Integration Guide: Replacing Mock Data with Real-World Threat Data

Table of Contents

Prerequisites

Required Skills

- Basic understanding of TypeScript/JavaScript
- Familiarity with REST APIs or database queries
- Basic knowledge of network security concepts (IPs, threat types)
- Understanding of async/await patterns in JavaScript

Required Tools & Access

☐ Node.js and npm installed
☐ Access to your threat data source (API, database, or log files)
☐ API credentials (if using external threat intelligence feeds)
☐ Development environment setup (VS Code recommended)
☐ Git for version control

Recommended Security Intelligence Sources

Choose one or more based on your needs:

Free/Open Source:

- AbuseIPDB API (https://www.abuseipdb.com)
- AlienVault OTX (https://otx.alienvault.com)
- GreyNoise (https://www.greynoise.io)

- Shodan API (https://www.shodan.io)

Commercial:

- IBM X-Force Exchange
- Recorded Future
- Threat Connect
- Cisco Talos Intelligence

Internal Sources:

- Firewall logs (pfSense, Fortinet, Cisco ASA)
- IDS/IPS systems (Snort, Suricata)
- SIEM platforms (Splunk, ELK Stack, QRadar)
- Web server logs (Apache, Nginx)

Understanding the Current Data Structure

Current Mock Data Interface

The application expects data in this TypeScript format:

typescript

```
// Severity levels
const Severity = {
  CRITICAL: 'Critical',
  HIGH: 'High',
  MEDIUM: 'Medium',
  LOW: 'Low',
}

// Threat types
const ThreatType = {
  DDOS: 'DDoS',
  SQL_INJECTION: 'SQL Injection',
  MALWARE: 'Malware',
  PHISHING: 'Phishing',
  BRUTE_FORCE: 'Brute Force',
  XSS: 'Cross-Site Scripting',
  ANOMALY: 'Traffic Anomaly',
}

// Main threat log structure
interface ThreatLog {
  id: string;                    // Unique identifier
  timestamp: string;             // ISO 8601 format: "2024-12-02T15:30:00.000Z"
  sourceIP: string;              // IPv4 or IPv6: "192.168.1.100"
  destinationIP: string;         // Your server/network IP
  severity: Severity;            // One of: Critical, High, Medium, Low
  type: ThreatType;              // One of the threat types above
  confidence: number;            // 0-100 (percentage)
  status: 'Detected' | 'Blocked' | 'Monitoring';
  country: string;               // Country name: "Zimbabwe", "USA", etc.
}
```

Example Mock Data Record

```
json
```

```
{
  "id": "abc123xyz",
  "timestamp": "2024-12-02T15:30:45.123Z",
  "sourceIP": "41.223.145.67",
  "destinationIP": "192.168.1.100",
  "severity": "High",
  "type": "SQL Injection",
  "confidence": 87,
  "status": "Blocked",
  "country": "Zimbabwe"
}
```

Data Source Options

Option A: REST API Integration

Best for: External threat intelligence feeds, cloud-based security platforms

Pros:

- Real-time data access
- Managed by third parties
- Regular updates

Cons:

- Rate limits
- API costs
- Network dependency

Option B: Database Integration

Best for: Enterprise SIEM systems, internal security databases

Pros:

- Full control over data
- No rate limits
- Historical data access

Cons:

- Requires database setup
- Need backend API layer

- Maintenance overhead

Option C: Log File Parsing

Best for: Firewall logs, server logs, IDS/IPS exports

Pros:

- Work with existing infrastructure
- No external dependencies
- Cost-effective

Cons:

- Manual processing required
- Delayed updates
- Storage requirements

Option D: WebSocket/Streaming

Best for: Real-time threat feeds, high-volume environments

Pros:

- True real-time updates
- Efficient for high volume
- Bidirectional communication

Cons:

- Complex implementation
- Connection management
- Server requirements

Dataset Format Requirements

Minimum Required Fields

Your data source **MUST** provide these fields (or allow you to derive them):

