      # Normal operation
    OPEN = "open"          # Circuit breaker triggered
    HALF_OPEN = "half_open"  # Testing if service recovered

class CircuitBreaker:
    """Circuit breaker for API calls with exponential backoff"""

    def __init__(self, failure_threshold: int = 5, recovery_timeout: int = 60,
                 expected_exception: tuple = (Exception,)):
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.expected_exception = expected_exception

        self.failure_count = 0
        self.last_failure_time = 0
        self.state = CircuitState.CLOSED

    async def call(self, func: Callable, *args, **kwargs) -> Any:
        """Execute function with circuit breaker protection"""

        if self.state == CircuitState.OPEN:
            if time.time() - self.last_failure_time > self.recovery_timeout:
                self.state = CircuitState.HALF_OPEN
            else:
                raise Exception("Circuit breaker is OPEN")

        try:
            result = await func(*args, **kwargs)
            self._on_success()
            return result

        except self.expected_exception as e:
            self._on_failure()
            raise e

    def _on_success(self):
        """Handle successful call"""
        self.failure_count = 0
        self.state = CircuitState.CLOSED

    def _on_failure(self):
        """Handle failed call"""
        self.failure_count += 1
        self.last_failure_time = time.time()

        if self.failure_count >= self.failure_threshold:
            self.state = CircuitState.OPEN
```

```
class RetryableFiMCPClient:
    """Fi MCP client with advanced retry and circuit breaker patterns"""

    def __init__(self, base_url: str = "http://localhost:8080"):
        self.base_url = base_url
        self.session_id = None
        self.authenticated = False

        # Circuit breakers for different tool types
        self.circuit_breakers = {
            'fetch_net_worth': CircuitBreaker(failure_threshold=3, recovery_timeout=30),
            'fetch_credit_report': CircuitBreaker(failure_threshold=3, recovery_timeout=6
            'fetch_epf_details': CircuitBreaker(failure_threshold=5, recovery_timeout=45)
            'fetch_mf_transactions': CircuitBreaker(failure_threshold=5, recovery_timeout
            'fetch_bank_transactions': CircuitBreaker(failure_threshold=5, recovery_timed
            'fetch_stock_transactions': CircuitBreaker(failure_threshold=5, recovery_time
        }

    async def call_tool_with_retry(self, tool_name: str, arguments: dict = None,
                                   max_retries: int = 3) -> dict:
        """Call tool with exponential backoff retry and circuit breaker"""

        circuit_breaker = self.circuit_breakers.get(tool_name)
        if not circuit_breaker:
            # Create circuit breaker for unknown tools
            circuit_breaker = CircuitBreaker()
            self.circuit_breakers[tool_name] = circuit_breaker

        for attempt in range(max_retries + 1):
            try:
                # Use circuit breaker to protect the call
                result = await circuit_breaker.call(self._make_tool_call, tool_name, argu
                return {
                    'success': True,
                    'data': result,
                    'attempt': attempt + 1,
                    'circuit_state': circuit_breaker.state.value
                }

            except Exception as e:
                logger.warning(f"Tool call attempt {attempt + 1} failed for {tool_name}:

                if attempt < max_retries:
                    # Exponential backoff with jitter
                    delay = (2 ** attempt) + random.uniform(0, 1)
                    await asyncio.sleep(delay)
                else:
                    return {
                        'success': False,
                        'error': str(e),
                        'attempts_made': attempt + 1,
                        'circuit_state': circuit_breaker.state.value
                    }

    async def _make_tool_call(self, tool_name: str, arguments: dict) -> dict:
```

```
        """Make actual tool call (wrapped by circuit breaker)"""

        if not self.authenticated:
            raise Exception("Not authenticated")

        payload = {
            "jsonrpc": "2.0",
            "id": 1,
            "method": "tools/call",
            "params": {
                "name": tool_name,
                "arguments": arguments
            }
        }

        headers = {
            "Content-Type": "application/json",
            "Mcp-Session-Id": self.session_id
        }

        async with aiohttp.ClientSession(timeout=aiohttp.ClientTimeout(total=30)) as sess
            async with session.post(f"{self.base_url}/mcp/stream", headers=headers, json=
                if response.status != 200:
                    raise Exception(f"HTTP {response.status}: {await response.text()}")

                result = await response.json()
                content = result.get("result", {}).get("content", [{}])[^0]
                return json.loads(content.get("text", "{}"))

    def get_circuit_breaker_status(self) -> dict:
        """Get status of all circuit breakers"""
        return {
            tool_name: {
                'state': cb.state.value,
                'failure_count': cb.failure_count,
                'last_failure_time': cb.last_failure_time
            }
            for tool_name, cb in self.circuit_breakers.items()
        }
```

9. Performance Optimization and Scaling

Caching and Data Management Strategies

Financial agents often need to access the same data multiple times during a conversation. Implementing smart caching can significantly improve performance and reduce API calls to the Fi MCP server:

