CyberSwarm Dashboard Architecture

Table of Contents

System Overview

The CyberSwarm Dashboard is a Next.js 14 application built with the App Router architecture. It serves as the frontend interface for a distributed multi-agent cybersecurity simulation system.

High-Level Architecture

```
┌─────────────────────────────────────────────────────────────────┐
│                         Browser Client                          │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │              React Components (UI Layer)                  │  │
│  │  ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐      │  │
│  │  │  Agent   │ │  Chain   │ │  Logic   │ │Knowledge │      │  │
│  │  │ Control  │ │    of    │ │  Pipes   │ │   Base   │      │  │
│  │  │  Panel   │ │ Thought  │ │ Monitor  │ │ Explorer │      │  │
│  │  └──────────┘ └──────────┘ └──────────┘ └──────────┘      │  │
│  └───────────────────────────────────────────────────────────┘  │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │           State Management & Hooks Layer                  │  │
│  │  ┌──────────────────┐  ┌──────────────────┐               │  │
│  │  │ useSimulation    │  │  Zustand Stores  │               │  │
│  │  │     Stream       │  │  (Global State)  │               │  │
│  │  └──────────────────┘  └──────────────────┘               │  │
│  └───────────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────────┘
                            ↕ HTTP/SSE
┌─────────────────────────────────────────────────────────────────┐
│                   Next.js Server (Edge/Node)                    │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │                    API Routes Layer                       │  │
│  │  ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐      │  │
│  │  │  /api/   │ │  /api/   │ │  /api/   │ │  /api/   │      │  │
│  │  │  agents  │ │simulation│ │knowledge │ │simulation│      │  │
│  │  │          │ │          │ │    -base │ │  /stream │      │  │
│  │  └──────────┘ └──────────┘ └──────────┘ └──────────┘      │  │
│  └───────────────────────────────────────────────────────────┘  │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │              Server-Side Logic & Utilities                │  │
│  │  ┌──────────────────┐  ┌──────────────────┐               │  │
│  │  │  Agent Simulator │  │  Event Emitter   │               │  │
│  │  │   (Mock/Proxy)   │  │   (SSE Handler)  │               │  │
│  │  └──────────────────┘  └──────────────────┘               │  │
│  └───────────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────────┘
                            ↕ HTTP/WebSocket
┌─────────────────────────────────────────────────────────────────┐
│              Backend Orchestrator (External System)             │
│                    Multi-Agent Docker Swarm                     │
└─────────────────────────────────────────────────────────────────┘
```

Technology Stack

Frontend
- **Next.js 14**: React framework with App Router
- **React 18**: UI library with concurrent features
- **TypeScript 5.2**: Type-safe development

Styling
- **Tailwind CSS 3.3**: Utility-first CSS
- **Radix UI**: Accessible component primitives
- **Framer Motion**: Animation library

State Management
- **Zustand**: Lightweight global state

- **React Query**: Server state management
- **SWR**: Data fetching and caching

Data Visualization
- **Recharts**: React charting library
- **Chart.js**: Canvas-based charts
- **Plotly.js**: Interactive scientific plots

Architecture Patterns

1. Component-Based Architecture

The application follows a strict component hierarchy:

```
app/
├── layout.tsx                 # Root layout
├── page.tsx                   # Home page
└── api/                       # API routes
    ├── agents/
    ├── simulation/
    └── knowledge-base/

components/
├── dashboard/                 # Feature components
│   ├── agent-control-panel.tsx
│   ├── chain-of-thought-panel.tsx
│   ├── logic-pipe-visualization.tsx
│   ├── real-time-monitor.tsx
│   └── knowledge-base-explorer.tsx
└── ui/                        # Reusable UI components
    ├── button.tsx
    ├── card.tsx
    ├── dialog.tsx
    └── ...

hooks/                        # Custom React hooks
├── use-simulation-stream.ts
└── ...

lib/                          # Utilities and business logic
├── agents/                    # Agent implementations
├── types.ts                   # TypeScript definitions
├── utils.ts                   # Helper functions
└── api-client.ts             # API communication
```

2. Server-Side Rendering (SSR) Strategy

Static Generation: Used for documentation and static pages

```
// app/page.tsx
export default function HomePage() {
  // Pre-rendered at build time
  return <Dashboard />;
}
```

Server Components: Used for data fetching

```
// app/agents/page.tsx
async function getAgents() {
  const res = await fetch('http://backend/agents');
  return res.json();
}

export default async function AgentsPage() {
  const agents = await getAgents();
  return <AgentList agents={agents} />;
}
```

Client Components: Used for interactivity

```
'use client';

export function AgentControlPanel() {
  const [agents, setAgents] = useState([]);
  // Interactive UI logic
}
```