Field	Type	Example	Notes
id	string	"evt_001"	Unique identifier for each threat
timestamp	string/date	"2024-12-02T15:30:00Z"	Must be convertible to ISO 8601
sourceIP	string	"192.168.1.50"	IPv4 or IPv6 address
destinationIP	string	"10.0.0.1"	Your protected network/server
severity	string	"High"	Must map to: Critical, High, Medium, or Low
type	string	"DDoS"	Must map to one of the 7 threat types
confidence	number	85	0-100 scale, can be calculated if not provided
status	string	"Blocked"	Detected, Blocked, or Monitoring
country	string	"Zimbabwe"	Can be derived from IP using GeoIP

Optional But Recommended Fields

- **port**: Target port number
- **protocol**: TCP, UDP, ICMP, etc.
- **payload**: Attack payload or signature
- **user_agent**: For web-based attacks
- **attack_signature**: IDS signature ID
- **bytes_transferred**: Data volume
- **duration**: Attack duration in seconds

Data Mapping Examples

If your data uses different field names, you'll need to map them:

Example: Firewall Log Mapping

javascript

```
// Your firewall log format
{
  "event_id": "12345",
  "time": "2024-12-02 15:30:00",
  "src": "41.223.145.67",
  "dst": "192.168.1.100",
  "action": "DROP",
  "threat_level": "5",
  "attack_type": "SQL_ATTACK"
}

// Maps to ThreatLog as:
{
  "id": "12345",
  "timestamp": "2024-12-02T15:30:00.000Z",
  "sourceIP": "41.223.145.67",
  "destinationIP": "192.168.1.100",
  "status": "Blocked",  // "DROP" -> "Blocked"
  "severity": "High",   // threat_level 5 -> "High"
  "type": "SQL Injection",  // SQL_ATTACK -> "SQL Injection"
  "confidence": 90,     // Calculated based on threat_level
  "country": "Zimbabwe" // Derived from IP geolocation
}
```

