YieldMax Data Architecture & Infrastructure

Executive Summary

This document outlines a production-grade data infrastructure designed to power YieldMax's cross-chain yield optimization with >99.5% accuracy, <30 second latency, and >99.9% uptime. The architecture combines on-chain data aggregation, Chainlink Data Streams, and sophisticated analytics to enable profitable rebalancing decisions.

1. System Architecture Overview

```mermaid
```

```
graph TB
    subgraph "Data Sources"
        A1[Chainlink Data Streams]
        A2[Protocol RPCs]
        A3[Graph Protocol]
        A4[Direct APIs]
        A5[DEX Data]
    end

    subgraph "Data Ingestion Layer"
        B1[Real-time Collectors]
        B2[Historical Scrapers]
        B3[WebSocket Listeners]
        B4[Data Validators]
    end

    subgraph "Processing Layer"
        C1[Stream Processor]
        C2[Yield Calculator]
        C3[Risk Analyzer]
        C4[Gas Optimizer]
        C5[Anomaly Detector]
    end

    subgraph "Storage Layer"
        D1[(Time Series DB)]
        D2[(Cache Layer)]
        D3[(Historical Archive)]
        D4[(Real-time Store)]
    end

    subgraph "Analytics Layer"
        E1[Trend Predictor]
```

```
    E2[Liquidity Analyzer]
    E3[Risk Scorer]
    E4[Optimization Engine]
end

subgraph "API & Delivery"
    F1[REST API]
    F2[WebSocket Server]
    F3[Chainlink Functions]
    F4[Dashboard]
end

A1 --> B1
A2 --> B1
A3 --> B2
A4 --> B3
A5 --> B1

B1 --> C1
B2 --> C2
B3 --> C1
B4 --> C1

C1 --> D1
C1 --> D2
C2 --> D1
C3 --> D4
C4 --> D2
C5 --> D4

D1 --> E1
D2 --> E2
D3 --> E1
```

```
D4 --> E3

E1 --> F1
E2 --> F2
E3 --> F3
E4 --> F4
```