```
import time
import hashlib
import pickle
from typing import Dict, Any, Optional
from dataclasses import dataclass, field
```

```
from datetime import datetime, timedelta

@dataclass
class CacheEntry:
    """Cache entry with metadata"""
    data: Any
    timestamp: float
    access_count: int = 0
    last_accessed: float = field(default_factory=time.time)
    ttl: float = 300  # Default 5 minutes TTL

class IntelligentFinancialCache:
    """Intelligent caching system for financial data"""

    def __init__(self, max_size: int = 1000, default_ttl: float = 300):
        self.cache: Dict[str, CacheEntry] = {}
        self.max_size = max_size
        self.default_ttl = default_ttl
        self.cache_stats = {
            'hits': 0,
            'misses': 0,
            'evictions': 0,
            'size': 0
        }

        # Different TTL for different data types
        self.data_type_ttl = {
            'fetch_net_worth': 300,        # 5 minutes - relatively static
            'fetch_credit_report': 3600,   # 1 hour - changes monthly
            'fetch_epf_details': 1800,     # 30 minutes - updates quarterly
            'fetch_mf_transactions': 1800, # 30 minutes - daily NAV updates
            'fetch_bank_transactions': 600, # 10 minutes - frequent updates
            'fetch_stock_transactions': 900 # 15 minutes - intraday changes
        }

    def _generate_cache_key(self, tool_name: str, phone_number: str, arguments: dict = Nc
        """Generate cache key for tool call"""
        key_data = f"{tool_name}:{phone_number}:{str(sorted((arguments or {}).items()))}'
        return hashlib.md5(key_data.encode()).hexdigest()

    def get(self, tool_name: str, phone_number: str, arguments: dict = None) -> Optional[
        """Get data from cache if available and valid"""

        cache_key = self._generate_cache_key(tool_name, phone_number, arguments)

        if cache_key not in self.cache:
            self.cache_stats['misses'] += 1
            return None

        entry = self.cache[cache_key]
        current_time = time.time()

        # Check if entry has expired
        ttl = self.data_type_ttl.get(tool_name, self.default_ttl)
        if current_time - entry.timestamp > ttl:
            del self.cache[cache_key]
```

```
            self.cache_stats['misses'] += 1
            self.cache_stats['size'] = len(self.cache)
            return None

        # Update access statistics
        entry.access_count += 1
        entry.last_accessed = current_time
        self.cache_stats['hits'] += 1

        return entry.data

    def put(self, tool_name: str, phone_number: str, data: Any, arguments: dict = None):
        """Store data in cache with intelligent eviction"""

        cache_key = self._generate_cache_key(tool_name, phone_number, arguments)

        # Check if we need to evict entries
        if len(self.cache) >= self.max_size:
            self._evict_entries()

        ttl = self.data_type_ttl.get(tool_name, self.default_ttl)

        self.cache[cache_key] = CacheEntry(
            data=data,
            timestamp=time.time(),
            ttl=ttl
        )

        self.cache_stats['size'] = len(self.cache)

    def _evict_entries(self):
        """Evict cache entries using LFU (Least Frequently Used) strategy"""

        current_time = time.time()

        # First, remove expired entries
        expired_keys = []
        for key, entry in self.cache.items():
            if current_time - entry.timestamp > entry.ttl:
                expired_keys.append(key)

        for key in expired_keys:
            del self.cache[key]
            self.cache_stats['evictions'] += 1

        # If still over limit, evict least frequently used
        if len(self.cache) >= self.max_size:
            entries_by_usage = sorted(
                self.cache.items(),
                key=lambda x: (x[^1].access_count, x[^1].last_accessed)
            )