Step-by-Step Integration

Step 1: Create a Data Service File

Create a new file: `src/services/realDataService.ts`

```
typescript
```

```
import type { ThreatLog } from '../types';
import { Severity, ThreatType } from '../types';

// Configuration
const API_ENDPOINT = 'YOUR_API_ENDPOINT_HERE';
const API_KEY = import.meta.env.VITE_THREAT_API_KEY;

/**
 * Fetch threat logs from your data source
 * @param limit - Number of records to fetch
 * @returns Promise<ThreatLog[]>
 */
export const fetchThreatLogs = async (limit: number = 50): Promise<ThreatLog[]> => {
  try {
    const response = await fetch(`${API_ENDPOINT}/threats?limit=${limit}`, {
      headers: {
        'Authorization': `Bearer ${API_KEY}`,
        'Content-Type': 'application/json'
      }
    });

    if (!response.ok) {
      throw new Error(`API Error: ${response.status}`);
    }

    const rawData = await response.json();

    // Transform raw data to ThreatLog format
    return rawData.map(transformToThreatLog);

  } catch (error) {
    console.error('Failed to fetch threat logs:', error);
    return []; // Return empty array on error
  }
};

/**
 * Transform raw API data to ThreatLog format
 */
const transformToThreatLog = (rawLog: any): ThreatLog => {
  return {
    id: rawLog.id || rawLog.event_id || String(Date.now()),
    timestamp: convertToISO(rawLog.timestamp || rawLog.time),
    sourceIP: rawLog.sourceIP || rawLog.src || rawLog.source,
    destinationIP: rawLog.destinationIP || rawLog.dst || rawLog.destination,
    severity: mapSeverity(rawLog.severity || rawLog.threat_level),
    type: mapThreatType(rawLog.type || rawLog.attack_type),
    confidence: rawLog.confidence || calculateConfidence(rawLog),
    status: mapStatus(rawLog.status || rawLog.action),
    country: rawLog.country || 'Unknown'
  };
};

/**
 * Convert various date formats to ISO 8601
 */
const convertToISO = (dateValue: any): string => {
  if (!dateValue) return new Date().toISOString();

  const date = new Date(dateValue);
  return isNaN(date.getTime()) ? new Date().toISOString() : date.toISOString();
};

/**
 * Map your severity values to application severity
 */
const mapSeverity = (rawSeverity: any): typeof Severity[keyof typeof Severity] => {
  // Map numeric values (1-5 scale)
  if (typeof rawSeverity === 'number') {
    if (rawSeverity >= 4) return Severity.CRITICAL;
    if (rawSeverity >= 3) return Severity.HIGH;
    if (rawSeverity >= 2) return Severity.MEDIUM;
    return Severity.LOW;
  }

  // Map string values
  const severity = String(rawSeverity).toUpperCase();
  if (severity.includes('CRIT')) return Severity.CRITICAL;
  if (severity.includes('HIGH')) return Severity.HIGH;
  if (severity.includes('MED')) return Severity.MEDIUM;
  return Severity.LOW;
};

/**
 * Map your threat types to application threat types
 */
const mapThreatType = (rawType: any): typeof ThreatType[keyof typeof ThreatType] => {
  const type = String(rawType).toUpperCase();

  if (type.includes('DDOS') || type.includes('DOS')) return ThreatType.DDOS;
  if (type.includes('SQL') || type.includes('INJECTION')) return ThreatType.SQL_INJECTION;
  if (type.includes('MALWARE') || type.includes('VIRUS')) return ThreatType.MALWARE;
  if (type.includes('PHISH')) return ThreatType.PHISHING;
  if (type.includes('BRUTE') || type.includes('FORCE')) return ThreatType.BRUTE_FORCE;
  if (type.includes('XSS') || type.includes('SCRIPT')) return ThreatType.XSS;

  return ThreatType.ANOMALY; // Default fallback
};

/**
 * Map status values
 */
const mapStatus = (rawStatus: any): 'Detected' | 'Blocked' | 'Monitoring' => {
  const status = String(rawStatus).toUpperCase();

  if (status.includes('BLOCK') || status.includes('DROP') || status.includes('DENY')) {
    return 'Blocked';
  }
  if (status.includes('MONITOR') || status.includes('WATCH')) {
    return 'Monitoring';
  }
  return 'Detected';
};

/**
 * Calculate confidence score if not provided
 */
const calculateConfidence = (rawLog: any): number => {
  // Example: base confidence on available data quality
  let confidence = 70; // Base confidence

  if (rawLog.signature_id) confidence += 10;
  if (rawLog.payload) confidence += 10;
  if (rawLog.verified) confidence += 10;

  return Math.min(confidence, 99); // Cap at 99%
};

/**
 * Stream real-time threats (if your API supports it)
 */
export const subscribeThreatStream = (
  callback: (threat: ThreatLog) => void,
  onError?: (error: Error) => void
): (() => void) => {
  // WebSocket example
  const ws = new WebSocket('wss://your-api.com/threats/stream');

  ws.onmessage = (event) => {
    try {
      const rawLog = JSON.parse(event.data);
      const threat = transformToThreatLog(rawLog);
      callback(threat);
    } catch (error) {
      onError?.(error as Error);
    }
  };

  ws.onerror = (error) => {
    onError?.(new Error('WebSocket error'));
  };

  // Return cleanup function
  return () => ws.close();
};
```

Step 2: Add GeoIP Lookup (Optional but Recommended)

Install a GeoIP library:

```bash
npm install geoip-lite
```

Create `src/services/geoipService.ts`:

```typescript
```

```
import geoip from 'geoip-lite';

/**
 * Get country from IP address
 */
export const getCountryFromIP = (ip: string): string => {
  try {
    const geo = geoip.lookup(ip);
    return geo?.country || 'Unknown';
  } catch (error) {
    console.error('GeoIP lookup failed:', error);
    return 'Unknown';
  }
};

/**
 * Get full location details
 */
export const getLocationFromIP = (ip: string) => {
  try {
    return geoip.lookup(ip);
  } catch (error) {
    return null;
  }
};
```

Update your `transformToThreatLog` function:

```typescript
import { getCountryFromIP } from './geoipService';

const transformToThreatLog = (rawLog: any): ThreatLog => {
  return {
    // ... other fields
    country: rawLog.country || getCountryFromIP(rawLog.sourceIP) || 'Unknown'
  };
};
```

Step 3: Update App.tsx

Replace mock data imports with real data:

```typescript
```

```
// OLD: import { generateHistory, generateMockLog } from './services/mockDataService';
// NEW:
import { fetchThreatLogs, subscribeThreatStream } from './services/realDataService';
```