2. Data Collection Infrastructure

2.1 Real-time Yield Collectors

```
python
```

```python
import asyncio
import aiohttp
from web3 import Web3
from typing import Dict, List, Tuple
import time
from dataclasses import dataclass
from decimal import Decimal

@dataclass
class YieldDataPoint:
    protocol: str
    chain: str
    asset: str
    apy: Decimal
    tvl: Decimal
    utilization: Decimal
    timestamp: int
    block_number: int
    data_source: str

class RealTimeYieldCollector:
    """
    Production-grade yield data collector with redundancy and validation
    """

    def __init__(self):
        self.web3_endpoints = {
            'ethereum': [
                'https://eth.llamarpc.com',
                'https://rpc.ankr.com/eth',
                'https://eth.drpc.org'
            ],
            'arbitrum': [
```

```
            'https://arb1.arbitrum.io/rpc',
            'https://rpc.ankr.com/arbitrum',
            'https://arbitrum.llamarpc.com'
        ],
        'polygon': [
            'https://polygon-rpc.com',
            'https://rpc.ankr.com/polygon',
            'https://polygon.llamarpc.com'
        ],
        'optimism': [
            'https://mainnet.optimism.io',
            'https://rpc.ankr.com/optimism',
            'https://optimism.llamarpc.com'
        ]
    }

    self.protocol_configs = {
        'aave_v3': {
            'pool_data_provider': {
                'ethereum': '0x7B4EB56E7CD4b454BA8ff71E4518426369a138a3',
                'arbitrum': '0x69FA688f1Dc47d4B5d8029D5a35FB7a548310654',
                'polygon': '0x69FA688f1Dc47d4B5d8029D5a35FB7a548310654',
                'optimism': '0x69FA688f1Dc47d4B5d8029D5a35FB7a548310654'
            },
            'decimals_offset': 27
        },
        'compound_v3': {
            'comet': {
                'ethereum': '0xc3d688B66703497DAA19211EEdff47f25384cdc3',
                'arbitrum': '0xA5EDBDD9646f8dFF606d7448e414884C7d905dCA',
                'polygon': '0xF25212E676D1F7F89Cd72fFEe66158f541246445'
            }
        },
```

```
            'morpho': {
                'lens': {
                    'ethereum': '0x930f1b46e1D081Ec1524efD95752bE3eCe51EF67'
                }
            },
            'spark': {
                'pool_data_provider': {
                    'ethereum': '0xFc21d6d146E6086B8359705C8b28512a983db0cb'
                }
            }
        }

        self.data_quality_thresholds = {
            'max_apy': Decimal('50'),  # 50% max believable APY
            'min_apy': Decimal('0'),
            'max_utilization': Decimal('100'),
            'stale_data_seconds': 300  # 5 minutes
        }

    async def collect_all_yields(self) -> List[YieldDataPoint]:
        """
        Collect yield data from all protocols across all chains
        """
        tasks = []

        for chain in self.web3_endpoints.keys():
            for protocol in self.protocol_configs.keys():
                tasks.append(
                    self.collect_protocol_yield(protocol, chain)
                )

        results = await asyncio.gather(*tasks, return_exceptions=True)
```

```python
    # Filter out errors and validate data
    valid_data = []
    for result in results:
        if isinstance(result, YieldDataPoint):
            if self.validate_data_point(result):
                valid_data.append(result)
            else:
                await self.log_invalid_data(result)
        else:
            await self.log_collection_error(result)

    return valid_data

async def collect_protocol_yield(
    self,
    protocol: str,
    chain: str
) -> YieldDataPoint:
    """
    Collect yield data for specific protocol on specific chain
    """
    if protocol == 'aave_v3':
        return await self.collect_aave_yield(chain)
    elif protocol == 'compound_v3':
        return await self.collect_compound_yield(chain)
    elif protocol == 'morpho':
        return await self.collect_morpho_yield(chain)
    elif protocol == 'spark':
        return await self.collect_spark_yield(chain)

async def collect_aave_yield(self, chain: str) -> YieldDataPoint:
    """
    Collect Aave V3 yield data with redundancy
```

```
        """
        web3 = await self.get_web3_connection(chain)

        pool_data_provider = self.protocol_configs['aave_v3']['pool_data_provider'][chain

        # ABI for getReserveData
        abi = [{
            "inputs": [{"name": "asset", "type": "address"}],
            "name": "getReserveData",
            "outputs": [
                {"name": "", "type": "uint256"},  # unbacked
                {"name": "", "type": "uint256"},  # accruedToTreasuryScaled
                {"name": "totalAToken", "type": "uint256"},
                {"name": "totalStableDebt", "type": "uint256"},
                {"name": "totalVariableDebt", "type": "uint256"},
                {"name": "liquidityRate", "type": "uint256"},
                {"name": "variableBorrowRate", "type": "uint256"},
                {"name": "stableBorrowRate", "type": "uint256"},
                {"name": "averageStableBorrowRate", "type": "uint256"},
                {"name": "liquidityIndex", "type": "uint256"},
                {"name": "variableBorrowIndex", "type": "uint256"},
                {"name": "lastUpdateTimestamp", "type": "uint40"}
            ],
            "type": "function"
        }]

        contract = web3.eth.contract(
            address=Web3.to_checksum_address(pool_data_provider),
            abi=abi
        )

        # Get USDC data
        usdc_address = self.get_usdc_address(chain)
```

```
        reserve_data = await contract.functions.getReserveData(usdc_address).call()

        # Calculate APY from ray (27 decimals)
        liquidity_rate = Decimal(reserve_data[5])
        seconds_per_year = Decimal(31536000)
        ray = Decimal(10**27)

        # Convert from ray to APY
        apy = ((liquidity_rate / ray) * seconds_per_year) * 100

        # Calculate utilization
        total_liquidity = Decimal(reserve_data[2])