3. API Route Handlers

Next.js API routes act as a proxy/adapter layer:

```
// app/api/agents/route.ts
import { NextRequest, NextResponse } from 'next/server';

export async function GET(request: NextRequest) {
  // Fetch from backend orchestrator
  const response = await fetch('http://orchestrator:8000/agents');
  const data = await response.json();

  // Transform data if needed
  return NextResponse.json(data);
}

export async function POST(request: NextRequest) {
  const body = await request.json();

  // Forward to backend
  const response = await fetch('http://orchestrator:8000/agents', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(body)
  });

  return NextResponse.json(await response.json());
}
```

Component Structure

Dashboard Components

1. Agent Control Panel

Purpose: Monitor and control individual agents

Key Features:
- Real-time agent status display
- Start/stop agent controls
- Task injection interface
- Agent configuration

Data Flow:

```
User Action → Component State → API Call → Backend
                    ↓
            Update UI Optimistically
                    ↓
            Confirm with SSE Event
```

Implementation:

```tsx
// components/dashboard/agent-control-panel.tsx
export function AgentControlPanel() {
  const { agents, controlAgent, injectTask } = useSimulationStream();

  const handleStartAgent = async (agentId: string) => {
    // Optimistic update
    setAgents(prev => prev.map(a =>
      a.id === agentId ? { ...a, status: 'BUSY' } : a
    ));

    // API call
    await controlAgent(agentId, 'start');
  };

  return (
    <div className="grid grid-cols-3 gap-4">
      {agents.map(agent => (
        <AgentCard
          key={agent.id}
          agent={agent}
          onStart={handleStartAgent}
        />
      ))}
    </div>
  );
}
```

2. Chain of Thought Panel

Purpose: Visualize agent reasoning processes

Key Features:
- Step-by-step reasoning display
- Confidence scores
- Data inspection
- Timeline view

Data Structure:

```
interface ChainOfThought {
  id: string;
  stepNumber: number;
  stepType: string;
  description: string;
  reasoning: string;
  data?: any;
  confidence?: number;
  timestamp: Date;
  agentId: string;
  taskId?: string;
}
```

3. Real-Time Monitor

Purpose: Display live events and system activity

Key Features:
- Event stream display
- Filtering by type/severity
- Event details modal
- Export functionality

Event Types:
- Agent status changes
- Task updates
- Security events
- System notifications

4. Logic Pipe Visualization

Purpose: Show automated decision workflows

Key Features:
- Execution timeline
- Rule application tracking
- Input/output display
- Performance metrics

5. Knowledge Base Explorer

Purpose: Browse and search security intelligence

Key Features:
- Category browsing
- Full-text search
- Advanced filtering
- Detail views

Data Flow

1. Initial Load

```
Browser Request
    ↓
Next.js Server (SSR)
    ↓
Fetch Initial Data from Backend
    ↓
Render HTML with Data
    ↓
Send to Browser
    ↓
Hydrate React Components
    ↓
Establish SSE Connection
    ↓
Start Receiving Real-Time Updates
```

2. Real-Time Updates

```
Backend Event Occurs
    ↓
Orchestrator Emits Event
    ↓
SSE Endpoint Receives Event
    ↓
Format as StreamEvent
    ↓
Send to Connected Clients
    ↓
useSimulationStream Hook Receives
    ↓
Update Component State
    ↓
React Re-renders UI
```

3. User Actions

```
User Clicks Button
    ↓
Event Handler Triggered
    ↓
Optimistic UI Update (Optional)
    ↓
API Call to Next.js Route
    ↓
Next.js Forwards to Backend
    ↓
Backend Processes Request
    ↓
Response Returned
    ↓
UI Updated with Result
    ↓
SSE Confirms Change (Eventually)
```

State Management

Global State (Zustand)

Used for application-wide state:

```
// lib/store.ts
import { create } from 'zustand';

interface AppState {
  theme: 'light' | 'dark';
  sidebarOpen: boolean;
  selectedAgent: string | null;

  setTheme: (theme: 'light' | 'dark') => void;
  toggleSidebar: () => void;
  selectAgent: (agentId: string | null) => void;
}

export const useAppStore = create<AppState>((set) => ({
  theme: 'dark',
  sidebarOpen: true,
  selectedAgent: null,

  setTheme: (theme) => set({ theme }),
  toggleSidebar: () => set((state) => ({ sidebarOpen: !state.sidebarOpen })),
  selectAgent: (agentId) => set({ selectedAgent: agentId })
}));
```

Server State (React Query / SWR)