Update the initialization:

typescript

```
const App: React.FC = () => {
  const [logs, setLogs] = useState<ThreatLog[]>([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);

  // Load initial data
  useEffect(() => {
    const loadInitialData = async () => {
      try {
        setLoading(true);
        const initialLogs = await fetchThreatLogs(50);
        setLogs(initialLogs);
      } catch (err) {
        setError('Failed to load threat data');
        console.error(err);
      } finally {
        setLoading(false);
      }
    };

    loadInitialData();
  }, []);

  // Subscribe to real-time updates
  useEffect(() => {
    const unsubscribe = subscribeThreatStream(
      (newThreat) => {
        setLogs(prev => [newThreat, ...prev].slice(0, 500));
      },
      (error) => {
        console.error('Stream error:', error);
      }
    );

    return () => unsubscribe();
  }, []);

  // Show loading state
  if (loading) {
    return (
      <div className="flex h-screen items-center justify-center bg-slate-950">
        <div className="text-center">
          <div className="animate-spin rounded-full h-16 w-16 border-b-2 border-cyan-500 mx-auto"></div>
          <p className="text-slate-400 mt-4">Loading threat intelligence...</p>
        </div>
      </div>
    );
  }

  // Show error state
  if (error) {
    return (
      <div className="flex h-screen items-center justify-center bg-slate-950">
        <div className="text-center">
          <p className="text-red-400 text-xl">{error}</p>
          <button
            onClick={() => window.location.reload()}
            className="mt-4 px-4 py-2 bg-cyan-500 text-white rounded hover:bg-cyan-600"
          >
            Retry
          </button>
        </div>
      </div>
    );
  }

  // ... rest of your component
};
```

Step 4: Environment Configuration

Update `.env` file:

```
properties

# Google Gemini API (already configured)
VITE_API_KEY=AIzaSyDpuTnQfs9k2UTgzWi-xbj4lIpAmAKlM9g

# Your Threat Intelligence API
VITE_THREAT_API_KEY=your_threat_api_key_here
VITE_THREAT_API_ENDPOINT=https://api.yourprovider.com/v1

# Optional: Refresh intervals (in milliseconds)
VITE_REFRESH_INTERVAL=30000
VITE_MAX_LOGS_DISPLAY=500
```

Step 5: Add Data Validation

Create `src/utils/validation.ts`:

```
typescript
```

```
import type { ThreatLog } from '../types';
import { Severity, ThreatType } from '../types';

/**
 * Validate a threat log record
 */
export const validateThreatLog = (log: any): log is ThreatLog => {
  // Check required fields
  if (!log.id || !log.timestamp || !log.sourceIP) {
    console.warn('Missing required fields:', log);
    return false;
  }

  // Validate IP format
  if (!isValidIP(log.sourceIP) || !isValidIP(log.destinationIP)) {
    console.warn('Invalid IP address:', log);
    return false;
  }

  // Validate severity
  if (!Object.values(Severity).includes(log.severity)) {
    console.warn('Invalid severity:', log.severity);
    return false;
  }

  // Validate threat type
  if (!Object.values(ThreatType).includes(log.type)) {
    console.warn('Invalid threat type:', log.type);
    return false;
  }

  // Validate confidence range
  if (log.confidence < 0 || log.confidence > 100) {
    console.warn('Confidence out of range:', log.confidence);
    return false;
  }

  return true;
};

/**
 * Validate IP address format (IPv4 and IPv6)
 */
const isValidIP = (ip: string): boolean => {
  // IPv4 regex
  const ipv4Regex = /^(\d{1,3}\.){3}\d{1,3}$/;
```

```typescript
  // IPv6 regex (simplified)
  const ipv6Regex = /^([0-9a-fA-F]{0,4}:){7}[0-9a-fA-F]{0,4}$/;

  return ipv4Regex.test(ip) || ipv6Regex.test(ip);
};

/**
 * Sanitize and validate multiple logs
 */
export const validateAndFilterLogs = (logs: any[]): ThreatLog[] => {
  return logs.filter(validateThreatLog);
};
```

Use in your data service:

typescript

```typescript
import { validateAndFilterLogs } from '../utils/validation';

export const fetchThreatLogs = async (limit: number = 50): Promise<ThreatLog[]> => {
  try {
    const response = await fetch(`${API_ENDPOINT}/threats?limit=${limit}`);
    const rawData = await response.json();
    const transformed = rawData.map(transformToThreatLog);

    // Validate before returning
    return validateAndFilterLogs(transformed);

  } catch (error) {
    console.error('Failed to fetch threat logs:', error);
    return [];
  }
};
```

