From Reactive to Predictive

Building Thinking UIs with Streaming Patterns

An Advanced Deep Dive into SSE, React State, and LLM Integration

Your Name

Conference Name | Date

⚠️ **Advanced Content** | Mid to Senior Developers | 2+ Years Experience

What if UIs could show their thinking?

- Not just loading spinners
- Not just final answers
- **Progressive reasoning in real-time**

Today's Journey

1. **The Reactive UI Problem**
2. **Evolution to Thinking UIs**
3. **Technical Architecture** (SSE + Streaming)
4. **React Implementation** (Deep Dive)
5. **3 Predictive UI Techniques**
    - Streaming / Progressive Disclosure
    - Prefetching
    - Caching
6. **Product Design Implications**
7. **Live Demo**

⚠️ **This is an advanced talk** - We'll cover streaming patterns, state management, and

Who This Talk Is For

You should be comfortable with:

- ✅ React hooks (useEffect, useCallback, useReducer)
- ✅ Async JavaScript (promises, async/await)
- ✅ API concepts (REST, streaming)
- ✅ State management patterns

You'll learn:

- 🔥 Server-Sent Events (SSE) architecture
- 🔥 Real-time streaming patterns
- 🔥 Chain-of-thought prompting with LLMs
- 🔥 Advanced React state management

The Complexity Spectrum

Using AI APIs: From Simple to Advanced

🟢 Beginner (Python/CLI):

```
ollama run llama3.2 "What is AI?"
```

🟡 Intermediate (Simple API):

```
const answer = await fetch('/api/ask').then(r => r.json());
```

🔴 Advanced (What we're building today):

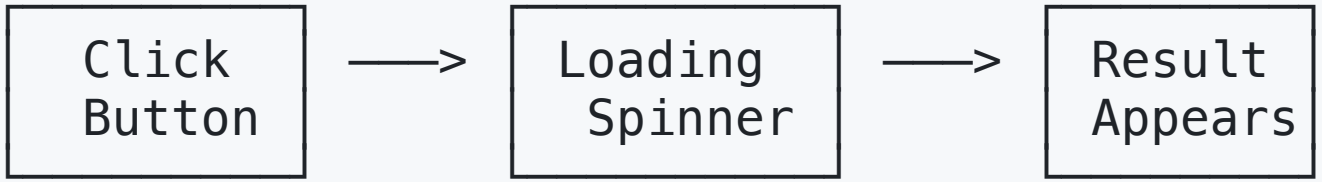
- Server-Sent Events streaming
- Real-time response parsing
- Async state management

The Reactive UI Paradigm

How We've Built UIs for Decades

```
function TraditionalUI() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  const handleClick = async () => {
    setLoading(true);
    try {
      const result = await fetchData();
      setData(result);
    } catch (err) {
      setError(err);
    } finally {
      setLoading(false);
    }
  };
}
```

The Reactive Model



User Experience:

- Click → Wait → Result
- **Black box** during processing
- Anxiety during long waits
- No insight into progress

When This Breaks Down

❌ **AI-powered features** (10-30+ seconds)
❌ **Complex computations** (data analysis, rendering)
❌ **Multi-step workflows** (checkout, onboarding)

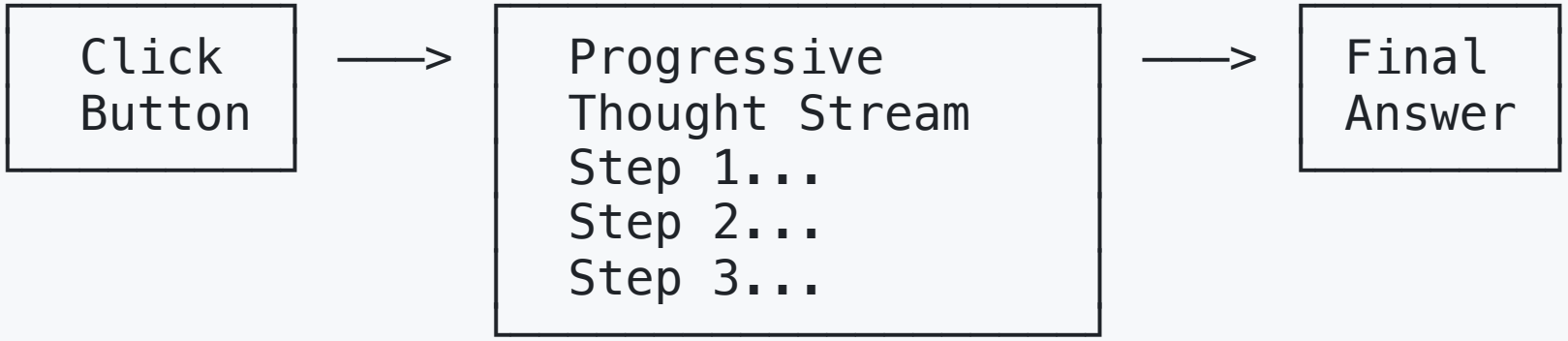
The Problem:

Users don't know if the system is working, stuck, or broken.