        total_debt = Decimal(reserve_data[3]) + Decimal(reserve_data[4])
        utilization = (total_debt / total_liquidity * 100) if total_liquidity > 0 else De

        return YieldDataPoint(
            protocol='aave_v3',
            chain=chain,
            asset='USDC',
            apy=apy,
            tvl=total_liquidity / Decimal(10**6),  # Convert to USD
            utilization=utilization,
            timestamp=int(time.time()),
            block_number=await web3.eth.block_number,
            data_source='on_chain'
        )

    async def get_web3_connection(self, chain: str) -> Web3:
        """
        Get Web3 connection with fallback endpoints
        """
        for endpoint in self.web3_endpoints[chain]:
            try:
```

```
                web3 = Web3(Web3.HTTPProvider(endpoint))
                if await web3.isConnected():
                    return web3
            except:
                continue

        raise Exception(f"Failed to connect to {chain}")

    def validate_data_point(self, data: YieldDataPoint) -> bool:
        """
        Validate data quality
        """
        # Check APY bounds
        if not (self.data_quality_thresholds['min_apy'] <=
                data.apy <=
                self.data_quality_thresholds['max_apy']):
            return False

        # Check utilization bounds
        if not (0 <= data.utilization <=
                self.data_quality_thresholds['max_utilization']):
            return False

        # Check data freshness
        if (int(time.time()) - data.timestamp >
            self.data_quality_thresholds['stale_data_seconds']):
            return False

        return True

### 2.2 Chainlink Data Streams Integration

class ChainlinkDataStreamsCollector:
```

```
"""
Integration with Chainlink Data Streams for verified yield data
"""

def __init__(self):
    self.stream_endpoints = {
        'mainnet': 'wss://data-streams.chainlink.eth',
        'arbitrum': 'wss://data-streams.chainlink.arbitrum',
        'polygon': 'wss://data-streams.chainlink.polygon',
        'optimism': 'wss://data-streams.chainlink.optimism'
    }

    self.feed_ids = {
        'aave_usdc_apy': '0x1234...',
        'compound_usdc_apy': '0x5678...',
        'morpho_usdc_apy': '0x9abc...',
        'spark_usdc_apy': '0xdef0...'
    }

async def connect_and_subscribe(self):
    """
    Connect to Chainlink Data Streams and subscribe to yield feeds
    """
    async with aiohttp.ClientSession() as session:
        for chain, endpoint in self.stream_endpoints.items():
            async with session.ws_connect(endpoint) as ws:
                # Subscribe to yield feeds
                await ws.send_json({
                    'action': 'subscribe',
                    'feeds': list(self.feed_ids.values())
                })

                # Handle incoming data
```

```
                    async for msg in ws:
                        if msg.type == aiohttp.WSMsgType.TEXT:
                            await self.process_stream_data(msg.data, chain)

    async def process_stream_data(self, data: str, chain: str):
        """
        Process incoming Chainlink data stream
        """
        parsed = json.loads(data)

        # Validate signature
        if not self.verify_chainlink_signature(parsed):
            return

        # Extract yield data
        feed_id = parsed['feedId']
        value = Decimal(parsed['value']) / Decimal(10**8)  # 8 decimals
        timestamp = parsed['timestamp']

        # Store in time series database
        await self.store_yield_data(
            feed_id=feed_id,
            chain=chain,
            value=value,
            timestamp=timestamp,
            source='chainlink'
        )

### 2.3 Gas Price Monitoring

class GasPriceMonitor:
    """
    Real-time gas price monitoring for optimization windows
```

```
"""

def __init__(self):
    self.gas_apis = {
        'ethereum': 'https://api.etherscan.io/api?module=gastracker&action=gasoracle'
        'arbitrum': 'https://api.arbiscan.io/api?module=proxy&action=eth_gasPrice',
        'polygon': 'https://api.polygonscan.com/api?module=gastracker&action=gasoracl
        'optimism': 'https://api-optimistic.etherscan.io/api?module=proxy&action=eth_
    }

    self.historical_gas_data = {}
    self.optimization_thresholds = {
        'ethereum': {'low': 15, 'medium': 30, 'high': 50},
        'arbitrum': {'low': 0.1, 'medium': 0.5, 'high': 1},
        'polygon': {'low': 30, 'medium': 100, 'high': 200},
        'optimism': {'low': 0.01, 'medium': 0.1, 'high': 0.5}
    }

async def monitor_gas_prices(self):
    """
    Continuously monitor gas prices across chains
    """
    while True:
        tasks = []
        for chain in self.gas_apis.keys():
            tasks.append(self.fetch_gas_price(chain))

        prices = await asyncio.gather(*tasks)

        # Analyze for optimization windows
        opportunities = self.identify_optimization_windows(prices)
        if opportunities:
            await self.notify_optimization_opportunities(opportunities)
```

```
            await asyncio.sleep(10)  # Check every 10 seconds

    def identify_optimization_windows(
        self,
        current_prices: Dict[str, float]
    ) -> List[Dict]:
        """
        Identify when gas prices present optimization opportunities
        """
        opportunities = []

        for chain, price in current_prices.items():
            # Get 24h average
            avg_24h = self.calculate_average_gas(chain, hours=24)

            if avg_24h and price < avg_24h * 0.7:  # 30% below average
                opportunities.append({
                    'chain': chain,
                    'current_gas': price,
                    'avg_24h': avg_24h,
                    'savings_percent': (1 - price/avg_24h) * 100,
                    'recommendation': 'execute_rebalance',
                    'urgency': 'high' if price < avg_24h * 0.5 else 'medium'
                })

        return opportunities