Used for data fetching and caching:

```
// hooks/use-agents.ts
import useSWR from 'swr';

export function useAgents() {
  const { data, error, mutate } = useSWR('/api/agents', fetcher, {
    refreshInterval: 5000,
    revalidateOnFocus: false
  });

  return {
    agents: data?.agents || [],
    isLoading: !error && !data,
    isError: error,
    refresh: mutate
  };
}
```

Local State (useState)

Used for component-specific state:

```
export function AgentCard({ agent }: { agent: Agent }) {
  const [isExpanded, setIsExpanded] = useState(false);
  const [showDetails, setShowDetails] = useState(false);

  // Component-specific state management
}
```

API Design

RESTful Endpoints

Agents

```
GET    /api/agents                # List all agents
POST   /api/agents                # Control agent (start/stop)
GET    /api/agents/:id            # Get agent details
```

Simulation

```
POST   /api/simulation            # Start/stop simulation
POST   /api/simulation/inject-task  # Inject custom task
GET    /api/simulation/stream     # SSE stream
```

Knowledge Base

```
GET    /api/knowledge-base        # Query knowledge base
POST   /api/knowledge-base        # Add entry (if enabled)
```

Request/Response Format

Standard Response:

```
interface APIResponse<T> {
  success: boolean;
  data?: T;
  error?: {
    code: string;
    message: string;
    details?: any;
  };
  metadata?: {
    timestamp: string;
    requestId: string;
  };
}
```

Error Handling:

```
// app/api/agents/route.ts
export async function GET() {
  try {
    const agents = await fetchAgents();
    return NextResponse.json({
      success: true,
      data: { agents }
    });
  } catch (error) {
    return NextResponse.json({
      success: false,
      error: {
        code: 'FETCH_FAILED',
        message: 'Failed to fetch agents',
        details: error.message
      }
    }, { status: 500 });
  }
}
```

Real-Time Communication

Server-Sent Events (SSE)

Why SSE over WebSocket?
- Simpler implementation
- Automatic reconnection
- Works through proxies/firewalls
- One-way communication sufficient
- Lower overhead

Implementation:

```
// app/api/simulation/stream/route.ts
export async function GET() {
  const encoder = new TextEncoder();

  const stream = new ReadableStream({
    async start(controller) {
      // Send initial status
      const initialData = await getSimulationStatus();
      controller.enqueue(
        encoder.encode(`data: ${JSON.stringify({
          type: 'initial_status',
          data: initialData,
          timestamp: new Date()
        })}\n\n`)
      );

      // Set up event listeners
      const eventHandler = (event: StreamEvent) => {
        controller.enqueue(
          encoder.encode(`data: ${JSON.stringify(event)}\n\n`)
        );
      };

      eventEmitter.on('agent_status', eventHandler);
      eventEmitter.on('event_created', eventHandler);

      // Heartbeat to keep connection alive
      const heartbeat = setInterval(() => {
        controller.enqueue(
          encoder.encode(`data: ${JSON.stringify({
            type: 'heartbeat',
            timestamp: new Date()
          })}\n\n`)
        );
      }, 30000);

      // Cleanup on close
      return () => {
        clearInterval(heartbeat);
        eventEmitter.off('agent_status', eventHandler);
        eventEmitter.off('event_created', eventHandler);
      };
    }
  });

  return new Response(stream, {
    headers: {
      'Content-Type': 'text/event-stream',
      'Cache-Control': 'no-cache',
      'Connection': 'keep-alive'
    }
  });
}
```

Client-Side Hook:

```
// hooks/use-simulation-stream.ts
export function useSimulationStream() {
  const [state, setState] = useState<SimulationState>({
    isConnected: false,
    isRunning: false,
    agents: [],
    recentEvents: []
  });

  useEffect(() => {
    const eventSource = new EventSource('/api/simulation/stream');

    eventSource.onopen = () => {
      setState(prev => ({ ...prev, isConnected: true }));
    };

    eventSource.onmessage = (event) => {
      const streamEvent: StreamEvent = JSON.parse(event.data);

      setState(prev => {
        switch (streamEvent.type) {
          case 'agent_status':
            return {
              ...prev,
              agents: updateAgentStatus(prev.agents, streamEvent.data)
            };
          case 'event_created':
            return {
              ...prev,
              recentEvents: [streamEvent.data, ...prev.recentEvents.slice(0, 49)]
            };
          default:
            return prev;
        }
      });
    };

    eventSource.onerror = () => {
      setState(prev => ({ ...prev, isConnected: false }));
    };

    return () => eventSource.close();
  }, []);

  return state;
}
```

Performance Considerations

1. Code Splitting

```
// Dynamic imports for heavy components
import dynamic from 'next/dynamic';

const ChartComponent = dynamic(() => import('./ChartComponent'), {
  loading: () => <Skeleton />,
  ssr: false
});
```