            # Evict 10% of entries or minimum 1
            evict_count = max(1, len(self.cache) // 10)

            for key, _ in entries_by_usage[:evict_count]:
```

```
                del self.cache[key]
                self.cache_stats['evictions'] += 1

        self.cache_stats['size'] = len(self.cache)

    def get_stats(self) -> dict:
        """Get cache performance statistics"""
        total_requests = self.cache_stats['hits'] + self.cache_stats['misses']
        hit_rate = (self.cache_stats['hits'] / total_requests * 100) if total_requests >

        return {
            **self.cache_stats,
            'hit_rate_percentage': round(hit_rate, 2),
            'total_requests': total_requests
        }

    def clear_user_cache(self, phone_number: str):
        """Clear all cache entries for a specific user"""
        keys_to_remove = []

        for key in self.cache.keys():
            # Extract phone number from cache key (simplified)
            if phone_number in key:
                keys_to_remove.append(key)

        for key in keys_to_remove:
            del self.cache[key]

        self.cache_stats['size'] = len(self.cache)

class PerformanceOptimizedMCPClient:
    """High-performance Fi MCP client with caching and batch operations"""

    def __init__(self, base_url: str = "http://localhost:8080"):
        self.base_url = base_url
        self.session_id = None
        self.authenticated = False
        self.cache = IntelligentFinancialCache()

        # Connection pooling
        self.connector = aiohttp.TCPConnector(
            limit=100,  # Total connection pool size
            limit_per_host=30,  # Connections per host
            keepalive_timeout=30,
            enable_cleanup_closed=True
        )

        # Performance metrics
        self.performance_metrics = {
            'total_requests': 0,
            'cache_hits': 0,
            'api_calls': 0,
            'total_response_time': 0,
            'batch_operations': 0
        }
```

```
    async def batch_fetch_financial_data(self, phone_number: str, tool_names: list) -> di
        """Fetch multiple financial data sources concurrently"""

        start_time = time.time()
        self.performance_metrics['batch_operations'] += 1

        # Check cache first
        cached_results = {}
        remaining_tools = []

        for tool_name in tool_names:
            cached_data = self.cache.get(tool_name, phone_number)
            if cached_data is not None:
                cached_results[tool_name] = cached_data
                self.performance_metrics['cache_hits'] += 1
            else:
                remaining_tools.append(tool_name)

        # Fetch remaining data concurrently
        api_results = {}
        if remaining_tools:
            async with aiohttp.ClientSession(connector=self.connector) as session:
                tasks = []

                for tool_name in remaining_tools:
                    task = self._fetch_single_tool(session, tool_name)
                    tasks.append((tool_name, task))

                # Execute all API calls concurrently
                for tool_name, task in tasks:
                    try:
                        result = await task
                        api_results[tool_name] = result

                        # Cache the result
                        self.cache.put(tool_name, phone_number, result)
                        self.performance_metrics['api_calls'] += 1

                    except Exception as e:
                        api_results[tool_name] = {'error': str(e)}

        # Combine cached and API results
        all_results = {**cached_results, **api_results}

        # Update performance metrics
        total_time = time.time() - start_time
        self.performance_metrics['total_response_time'] += total_time
        self.performance_metrics['total_requests'] += len(tool_names)

        return {
            'data': all_results,
            'performance': {
                'total_time_ms': int(total_time * 1000),
                'cache_hits': len(cached_results),
                'api_calls': len(api_results),
                'tools_requested': len(tool_names)
```

```
            },
            'cache_stats': self.cache.get_stats()
        }

    async def _fetch_single_tool(self, session: aiohttp.ClientSession, tool_name: str) ->
        """Fetch data from a single tool"""

        payload = {
            "jsonrpc": "2.0",
            "id": 1,
            "method": "tools/call",
            "params": {
                "name": tool_name,
                "arguments": {}
            }
        }

        headers = {
            "Content-Type": "application/json",
            "Mcp-Session-Id": self.session_id
        }

        async with session.post(f"{self.base_url}/mcp/stream", headers=headers, json=pay]
            if response.status != 200:
                raise Exception(f"HTTP {response.status}")

            result = await response.json()
            content = result.get("result", {}).get("content", [{}])[^0]
            return json.loads(content.get("text", "{}"))

    def get_performance_report(self) -> dict:
        """Generate comprehensive performance report"""

        total_requests = self.performance_metrics['total_requests']
        avg_response_time = (
            self.performance_metrics['total_response_time'] / self.performance_metrics['b
            if self.performance_metrics['batch_operations'] > 0 else 0
        )