## 3. Data Processing & Analytics

### 3.1 Stream Processing Engine

class YieldStreamProcessor:
```

```
    """
    Real-time processing of yield data streams
    """

    def __init__(self):
        self.processing_pipeline = [
            self.normalize_data,
            self.calculate_effective_yield,
            self.detect_anomalies,
            self.enrich_with_risk_scores,
            self.calculate_optimization_signals
        ]

        self.risk_factors = {
            'protocol_maturity': 0.2,
            'tvl_stability': 0.2,
            'utilization_volatility': 0.15,
            'historical_incidents': 0.25,
            'audit_score': 0.2
        }

    async def process_yield_update(self, data: YieldDataPoint) -> Dict:
        """
        Process incoming yield data through analytics pipeline
        """
        processed_data = data

        for processor in self.processing_pipeline:
            processed_data = await processor(processed_data)

        return processed_data

    async def calculate_effective_yield(self, data: YieldDataPoint) -> Dict:
```

```python
        """
        Calculate yield after accounting for gas costs and risks
        """
        # Get current gas costs
        gas_cost = await self.estimate_transaction_costs(
            data.chain,
            data.protocol
        )

        # Calculate minimum profitable position
        min_position = (gas_cost * 365 * 24 / 4) / (data.apy / 100)

        # Adjust yield for risk
        risk_score = await self.calculate_protocol_risk(
            data.protocol,
            data.chain
        )
        risk_adjusted_apy = data.apy * (1 - risk_score)

        return {
            **data.__dict__,
            'effective_apy': risk_adjusted_apy,
            'min_profitable_position': min_position,
            'gas_cost_impact': (gas_cost / min_position) * 100,
            'risk_score': risk_score
        }

    async def detect_anomalies(self, data: Dict) -> Dict:
        """
        Detect anomalous yield movements
        """
        # Get historical data
        historical = await self.get_historical_yields(
```

```
        data['protocol'],
        data['chain'],
        hours=168  # 7 days
    )

    if not historical:
        return data

    # Calculate statistics
    yields = [h['apy'] for h in historical]
    mean = statistics.mean(yields)
    std = statistics.stdev(yields) if len(yields) > 1 else 0

    # Detect anomalies (3 sigma)
    z_score = (data['apy'] - mean) / std if std > 0 else 0
    is_anomaly = abs(z_score) > 3

    # Identify anomaly type
    anomaly_type = None
    if is_anomaly:
        if data['apy'] > mean + 3*std:
            anomaly_type = 'spike_up'
        elif data['apy'] < mean - 3*std:
            anomaly_type = 'spike_down'

    return {
        **data,
        'anomaly_detected': is_anomaly,
        'anomaly_type': anomaly_type,
        'z_score': z_score,
        'historical_mean': mean,
        'historical_std': std
    }
```

```python
### 3.2 Liquidity Analysis Engine

class LiquidityAnalyzer:
    """
    Analyze liquidity depth for large rebalancing operations
    """

    def __init__(self):
        self.dex_aggregators = {
            'ethereum': ['0x', '1inch', 'paraswap'],
            'arbitrum': ['1inch', 'paraswap', 'odos'],
            'polygon': ['1inch', 'paraswap', '0x'],
            'optimism': ['1inch', 'velodrome', 'beethoven']
        }

        self.slippage_models = {}

    async def analyze_rebalancing_liquidity(
        self,
        from_protocol: str,
        to_protocol: str,
        amount: Decimal,
        chain: str
    ) -> Dict:
        """
        Analyze liquidity impact of large rebalancing
        """
        # Get available liquidity
        exit_liquidity = await self.get_protocol_liquidity(
            from_protocol,
            chain,
            'withdraw'
```

```
    )

    entry_liquidity = await self.get_protocol_liquidity(
        to_protocol,
        chain,
        'deposit'
    )

    # Calculate slippage estimates
    exit_slippage = self.estimate_slippage(
        amount,
        exit_liquidity,
        'withdraw'
    )

    entry_slippage = self.estimate_slippage(
        amount,
        entry_liquidity,
        'deposit'
    )

    # Determine optimal execution strategy
    if amount > exit_liquidity * Decimal('0.1'):  # >10% of liquidity
        strategy = 'split_execution'
        recommended_chunks = math.ceil(amount / (exit_liquidity * Decimal('0.05')))
    else:
        strategy = 'single_execution'
        recommended_chunks = 1

    return {
        'amount': amount,
        'exit_liquidity': exit_liquidity,
        'entry_liquidity': entry_liquidity,
```

```
            'estimated_exit_slippage': exit_slippage,
            'estimated_entry_slippage': entry_slippage,
            'total_slippage': exit_slippage + entry_slippage,
            'execution_strategy': strategy,
            'recommended_chunks': recommended_chunks,
            'estimated_execution_time': recommended_chunks * 15 * 60  # 15 min per chunk
        }

    def estimate_slippage(
        self,
        amount: Decimal,
        available_liquidity: Decimal,
        operation_type: str
    ) -> Decimal:
        """
        Estimate slippage based on amount and liquidity
        """
        if available_liquidity == 0:
            return Decimal('1')  # 100% slippage (impossible)

        # Liquidity utilization ratio
        utilization = amount / available_liquidity

        # Non-linear slippage model
        if utilization < Decimal('0.01'):  # <1% of liquidity
            slippage = utilization * Decimal('0.1')  # 0.1% per 1% utilized
        elif utilization < Decimal('0.1'):  # <10% of liquidity
            slippage = Decimal('0.001') + (utilization - Decimal('0.01')) * Decimal('0.5'
        else:  # >10% of liquidity
            slippage = Decimal('0.046') + (utilization - Decimal('0.1')) * Decimal('2')

        return slippage
```

```python
### 3.3 Risk Scoring Engine

class ProtocolRiskScorer:
    """