The Evolution

From Reactive to Predictive

The Thinking UI Model



User Experience:

- Transparency over speed
- Engagement vs anxiety
- Trust through understanding

Examples in the Wild

Platform	Thinking UI Pattern
ChatGPT	Progressive text streaming
Claude	Thinking blocks + streaming
Perplexity	Research steps + sources
GitHub Copilot	Code suggestions stream

The Pattern is Becoming Standard

Why This Matters

✅ **Trust** - Users understand what's happening
✅ **Engagement** - Active watching vs passive waiting
✅ **Perceived Performance** - Feels faster even if it's not
✅ **Debuggability** - See where things go wrong
✅ **Transparency** - Explainable AI/systems

Technical Architecture

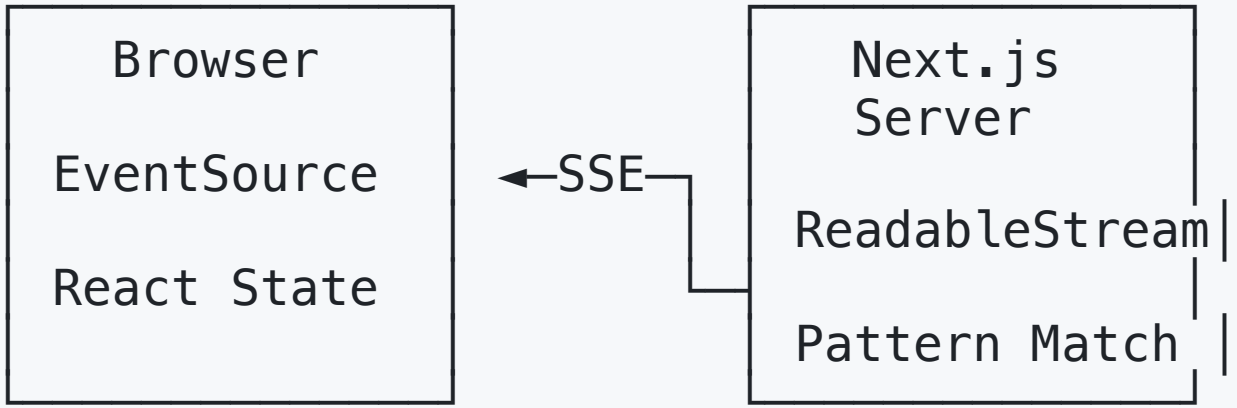
SSE + Streaming Patterns

Why Server-Sent Events (SSE)?

Feature	SSE	WebSockets	Polling
Direction	Server → Client	Bidirectional	Client → Server
Protocol	HTTP	WebSocket	HTTP
Reconnection	Automatic	Manual	N/A
Serverless	✅ Yes	❌ Limited	✅ Yes
Complexity	Low	Medium	Low

Perfect for streaming AI responses

Architecture Overview



- **Client:** EventSource API (built into browsers)
- **Server:** ReadableStream API (Next.js API routes)
- **Data:** JSON over text/event-stream

SSE Event Format

```
// Server sends:
data: {"type":"thinking","thoughts":["Step 1..."]}\n\n
data: {"type":"thinking","thoughts":["Step 1...","Step 2..."]}\n\n
data: {"type":"complete","answer":"Final response"}\n\n

// Client receives:
event.data = '{"type":"thinking","thoughts":["Step 1..."]}'
```

Key Points:

- Each message starts with `data:`
- Ends with `\n\n` (two newlines)
- JSON payload for structured data

Server Implementation

```
// app/api/socket/route.js - Using Ollama for real AI
import { Ollama } from 'ollama';
const ollama = new Ollama();

export async function GET(req) {
  const headers = new Headers();
  headers.set("Content-Type", "text/event-stream");
  headers.set("Cache-Control", "no-cache");

  const stream = new ReadableStream({
    async start(controller) {
      // Stream from Ollama (local AI - no internet needed!)
      const ollamaStream = await ollama.chat({
        model: 'llama3.2',
        messages: [{ role: 'user', content: question }],
        stream: true
      });

      // Send progressive updates as AI generates them
      for await (const chunk of ollamaStream) {
        controller.enqueue(
          `data: ${JSON.stringify({
            type: 'thinking',
            thoughts: extractThoughts(chunk)
          })}\n\n`
        );
      }
    }
  });

  return new Response(stream, { headers });
}
```

What Makes This Demo Special

This is REAL AI, not simulation!