        cache_hit_rate = (
            self.performance_metrics['cache_hits'] / total_requests * 100
            if total_requests > 0 else 0
        )

        return {
            'performance_metrics': self.performance_metrics,
            'average_response_time_ms': int(avg_response_time * 1000),
            'cache_hit_rate_percentage': round(cache_hit_rate, 2),
            'cache_statistics': self.cache.get_stats(),
            'api_efficiency': {
                'requests_per_batch': total_requests / max(1, self.performance_metrics['b
                'cache_effectiveness': cache_hit_rate
            }
        }
```

Asynchronous Processing and Queue Management

For production financial agents that handle multiple users concurrently, implementing proper queue management and asynchronous processing is crucial:

```
import asyncio
from asyncio import Queue, Semaphore
from typing import Callable, Any
import logging
from dataclasses import dataclass
from enum import Enum
import uuid

class TaskPriority(Enum):
    LOW = 1
    NORMAL = 2
    HIGH = 3
    CRITICAL = 4

@dataclass
class FinancialTask:
    """Financial processing task with metadata"""
    task_id: str
    user_id: str
    task_type: str
    payload: dict
    priority: TaskPriority
    callback: Callable
    created_at: float
    timeout: float = 30.0

class FinancialTaskProcessor:
    """Asynchronous task processor for financial operations"""

    def __init__(self, max_concurrent_tasks: int = 10, max_queue_size: int = 1000):
        self.max_concurrent_tasks = max_concurrent_tasks
        self.max_queue_size = max_queue_size

        # Priority queues for different task types
        self.task_queues = {
            TaskPriority.CRITICAL: Queue(),
            TaskPriority.HIGH: Queue(),
            TaskPriority.NORMAL: Queue(),
            TaskPriority.LOW: Queue()
        }

        # Semaphore to limit concurrent processing
        self.semaphore = Semaphore(max_concurrent_tasks)

        # Task tracking
        self.active_tasks = {}
        self.completed_tasks = {}
        self.failed_tasks = {}

        # Performance metrics
```

```
        self.metrics = {
            'tasks_processed': 0,
            'tasks_failed': 0,
            'average_processing_time': 0,
            'queue_sizes': {}
        }

        # Worker tasks
        self.workers = []
        self.running = False

    async def start(self):
        """Start the task processor workers"""
        self.running = True

        # Start worker tasks for each priority level
        for priority in TaskPriority:
            worker = asyncio.create_task(self._worker(priority))
            self.workers.append(worker)

        logger.info(f"Started {len(self.workers)} worker tasks")

    async def stop(self):
        """Stop the task processor"""
        self.running = False

        # Cancel all workers
        for worker in self.workers:
            worker.cancel()

        # Wait for workers to finish
        await asyncio.gather(*self.workers, return_exceptions=True)
        logger.info("Task processor stopped")

    async def submit_task(self, task: FinancialTask) -> str:
        """Submit a task for processing"""

        # Check queue size limits
        total_queued = sum(queue.qsize() for queue in self.task_queues.values())
        if total_queued >= self.max_queue_size:
            raise Exception("Task queue is full")

        # Add to appropriate priority queue
        await self.task_queues[task.priority].put(task)
        self.active_tasks[task.task_id] = task

        logger.info(f"Task {task.task_id} submitted with priority {task.priority.name}")
        return task.task_id

    async def _worker(self, priority: TaskPriority):
        """Worker coroutine for processing tasks of specific priority"""

        queue = self.task_queues[priority]

        while self.running:
            try:
```

```
                # Get task from queue with timeout
                task = await asyncio.wait_for(queue.get(), timeout=1.0)

                # Process task with semaphore protection
                async with self.semaphore:
                    await self._process_task(task)

            except asyncio.TimeoutError:
                # No tasks in queue, continue
                continue
            except Exception as e:
                logger.error(f"Worker error for priority {priority.name}: {e}")

    async def _process_task(self, task: FinancialTask):
        """Process an individual task"""

        start_time = time.time()

        try:
            # Execute task with timeout
            result = await asyncio.wait_for(
                task.callback(task.payload),
                timeout=task.timeout
            )

            # Record successful completion
            processing_time = time.time() - start_time
            self.completed_tasks[task.task_id] = {
                'task': task,
                'result': result,
                'processing_time': processing_time,
                'completed_at': time.time()
            }