    Calculate comprehensive risk scores for protocols
    """

    def __init__(self):
        self.risk_metrics = {
            'tvl_threshold': 10_000_000,  # $10M minimum
            'age_threshold': 365,  # 1 year minimum
            'audit_requirements': ['tier1', 'tier2'],
            'incident_penalty': 0.2  # 20% penalty per incident
        }

        self.protocol_data = {
            'aave_v3': {
                'launch_date': '2022-03-16',
                'audits': ['openZeppelin', 'trailOfBits', 'certik'],
                'incidents': 0,
                'governance': 'dao'
            },
            'compound_v3': {
                'launch_date': '2022-08-26',
                'audits': ['openZeppelin', 'chainSecurity'],
                'incidents': 0,
                'governance': 'dao'
            },
            'morpho': {
                'launch_date': '2022-01-08',
                'audits': ['chainSecurity', 'spearbit'],
                'incidents': 0,
                'governance': 'multisig'
```

```
        },
        'spark': {
            'launch_date': '2023-05-09',
            'audits': ['chainSecurity'],
            'incidents': 0,
            'governance': 'dao'
        }
    }

async def calculate_risk_score(
    self,
    protocol: str,
    chain: str,
    current_tvl: Decimal
) -> Dict:
    """
    Calculate comprehensive risk score (0-1, lower is better)
    """
    scores = {}

    # TVL risk (inverse relationship)
    tvl_score = 1 / (1 + math.log10(
        float(current_tvl) / self.risk_metrics['tvl_threshold']
    )) if current_tvl > self.risk_metrics['tvl_threshold'] else 0.9
    scores['tvl'] = tvl_score

    # Age risk
    protocol_age = self.calculate_protocol_age(protocol)
    age_score = 1 / (1 + protocol_age / self.risk_metrics['age_threshold']) \
              if protocol_age > 0 else 1
    scores['age'] = age_score

    # Audit risk
```

```
        audit_score = self.calculate_audit_score(protocol)
        scores['audit'] = audit_score

        # Incident risk
        incident_score = self.protocol_data[protocol]['incidents'] * \
                         self.risk_metrics['incident_penalty']
        scores['incidents'] = min(incident_score, 1)

        # Governance risk
        gov_scores = {
            'dao': 0.2,
            'multisig': 0.4,
            'admin': 0.8,
            'immutable': 0.1
        }
        scores['governance'] = gov_scores.get(
            self.protocol_data[protocol]['governance'],
            0.5
        )

        # Calculate weighted total
        weights = {
            'tvl': 0.25,
            'age': 0.20,
            'audit': 0.25,
            'incidents': 0.20,
            'governance': 0.10
        }

        total_score = sum(
            scores[metric] * weight
            for metric, weight in weights.items()
        )
```

```
        return {
            'protocol': protocol,
            'chain': chain,
            'total_risk_score': total_score,
            'risk_components': scores,
            'risk_rating': self.get_risk_rating(total_score),
            'recommendations': self.get_risk_recommendations(scores)
        }

    def get_risk_rating(self, score: float) -> str:
        """
        Convert numeric score to risk rating
        """
        if score < 0.2:
            return 'very_low'
        elif score < 0.4:
            return 'low'
        elif score < 0.6:
            return 'medium'
        elif score < 0.8:
            return 'high'
        else:
            return 'very_high'

## 4. Real-time Dashboard Specifications

### 4.1 WebSocket Server

class YieldMaxWebSocketServer:
    """
    Real-time data delivery for dashboard
    """
```

```
def __init__(self):
    self.connections = set()
    self.subscription_types = {
        'yields': self.stream_yields,
        'gas': self.stream_gas_prices,
        'liquidity': self.stream_liquidity,
        'alerts': self.stream_alerts,
        'optimization': self.stream_optimization_signals
    }

async def handle_connection(self, websocket, path):
    """
    Handle new WebSocket connection
    """
    self.connections.add(websocket)
    try:
        async for message in websocket:
            await self.handle_message(websocket, message)
    finally:
        self.connections.remove(websocket)

async def broadcast_update(self, update_type: str, data: Dict):
    """
    Broadcast updates to all connected clients
    """
    message = json.dumps({
        'type': update_type,
        'timestamp': int(time.time()),
        'data': data
    })

    # Send to all connected clients
```

```
        if self.connections:
            await asyncio.gather(
                *[ws.send(message) for ws in self.connections],
                return_exceptions=True
            )

    async def stream_yields(self, websocket):
        """
        Stream real-time yield updates
        """
        while True:
            # Get latest yield data
            yields = await self.get_current_yields()

            # Format for dashboard
            formatted = {
                'yields': [
                    {
                        'protocol': y['protocol'],
                        'chain': y['chain'],
                        'apy': float(y['apy']),
                        'effective_apy': float(y['effective_apy']),
                        'tvl': float(y['tvl']),
                        'utilization': float(y['utilization']),
                        'risk_score': float(y['risk_score']),
                        'trending': y['apy'] > y['historical_mean']
                    }
                    for y in yields
                ],
                'best_opportunity': self.identify_best_opportunity(yields),
                'update_latency': self.calculate_latency()
            }
```

```
            await websocket.send(json.dumps({
                'type': 'yield_update',
                'data': formatted
            }))

            await asyncio.sleep(5)  # Update every 5 seconds
```