2. Virtualization

For large lists:

```
import { useVirtualizer } from '@tanstack/react-virtual';

export function EventList({ events }: { events: CyberEvent[] }) {
  const parentRef = useRef<HTMLDivElement>(null);

  const virtualizer = useVirtualizer({
    count: events.length,
    getScrollElement: () => parentRef.current,
    estimateSize: () => 80,
    overscan: 5
  });

  return (
    <div ref={parentRef} style={{ height: '600px', overflow: 'auto' }}>
      <div style={{ height: `${virtualizer.getTotalSize()}px` }}>
        {virtualizer.getVirtualItems().map(virtualRow => (
          <EventCard
            key={virtualRow.key}
            event={events[virtualRow.index]}
            style={{
              position: 'absolute',
              top: 0,
              left: 0,
              width: '100%',
              transform: `translateY(${virtualRow.start}px)`
            }}
          />
        ))}
      </div>
    </div>
  );
}
```

3. Memoization

```
import { memo, useMemo } from 'react';

export const AgentCard = memo(({ agent }: { agent: Agent }) => {
  const statusColor = useMemo(() => {
    switch (agent.status) {
      case 'IDLE': return 'green';
      case 'BUSY': return 'yellow';
      case 'ERROR': return 'red';
      default: return 'gray';
    }
  }, [agent.status]);

  return <Card style={{ borderColor: statusColor }}>...</Card>;
});
```

4. Debouncing and Throttling

```
import { useDebouncedCallback } from 'use-debounce';

export function SearchInput() {
  const [search, setSearch] = useState('');

  const debouncedSearch = useDebouncedCallback(
    (value: string) => {
      // Perform search
      performSearch(value);
    },
    500
  );

  return (
    <input
      value={search}
      onChange={(e) => {
        setSearch(e.target.value);
        debouncedSearch(e.target.value);
      }}
    />
  );
}
```

Security Architecture

1. Authentication

```
// middleware.ts
import { NextResponse } from 'next/server';
import type { NextRequest } from 'next/server';

export function middleware(request: NextRequest) {
  const token = request.cookies.get('auth-token');

  if (!token && !request.nextUrl.pathname.startsWith('/login')) {
    return NextResponse.redirect(new URL('/login', request.url));
  }

  return NextResponse.next();
}
```

2. API Security

```
// lib/api-security.ts
export function validateAPIKey(request: NextRequest): boolean {
  const apiKey = request.headers.get('X-API-Key');
  return apiKey === process.env.API_SECRET_KEY;
}

export function rateLimiter(identifier: string): boolean {
  // Implement rate limiting logic
  const requests = getRequestCount(identifier);
  return requests < MAX_REQUESTS_PER_MINUTE;
}
```

3. Input Validation

```
import { z } from 'zod';

const AgentControlSchema = z.object({
  agentId: z.string().uuid(),
  action: z.enum(['start', 'stop', 'restart'])
});

export async function POST(request: NextRequest) {
  const body = await request.json();

  try {
    const validated = AgentControlSchema.parse(body);
    // Process validated data
  } catch (error) {
    return NextResponse.json({
      success: false,
      error: { code: 'INVALID_INPUT', message: 'Invalid request data' }
    }, { status: 400 });
  }
}
```

Scalability

Horizontal Scaling

The dashboard can be scaled horizontally by:

1. **Load Balancing**: Multiple Next.js instances behind a load balancer
2. **Stateless Design**: No server-side session state
3. **CDN Integration**: Static assets served from CDN
4. **Edge Functions**: API routes deployed to edge locations

Caching Strategy

```
// app/api/agents/route.ts
export async function GET() {
  const cached = await redis.get('agents');

  if (cached) {
    return NextResponse.json(JSON.parse(cached), {
      headers: {
        'Cache-Control': 'public, s-maxage=10, stale-while-revalidate=59'
      }
    });
  }

  const agents = await fetchAgents();
  await redis.set('agents', JSON.stringify(agents), 'EX', 60);

  return NextResponse.json(agents);
}
```

Database Optimization

If using Prisma:

```
// Efficient queries with select
const agents = await prisma.agent.findMany({
  select: {
    id: true,
    agentName: true,
    status: true,
    lastSeen: true
  },
  where: {
    status: { not: 'OFFLINE' }
  },
  orderBy: {
    lastSeen: 'desc'
  },
  take: 50
});

// Use indexes
model Agent {
  id        String   @id @default(uuid())
  agentId   String   @unique
  status    String   @index
  lastSeen  DateTime @index
}
```

This architecture provides a solid foundation for a scalable, performant, and maintainable cybersecurity dashboard application.