            # Update metrics
            self._update_metrics(processing_time, success=True)

            logger.info(f"Task {task.task_id} completed in {processing_time:.2f}s")

        except Exception as e:
            # Record failure
            processing_time = time.time() - start_time
            self.failed_tasks[task.task_id] = {
                'task': task,
                'error': str(e),
                'processing_time': processing_time,
                'failed_at': time.time()
            }

            # Update metrics
            self._update_metrics(processing_time, success=False)

            logger.error(f"Task {task.task_id} failed after {processing_time:.2f}s: {e}")

        finally:
            # Remove from active tasks
```

```
            if task.task_id in self.active_tasks:
                del self.active_tasks[task.task_id]

    def _update_metrics(self, processing_time: float, success: bool):
        """Update performance metrics"""

        if success:
            self.metrics['tasks_processed'] += 1
        else:
            self.metrics['tasks_failed'] += 1

        # Update average processing time
        total_tasks = self.metrics['tasks_processed'] + self.metrics['tasks_failed']
        current_avg = self.metrics['average_processing_time']

        self.metrics['average_processing_time'] = (
            (current_avg * (total_tasks - 1) + processing_time) / total_tasks
        )

        # Update queue sizes
        self.metrics['queue_sizes'] = {
            priority.name: queue.qsize()
            for priority, queue in self.task_queues.items()
        }

    def get_status(self) -> dict:
        """Get current processor status"""

        total_tasks = self.metrics['tasks_processed'] + self.metrics['tasks_failed']
        success_rate = (
            self.metrics['tasks_processed'] / total_tasks * 100
            if total_tasks > 0 else 0
        )

        return {
            'running': self.running,
            'active_tasks': len(self.active_tasks),
            'queue_sizes': self.metrics['queue_sizes'],
            'total_queued': sum(self.metrics['queue_sizes'].values()),
            'performance': {
                'tasks_processed': self.metrics['tasks_processed'],
                'tasks_failed': self.metrics['tasks_failed'],
                'success_rate_percentage': round(success_rate, 2),
                'average_processing_time_ms': int(self.metrics['average_processing_time']
            }
        }

class ScalableFinancialAgentSystem:
    """Scalable financial agent system with task queue management"""

    def __init__(self, mcp_client, max_concurrent_users: int = 50):
        self.mcp_client = mcp_client
        self.task_processor = FinancialTaskProcessor(max_concurrent_tasks=max_concurrent_

        # User session management
        self.user_sessions = {}
```

```
        self.session_locks = {}

    async def start(self):
        """Start the scalable agent system"""
        await self.task_processor.start()
        logger.info("Scalable financial agent system started")

    async def stop(self):
        """Stop the agent system"""
        await self.task_processor.stop()
        logger.info("Scalable financial agent system stopped")

    async def process_user_request(self, user_id: str, request_type: str,
                                   request_data: dict, priority: TaskPriority = TaskPriorit
        """Process user request asynchronously"""

        task_id = str(uuid.uuid4())

        # Create task
        task = FinancialTask(
            task_id=task_id,
            user_id=user_id,
            task_type=request_type,
            payload={
                'user_id': user_id,
                'request_type': request_type,
                'request_data': request_data
            },
            priority=priority,
            callback=self._process_financial_request,
            created_at=time.time()
        )

        # Submit to task processor
        await self.task_processor.submit_task(task)
        return task_id

    async def _process_financial_request(self, payload: dict) -> dict:
        """Process individual financial request"""

        user_id = payload['user_id']
        request_type = payload['request_type']
        request_data = payload['request_data']

        # Ensure user session exists and is locked during processing
        async with self._get_user_lock(user_id):

            # Initialize user session if needed
            if user_id not in self.user_sessions:
                await self._initialize_user_session(user_id)

            # Route to appropriate handler
            if request_type == 'comprehensive_analysis':
                return await self._handle_comprehensive_analysis(user_id, request_data)
            elif request_type == 'goal_planning':
                return await self._handle_goal_planning(user_id, request_data)
```

```
            elif request_type == 'portfolio_analysis':
                return await self._handle_portfolio_analysis(user_id, request_data)
            elif request_type == 'debt_optimization':
                return await self._handle_debt_optimization(user_id, request_data)
            else:
                raise ValueError(f"Unknown request type: {request_type}")

    async def _get_user_lock(self, user_id: str) -> asyncio.Lock:
        """Get or create lock for user session"""
        if user_id not in self.session_locks:
            self.session_locks[user_id] = asyncio.Lock()
        return self.session_locks[user_id]

    async def _initialize_user_session(self, user_id: str):
        """Initialize user session with authentication"""