4.2 Dashboard Data Schema

```typescript
// TypeScript interfaces for dashboard data

interface YieldData {
  protocol: string;
  chain: string;
  apy: number;
  effectiveApy: number;
  tvl: number;
  utilization: number;
  riskScore: number;
  trending: boolean;
  lastUpdate: number;
}

interface GasData {
  chain: string;
  currentGwei: number;
  avg24h: number;
  trend: 'up' | 'down' | 'stable';
  optimizationWindow: boolean;
  savingsPercent?: number;
}
```

```
interface LiquidityData {
  protocol: string;
  chain: string;
  availableLiquidity: number;
  maxSingleTrade: number;
  estimatedSlippage: SlippageCurve;
}

interface OptimizationSignal {
  id: string;
  type: 'rebalance' | 'harvest' | 'migration';
  urgency: 'low' | 'medium' | 'high' | 'critical';
  fromProtocol: string;
  toProtocol: string;
  amount: number;
  estimatedProfit: number;
  gasConst: number;
  netProfit: number;
  confidence: number;
  expiresAt: number;
}

interface DashboardState {
  yields: YieldData[];
  gas: GasData[];
  liquidity: LiquidityData[];
  signals: OptimizationSignal[];
  metrics: {
    dataAccuracy: number;
    updateLatency: number;
    systemUptime: number;
    activeProtocols: number;
    totalTVL: number;
```

```
    };
  }
```