Testing & Validation

Phase 1: Development Testing

1. **Test with Sample Data First**

typescript

```
// Create test data file: src/services/__tests__/sampleData.json
[
  {
    "id": "test_001",
    "timestamp": "2024-12-02T15:30:00Z",
    "sourceIP": "41.223.145.67",
    "destinationIP": "192.168.1.100",
    "severity": "High",
    "type": "SQL Injection",
    "confidence": 87,
    "status": "Blocked",
    "country": "Zimbabwe"
  }
]
```

2. **Create a Test Service**

```typescript
// src/services/testDataService.ts
import sampleData from './__tests__/sampleData.json';

export const fetchTestData = async (): Promise<ThreatLog[]> => {
  // Simulate API delay
  await new Promise(resolve => setTimeout(resolve, 1000));
  return sampleData as ThreatLog[];
};
```

3. **Test Data Transformation**

```typescript
// Test your mapping functions
console.log('Testing transformation...');
const testLog = {
  event_id: "12345",
  time: "2024-12-02 15:30:00",
  src: "41.223.145.67"
};
const transformed = transformToThreatLog(testLog);
console.log('Result:', transformed);
```

Phase 2: Integration Testing

Create a test checklist:

□ API connection successful

□ Data transforms correctly

□ All required fields present

□ Date formats convert properly

□ Severity levels map correctly

□ Threat types map correctly

□ GeoIP lookups work

□ Real-time updates functioning

□ Error handling works

□ Performance acceptable (< 2s load time)

Phase 3: User Acceptance Testing

Test these user scenarios:

1. **Dashboard loads with real data**
2. **Filters work correctly**
3. **Search functions properly**
4. **Export to CSV includes all data**
5. **AI analysis works with real threats**
6. **Real-time updates appear smoothly**
7. **Application handles API failures gracefully**

Validation Checklist

```typescript
```

```
// Add this to your console for quick validation
const validateDeployment = () => {
  console.group('🔍 Deployment Validation');

  console.log('✓ Environment variables:', {
    apiKey: !!import.meta.env.VITE_THREAT_API_KEY,
    endpoint: !!import.meta.env.VITE_THREAT_API_ENDPOINT
  });

  console.log('✓ Data service:', typeof fetchThreatLogs === 'function');
  console.log('✓ Transform function:', typeof transformToThreatLog === 'function');
  console.log('✓ Validation:', typeof validateThreatLog === 'function');

  console.groupEnd();
};

// Run on app startup
validateDeployment();
```

Troubleshooting

Common Issues and Solutions

Issue 1: CORS Errors

Symptom: Console shows "Access-Control-Allow-Origin" errors

Solution:

typescript

```
// Add proxy in vite.config.ts
export default defineConfig({
  server: {
    proxy: {
      '/api': {
        target: 'https://your-api.com',
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/api/, '')
      }
    }
  }
});

// Update API_ENDPOINT
const API_ENDPOINT = '/api';
```

Issue 2: Data Not Appearing

Symptom: Application loads but shows no threats

Debug steps:

```typescript
```

```
// Add debug logging
export const fetchThreatLogs = async (limit: number = 50): Promise<ThreatLog[]> => {
  console.log('🔍 Fetching threats from:', API_ENDPOINT);

  try {
    const response = await fetch(`${API_ENDPOINT}/threats?limit=${limit}`);
    console.log('📡 Response status:', response.status);

    const rawData = await response.json();
    console.log('📦 Raw data received:', rawData.length, 'records');

    const transformed = rawData.map(transformToThreatLog);
    console.log('✨ Transformed data:', transformed.length, 'records');

    const validated = validateAndFilterLogs(transformed);
    console.log('✅ Validated data:', validated.length, 'records');

    return validated;
  } catch (error) {
    console.error('❌ Fetch error:', error);
    return [];
  }
};
```

Issue 3: Incorrect Field Mapping

Symptom: Data appears but values are wrong or "Unknown"

Solution:

1. Log the raw API response:

```typescript
console.log('Raw API response:', JSON.stringify(rawData[0], null, 2));
```

2. Adjust your mapping functions based on actual field names

3. Create a mapping configuration file:

```typescript
```

```
// src/config/fieldMapping.ts
export const FIELD_MAPPING = {
  id: ['id', 'event_id', 'alert_id'],
  timestamp: ['timestamp', 'time', 'datetime', 'created_at'],
  sourceIP: ['sourceIP', 'src', 'source', 'src_ip'],
  destinationIP: ['destinationIP', 'dst', 'dest', 'dst_ip'],
  // ... etc
};
```