        # Authenticate with Fi MCP
        auth_success = await self.mcp_client.authenticate_with_retry(user_id)
        if not auth_success:
            raise Exception(f"Authentication failed for user {user_id}")

        # Fetch initial financial data
        financial_data = await self.mcp_client.batch_fetch_financial_data(
            user_id,
            ['fetch_net_worth', 'fetch_credit_report', 'fetch_epf_details']
        )

        self.user_sessions[user_id] = {
            'authenticated': True,
            'financial_data': financial_data,
            'last_updated': time.time(),
            'request_count': 0
        }

    async def _handle_comprehensive_analysis(self, user_id: str, request_data: dict) -> d
        """Handle comprehensive financial analysis request"""

        session = self.user_sessions[user_id]
        session['request_count'] += 1

        # Get fresh data if needed
        if time.time() - session['last_updated'] > 300:  # 5 minutes
            session['financial_data'] = await self.mcp_client.batch_fetch_financial_data(
                user_id,
                ['fetch_net_worth', 'fetch_credit_report', 'fetch_mf_transactions']
            )
            session['last_updated'] = time.time()

        # Perform analysis
        processor = RobustFinancialDataProcessor()

        net_worth_analysis = processor.process_net_worth_data(
            session['financial_data']['data'].get('fetch_net_worth', {})
        )

        credit_analysis = processor.process_credit_report_data(
```

```
            session['financial_data']['data'].get('fetch_credit_report', {})
        )

        return {
            'user_id': user_id,
            'analysis_type': 'comprehensive',
            'net_worth_analysis': net_worth_analysis,
            'credit_analysis': credit_analysis,
            'recommendations': self._generate_comprehensive_recommendations(
                net_worth_analysis, credit_analysis
            ),
            'timestamp': time.time()
        }

    def get_system_status(self) -> dict:
        """Get comprehensive system status"""

        return {
            'task_processor': self.task_processor.get_status(),
            'active_user_sessions': len(self.user_sessions),
            'mcp_client': self.mcp_client.get_performance_report() if hasattr(self.mcp_cl
            'system_health': 'healthy' if self.task_processor.running else 'stopped'
        }
```

10. Security and Compliance Guidelines

Data Privacy and Security Implementation

When building financial agents with Fi MCP, security and data privacy are paramount. Here's a comprehensive implementation of security best practices:

```
import hashlib
import hmac
import secrets
import base64
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import logging
from datetime import datetime, timedelta
from typing import Dict, Any, Optional
import re

class FinancialDataSecurity:
    """Comprehensive security implementation for financial data"""

    def __init__(self, master_key: Optional[str] = None):
        self.master_key = master_key or self._generate_master_key()
        self.cipher_suite = self._setup_encryption()
        self.audit_logger = self._setup_audit_logging()

        # PII patterns for detection and redaction
        self.pii_patterns = {
            'phone_number': re.compile(r'\b\d{10}\b'),
```

```
            'account_number': re.compile(r'\b\d{8,16}\b'),
            'pan_number': re.compile(r'\b[A-Z]{5}\d{4}[A-Z]\b'),
            'aadhaar_number': re.compile(r'\b\d{4}\s?\d{4}\s?\d{4}\b'),
            'credit_card': re.compile(r'\b\d{4}[\s-]?\d{4}[\s-]?\d{4}[\s-]?\d{4}\b')
        }

    def _generate_master_key(self) -> str:
        """Generate a secure master key"""
        return base64.urlsafe_b64encode(secrets.token_bytes(32)).decode()

    def _setup_encryption(self) -> Fernet:
        """Setup encryption using Fernet symmetric encryption"""
        key_bytes = base64.urlsafe_b64decode(self.master_key.encode())
        return Fernet(base64.urlsafe_b64encode(key_bytes[:32]))

    def _setup_audit_logging(self) -> logging.Logger:
        """Setup audit logging for security events"""
        audit_logger = logging.getLogger('financial_security_audit')
        audit_logger.setLevel(logging.INFO)