5. API Documentation

5.1 REST API Endpoints

yaml

```yaml
openapi: 3.0.0
info:
  title: YieldMax Data API
  version: 1.0.0
  description: Real-time yield optimization data API

paths:
  /api/v1/yields/current:
    get:
      summary: Get current yield rates
      parameters:
        - name: chain
          in: query
          schema:
            type: string
            enum: [ethereum, arbitrum, polygon, optimism]
        - name: protocol
          in: query
          schema:
            type: string
            enum: [aave_v3, compound_v3, morpho, spark]
        - name: include_risk
          in: query
          schema:
            type: boolean
            default: true
      responses:
        200:
          description: Current yield data
          content:
            application/json:
              schema:
                type: array
```

```yaml
              items:
                $ref: '#/components/schemas/YieldData'

  /api/v1/yields/historical:
    get:
      summary: Get historical yield data
      parameters:
        - name: chain
          in: query
          required: true
        - name: protocol
          in: query
          required: true
        - name: from
          in: query
          schema:
            type: integer
            description: Unix timestamp
        - name: to
          in: query
          schema:
            type: integer
        - name: interval
          in: query
          schema:
            type: string
            enum: [1m, 5m, 1h, 1d]
            default: 1h

  /api/v1/gas/current:
    get:
      summary: Get current gas prices
      responses:
```

```yaml
      200:
        description: Gas prices by chain

/api/v1/liquidity/depth:
  get:
    summary: Get liquidity depth analysis
    parameters:
      - name: protocol
        in: query
        required: true
      - name: chain
        in: query
        required: true
      - name: amount
        in: query
        schema:
          type: number
          description: Amount in USD

/api/v1/optimization/signals:
  get:
    summary: Get current optimization signals
    parameters:
      - name: min_profit
        in: query
        schema:
          type: number
          default: 100
      - name: max_risk
        in: query
        schema:
          type: number
          default: 0.5
```

```yaml
/api/v1/ws:
  get:
    summary: WebSocket endpoint for real-time updates
    description: |
      Connect to receive real-time updates.
      Subscribe to channels: yields, gas, liquidity, signals
```

5.2 Data Models

```
python
```

```python
# SQLAlchemy models for time series data

from sqlalchemy import Column, Integer, String, Numeric, DateTime, Index
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class YieldSnapshot(Base):
    __tablename__ = 'yield_snapshots'

    id = Column(Integer, primary_key=True)
    timestamp = Column(DateTime, nullable=False)
    protocol = Column(String(50), nullable=False)
    chain = Column(String(50), nullable=False)
    asset = Column(String(50), nullable=False)
    apy = Column(Numeric(10, 4), nullable=False)
    tvl = Column(Numeric(20, 2), nullable=False)
    utilization = Column(Numeric(5, 2), nullable=False)
    block_number = Column(Integer, nullable=False)
    data_source = Column(String(50), nullable=False)

    __table_args__ = (
        Index('idx_protocol_chain_time', 'protocol', 'chain', 'timestamp'),
        Index('idx_timestamp', 'timestamp'),
    )

class GasPrice(Base):
    __tablename__ = 'gas_prices'

    id = Column(Integer, primary_key=True)
    timestamp = Column(DateTime, nullable=False)
    chain = Column(String(50), nullable=False)
    fast = Column(Numeric(10, 2), nullable=False)
```

```
    standard = Column(Numeric(10, 2), nullable=False)
    slow = Column(Numeric(10, 2), nullable=False)

    __table_args__ = (
        Index('idx_chain_time', 'chain', 'timestamp'),
    )

class OptimizationSignal(Base):
    __tablename__ = 'optimization_signals'

    id = Column(String(64), primary_key=True)  # UUID
    created_at = Column(DateTime, nullable=False)
    signal_type = Column(String(50), nullable=False)
    from_protocol = Column(String(50))
    to_protocol = Column(String(50))
    chain = Column(String(50), nullable=False)
    amount = Column(Numeric(20, 2), nullable=False)
    estimated_profit = Column(Numeric(10, 2), nullable=False)
    gas_cost = Column(Numeric(10, 2), nullable=False)
    net_profit = Column(Numeric(10, 2), nullable=False)
    confidence = Column(Numeric(3, 2), nullable=False)
    expires_at = Column(DateTime, nullable=False)
    executed = Column(Boolean, default=False)
```