Issue 4: Performance Issues

Symptom: Application is slow or unresponsive

Solutions:

1. **Implement pagination:**

```typescript
const [page, setPage] = useState(1);
const ITEMS_PER_PAGE = 50;

const paginatedLogs = useMemo(() => {
  const start = (page - 1) * ITEMS_PER_PAGE;
  return logs.slice(start, start + ITEMS_PER_PAGE);
}, [logs, page]);
```

2. **Debounce real-time updates:**

```typescript
import { debounce } from 'lodash';

const debouncedUpdate = debounce((newThreat: ThreatLog) => {
  setLogs(prev => [newThreat, ...prev].slice(0, 500));
}, 500);
```

3. **Use virtual scrolling for large lists:**

```bash
npm install react-window
```

Issue 5: Invalid Dates

Symptom: Dates show as "Invalid Date" or wrong timezone

Solution:

```typescript
import { parseISO, format } from 'date-fns';

const convertToISO = (dateValue: any): string => {
  try {
    // Handle Unix timestamp (seconds)
    if (typeof dateValue === 'number' && dateValue < 10000000000) {
      return new Date(dateValue * 1000).toISOString();
    }

    // Handle Unix timestamp (milliseconds)
    if (typeof dateValue === 'number') {
      return new Date(dateValue).toISOString();
    }

    // Handle string dates
    if (typeof dateValue === 'string') {
      // Try parsing as ISO first
      const parsed = parseISO(dateValue);
      if (!isNaN(parsed.getTime())) {
        return parsed.toISOString();
      }

      // Fallback to Date constructor
      const date = new Date(dateValue);
      if (!isNaN(date.getTime())) {
        return date.toISOString();
      }
    }

    // Fallback to current time
    return new Date().toISOString();
  } catch (error) {
    console.error('Date conversion error:', error);
    return new Date().toISOString();
  }
};
```

Getting Help

If you encounter issues:

1. **Check the browser console** for error messages
2. **Review the network tab** in DevTools to see API requests/responses

3. **Enable verbose logging** in your data service
4. **Test with mock data** to isolate the issue
5. **Consult your API documentation** for correct field names and formats

---

Deployment Checklist

Before deploying to production:

- □ Remove all console.log statements (or use proper logging)
- □ Set up environment variables on production server
- □ Configure CORS properly on your API
- □ Test with production API credentials
- □ Implement rate limiting if needed
- □ Set up error monitoring (e.g., Sentry)
- □ Configure appropriate cache headers
- □ Test with large datasets (1000+ records)
- □ Verify SSL/TLS certificates
- □ Set up health check endpoint
- □ Document any custom configurations
- □ Create backup strategy for logs
- □ Train team on new data source

Support Resources

Documentation to Review

- TypeScript: https://www.typescriptlang.org/docs/
- React Hooks: https://react.dev/reference/react
- Fetch API: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
- WebSocket API: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

Useful Tools

- **Postman**: Test your API endpoints
- **JSON Formatter**: Validate JSON responses
- **RegEx101**: Test your mapping regex patterns
- **IP Address Lookup**: Verify GeoIP results

Maintenance Guidelines

Regular Tasks

Daily:

- Monitor error logs
- Check API rate limits
- Verify data freshness

Weekly:

- Review threat statistics
- Update threat type mappings if needed
- Check for API updates

Monthly:

- Update dependencies: `npm update`
- Review and optimize queries
- Archive old threat data

Version Control

Always commit changes with clear messages:

```bash
git add src/services/realDataService.ts
git commit -m "feat: integrate AbuseIPDB API for real threat data"
git push origin main
```

Conclusion

You've now successfully integrated real-world threat data into Sentinel AI! The application will:

✅ Fetch real threat intelligence from your data source
✅ Transform it to the correct format
✅ Display it in all dashboards and views
✅ Enable AI analysis on real threats
✅ Provide real-time updates as threats occur

Remember: Start with a test environment, validate thoroughly, and gradually roll out to production.

For questions or issues, contact your development team lead.