        # Create file handler for audit logs
        handler = logging.FileHandler('financial_audit.log')
        formatter = logging.Formatter(
            '%(asctime)s - %(levelname)s - %(message)s'
        )
        handler.setFormatter(formatter)
        audit_logger.addHandler(handler)

        return audit_logger

    def encrypt_sensitive_data(self, data: str) -> str:
        """Encrypt sensitive data"""
        try:
            encrypted_data = self.cipher_suite.encrypt(data.encode())
            return base64.urlsafe_b64encode(encrypted_data).decode()
        except Exception as e:
            self.audit_logger.error(f"Encryption failed: {str(e)}")
            raise

    def decrypt_sensitive_data(self, encrypted_data: str) -> str:
        """Decrypt sensitive data"""
        try:
            decoded_data = base64.urlsafe_b64decode(encrypted_data.encode())
            decrypted_data = self.cipher_suite.decrypt(decoded_data)
            return decrypted_data.decode()
        except Exception as e:
            self.audit_logger.error(f"Decryption failed: {str(e)}")
            raise

    def redact_pii(self, text: str) -> str:
        """Redact personally identifiable information from text"""
        redacted_text = text

        for pii_type, pattern in self.pii_patterns.items():
            if pii_type == 'phone_number':
                redacted_text = pattern.sub('XXXXXXXXXX', redacted_text)
```

```
            elif pii_type == 'account_number':
                redacted_text = pattern.sub('XXXXXXXX', redacted_text)
            elif pii_type == 'pan_number':
                redacted_text = pattern.sub('XXXXX####X', redacted_text)
            elif pii_type == 'aadhaar_number':
                redacted_text = pattern.sub('XXXX XXXX XXXX', redacted_text)
            elif pii_type == 'credit_card':
                redacted_text = pattern.sub('XXXX XXXX XXXX XXXX', redacted_text)

        return redacted_text

    def log_data_access(self, user_id: str, data_type: str, action: str,
                        ip_address: str = None, user_agent: str = None):
        """Log data access for audit purposes"""

        log_entry = {
            'timestamp': datetime.utcnow().isoformat(),
            'user_id': self.hash_identifier(user_id),
            'data_type': data_type,
            'action': action,
            'ip_address': ip_address,
            'user_agent': user_agent
        }

        self.audit_logger.info(f"DATA_ACCESS: {log_entry}")

    def hash_identifier(self, identifier: str) -> str:
        """Create one-way hash of identifier for logging"""
        return hashlib.sha256(identifier.encode()).hexdigest()[:16]

    def validate_data_integrity(self, data: dict, expected_checksum: str) -> bool:
        """Validate data integrity using checksum"""

        # Create deterministic string representation
        data_string = json.dumps(data, sort_keys=True, separators=(',', ':'))
        calculated_checksum = hashlib.sha256(data_string.encode()).hexdigest()

        return hmac.compare_digest(calculated_checksum, expected_checksum)

    def create_data_checksum(self, data: dict) -> str:
        """Create integrity checksum for data"""
        data_string = json.dumps(data, sort_keys=True, separators=(',', ':'))
        return hashlib.sha256(data_string.encode()).hexdigest()

class SecureFinancialAgent:
    """Security-focused financial agent implementation"""

    def __init__(self, mcp_client):
        self.mcp_client = mcp_client
        self.security = FinancialDataSecurity()
        self.session_tokens = {}
        self.rate_limiters = {}
        self.setup_secure_agent()

    def setup_secure_agent(self):
        """Setup agent with security controls"""
```

```
        secure_tools = [
            self.create_secure_data_fetcher(),
            self.create_secure_analyzer(),
            self.create_audit_trail_manager()
        ]

        self.agent = LlmAgent(
            model="gemini-2.0-flash",
            name="secure_financial_advisor",
            instruction="""
            You are a secure financial advisor that handles sensitive financial informati

            SECURITY PROTOCOLS:
            1. Never log or display sensitive financial data in plain text
            2. Always use anonymized/masked data in examples
            3. Validate all user inputs before processing
            4. Report suspicious activity or data inconsistencies
            5. Provide security recommendations as part of financial advice

            DATA HANDLING:
            - Treat all financial data as confidential
            - Use secure references instead of actual account numbers
            - Redact sensitive information in responses
            - Implement principle of least privilege for data access