6. Data Quality Monitoring

6.1 Monitoring System

```
python
```

```python
class DataQualityMonitor:
    """
    Continuous monitoring of data quality metrics
    """

    def __init__(self):
        self.quality_metrics = {
            'accuracy': [],
            'latency': [],
            'completeness': [],
            'consistency': []
        }

        self.alert_thresholds = {
            'accuracy': 0.995,  # 99.5%
            'latency': 30,      # 30 seconds
            'completeness': 0.98,  # 98%
            'consistency': 0.99    # 99%
        }

    async def monitor_data_quality(self):
        """
        Main monitoring loop
        """
        while True:
            # Check accuracy
            accuracy = await self.check_accuracy()
            self.quality_metrics['accuracy'].append(accuracy)

            # Check latency
            latency = await self.check_latency()
            self.quality_metrics['latency'].append(latency)
```

```python
            # Check completeness
            completeness = await self.check_completeness()
            self.quality_metrics['completeness'].append(completeness)

            # Check consistency
            consistency = await self.check_consistency()
            self.quality_metrics['consistency'].append(consistency)

            # Generate alerts if needed
            await self.check_alerts()

            # Log metrics
            await self.log_metrics()

            await asyncio.sleep(60)  # Check every minute

    async def check_accuracy(self) -> float:
        """
        Compare multiple data sources for accuracy
        """
        # Get data from multiple sources
        chainlink_data = await self.get_chainlink_yields()
        onchain_data = await self.get_onchain_yields()
        api_data = await self.get_api_yields()

        # Calculate agreement percentage
        total_comparisons = 0
        agreements = 0

        for protocol in chainlink_data:
            if protocol in onchain_data and protocol in api_data:
                total_comparisons += 1
```

```python
            # Allow 1% variance
            chainlink_val = chainlink_data[protocol]
            onchain_val = onchain_data[protocol]

            if abs(chainlink_val - onchain_val) / chainlink_val < 0.01:
                agreements += 1

    return agreements / total_comparisons if total_comparisons > 0 else 0

async def check_latency(self) -> float:
    """
    Measure data update latency
    """
    # Trigger test transaction
    test_tx = await self.send_test_transaction()

    # Measure time until it appears in our data
    start_time = time.time()
    timeout = 60  # 1 minute timeout

    while time.time() - start_time < timeout:
        if await self.check_test_transaction_processed(test_tx):
            return time.time() - start_time

        await asyncio.sleep(1)

    return timeout  # Max latency

async def generate_alert(self, metric: str, value: float, threshold: float):
    """
    Generate alerts for quality issues
    """
    alert = {
```

```
            'timestamp': int(time.time()),
            'metric': metric,
            'value': value,
            'threshold': threshold,
            'severity': self.calculate_severity(metric, value, threshold),
            'message': f"{metric} degraded: {value:.2f} (threshold: {threshold})"
        }

        # Send to monitoring systems
        await self.send_to_pagerduty(alert)
        await self.send_to_slack(alert)
        await self.log_alert(alert)
```

7. Performance Metrics

7.1 System Performance Tracking

```
python
```

```python
class PerformanceTracker:
    """
    Track and report system performance metrics
    """

    def __init__(self):
        self.metrics = {
            'data_points_processed': 0,
            'api_requests_served': 0,
            'websocket_messages_sent': 0,
            'average_processing_time': 0,
            'uptime_start': time.time()
        }

    def calculate_metrics(self) -> Dict:
        """
        Calculate current performance metrics
        """
        uptime_seconds = time.time() - self.metrics['uptime_start']
        uptime_percent = (uptime_seconds / (uptime_seconds + self.downtime)) * 100

        return {
            'uptime_percent': min(uptime_percent, 100),
            'data_accuracy': self.calculate_accuracy(),
            'update_latency': self.calculate_average_latency(),
            'throughput': {
                'data_points_per_second': self.metrics['data_points_processed'] / uptime_
                'api_requests_per_second': self.metrics['api_requests_served'] / uptime_se
            },
            'error_rates': {
                'data_collection_errors': self.error_counts['collection'] / self.total_at
                'processing_errors': self.error_counts['processing'] / self.total_attempts
```

```
        }
    }
```

8. Implementation Timeline

Phase 1: Core Infrastructure (Week 1-2)

- Set up data collection for all protocols
- Implement Chainlink Data Streams integration
- Deploy time series database
- Basic API endpoints

Phase 2: Analytics Engine (Week 3-4)

- Stream processing pipeline
- Risk scoring implementation
- Liquidity analysis
- Gas optimization algorithms

Phase 3: Real-time Dashboard (Week 5-6)

- WebSocket server deployment
- Dashboard frontend
- Alert system
- Performance optimization

Phase 4: Production Hardening (Week 7-8)

- Redundancy implementation
- Monitoring deployment

- Load testing
- Documentation finalization

Conclusion

This data infrastructure provides YieldMax with institutional-grade analytics capabilities, enabling profitable yield optimization decisions with:

- **>99.5% accuracy** through multi-source validation
- **<30 second latency** via optimized data pipelines
- **>99.9% uptime** with redundant systems
- **Real-time insights** for immediate action

The system is designed to scale with YieldMax's growth while maintaining performance standards critical for mainnet profitability.