            USER COMMUNICATION:
            - Educate users on financial security best practices
            - Warn about common financial scams and threats
            - Recommend secure financial habits
            """,
            tools=secure_tools,
            before_agent_callback=self.security_pre_check,
            after_agent_callback=self.security_post_check
        )

    def create_secure_data_fetcher(self):
        """Create data fetcher with security controls"""

        async def secure_fetch_financial_data(user_id: str, data_types: list,
                                              session_token: str, tool_context: ToolContext
            """
            Securely fetch financial data with authentication and logging
            """

            # Validate session token
            if not self.validate_session_token(user_id, session_token):
                self.security.log_data_access(user_id, 'ALL', 'UNAUTHORIZED_ACCESS')
                return {'error': 'Invalid session token', 'access_denied': True}

            # Rate limiting check
            if self.is_rate_limited(user_id):
                self.security.log_data_access(user_id, 'ALL', 'RATE_LIMITED')
                return {'error': 'Rate limit exceeded', 'retry_after': 60}

            # Log data access
```

```
            self.security.log_data_access(user_id, ','.join(data_types), 'FETCH_REQUESTED

            # Fetch data through secure client
            try:
                raw_data = await self.mcp_client.batch_fetch_financial_data(user_id, data

                # Create secure data package
                secure_data = self.create_secure_data_package(raw_data, user_id)

                # Store in secure session state
                tool_context.state[f"secure_data_{self.security.hash_identifier(user_id)}

                self.security.log_data_access(user_id, ','.join(data_types), 'FETCH_COMPL

                return {
                    'status': 'success',
                    'data_types_fetched': data_types,
                    'data_quality': secure_data.get('quality_assessment', 'unknown'),
                    'security_level': 'encrypted'
                }

            except Exception as e:
                self.security.log_data_access(user_id, ','.join(data_types), 'FETCH_FAILE
                return {'error': 'Data fetch failed', 'details': 'Check audit logs'}

        return secure_fetch_financial_data

    def create_secure_data_package(self, raw_data: dict, user_id: str) -> dict:
        """Create secure, encrypted data package"""

        # Process and encrypt sensitive data
        secure_package = {
            'user_hash': self.security.hash_identifier(user_id),
            'encrypted_data': {},
            'data_checksums': {},
            'created_at': time.time(),
            'quality_assessment': 'good'
        }

        for data_type, data in raw_data.get('data', {}).items():
            if data and not data.get('error'):
                # Create checksum for integrity
                checksum = self.security.create_data_checksum(data)
                secure_package['data_checksums'][data_type] = checksum

                # Encrypt sensitive data
                data_json
```


[^1]: https://developers.cloudflare.com/agents/model-context-protocol/
[^2]: https://docs.anthropic.com/en/docs/mcp
[^3]: https://github.com/MicrosoftDocs/mcp
[^4]: https://github.com/topics/mcp?l=html&o=desc&s=updated
[^5]: https://fi.money/features/getting-started-with-fi-mcp
[^6]: Working-Reference-Agent.py

[^7]: https://github.com/epiFi/fi-mcp-dev
[^8]: https://www.getambassador.io/blog/model-context-protocol-mcp-connecting-llms-to-api
[^9]: https://gitmcp.io
[^10]: https://www.youtube.com/watch?v=3F4re1zpZP4
[^11]: https://github.com/epiFi
[^12]: https://docs.spring.io/spring-ai/reference/api/mcp/mcp-overview.html
[^13]: https://github.com/github/github-mcp-server
[^14]: https://fi.money/guides/personal-loans/here-is-the-list-of-documents-required-for-
[^15]: https://treblle.com/blog/model-context-protocol-guide
[^16]: https://github.com/epiFi/fi-mcp-dev/pulls
[^17]: https://fi.money/FAQs/wealth-analyzer-(fi-mcp)/for-coders/what-kind-of-financial-d
[^18]: https://modelcontextprotocol.io/specification/2025-06-18
[^19]: https://fi.money/blog/tnc
[^20]: https://modelcontextprotocol.io
[^21]: https://fi.money/features/using-fi-mcp-for-money-management
[^22]: https://github.com/epiFi/mcp-docs
[^23]: https://github.com/epiFi/mcp-docs/blob/master/sample_responses/fetch_net_worth.jso
[^24]: https://github.com/epiFi/mcp-docs/blob/master/sample_responses/fetch_credit_report
[^25]: https://github.com/epiFi/mcp-docs/blob/master/sample_responses/fetch_epf_details.j
[^26]: https://github.com/epiFi/mcp-docs/blob/master/sample_responses/fetch_mf_transactio