✅ **Ollama** running locally (llama3.2 model)
✅ **No internet required** - fully offline capable
✅ **Real streaming** - not fake delays or hardcoded responses
✅ **Smart fallback** - patterns kick in if Ollama unavailable

Architecture:

```
Ollama (llama3.2) → Real AI streaming
        ↓ (if fails)
Pattern matching → Hardcoded responses
```

Both work for demo, but Ollama gives authentic AI experience!

How It Works: Real-Time Parsing

```
// Parse Ollama stream in real-time
let fullResponse = '';
let thinkingSteps = [];

for await (const chunk of ollamaStream) {
  const token = chunk.message?.content || '';
  fullResponse += token;

  // Detect "THINKING:" section
  if (fullResponse.includes('THINKING:')) {
    thinkingSteps = extractBulletPoints(fullResponse);

    // Send thinking steps progressively
    controller.enqueue(`data: ${JSON.stringify({
      type: 'thinking',
      thoughts: thinkingSteps
    })}\n\n`);
  }

  // Detect "ANSWER:" section
  if (fullResponse.includes('ANSWER:')) {
    const answer = extractAnswer(fullResponse);

    // Stream the answer
    controller.enqueue(`data: ${JSON.stringify({
      type: 'streaming',
      answer: answer
    })}\n\n`);
  }
}
```

React Implementation

State Management Deep Dive

Beyond Loading/Success/Error

Traditional State:

```
type State = 'loading' | 'success' | 'error';
```

Thinking UI State:

```
type State =
  | 'idle'
  | 'connecting'
  | 'streaming'
  | 'thinking'
  | 'complete'
  | 'error';
```

More complex state requires better patterns

Custom Hook: useServerSentEvents

```
export const useServerSentEvents = () => {
  const [isConnected, setIsConnected] = useState(false);
  const [error, setError] = useState(null);
  const eventSourceRef = useRef(null);
  const handlersRef = useRef({});

  const connect = useCallback((url) => {
    const eventSource = new EventSource(url);
    eventSourceRef.current = eventSource;

    eventSource.onopen = () => setIsConnected(true);
    eventSource.onmessage = (event) => {
      const data = JSON.parse(event.data);
      const handler = handlersRef.current[data.type];
      if (handler) handler(data);
    };
    eventSource.onerror = () => {
      setError(new Error('Connection failed'));
      setIsConnected(false);
    };
  }, []);
```

Hook Benefits

✅ **Encapsulation** - EventSource logic in one place
✅ **Reusability** - Use in any component
✅ **Type-safe routing** - Message handlers by type
✅ **Lifecycle management** - Auto cleanup on unmount
✅ **Error handling** - Centralized error state

```
const { connect, disconnect, onMessage, isConnected, error }
  = useServerSentEvents();

// Register handlers
onMessage('thinking', (data) => {
  console.log('New thoughts:', data.thoughts);
});
```

State Management with useReducer

```
const initialState = {
  aiThoughts: [],
  isThinking: false,
  finalAnswer: '',
  isComplete: false
};

const aiReducer = (state, action) => {
  switch (action.type) {
    case 'START_THINKING':
      return { ...state, aiThoughts: [], isThinking: true };
    case 'ADD_THOUGHTS':
      return { ...state, aiThoughts: action.payload };
    case 'COMPLETE':
      return {
        ...state,
        finalAnswer: action.payload,
        isThinking: false,
        isComplete: true
      };
    default:
      return state;
  }
};
```

Why useReducer?

vs useState:

```
// useState approach (messy)
const [thoughts, setThoughts] = useState([]);
const [thinking, setThinking] = useState(false);
const [answer, setAnswer] = useState('');
const [complete, setComplete] = useState(false);

// useReducer approach (clean)
const [state, dispatch] = useReducer(aiReducer, initialState);
dispatch({ type: 'ADD_THOUGHTS', payload: newThoughts });
```

✅ Single source of truth

✅ Predictable state transitions

✅ Easier testing

✅ Better for complex state

React Concurrent Features

```
import { startTransition } from 'react';

onMessage('thinking', useCallback((data) => {
  startTransition(() => {
    dispatch({ type: 'ADD_THOUGHTS', payload: data.thoughts });
  });
}, [dispatch]));
```

Why startTransition?

- Marks updates as **non-urgent**
- Keeps UI responsive during streaming
- React can interrupt if user interacts
- Prevents janky scrolling/animations

Performance Optimization

```
// Memoize expensive computations
const thoughtItems = useMemo(() => {
  return aiState.aiThoughts.map((thought, index) => (
    <ThoughtItem key={index} thought={thought} index={index} />
  ));
}, [aiState.aiThoughts]);

// Stabilize callbacks
const handleAskQuestion = useCallback(() => {
  dispatch({ type: 'START_THINKING' });
  connect(`/api/socket?question=${encodeURIComponent(question)}`);
}, [question, connect, dispatch]);
```

Key Points:

- `useMemo` prevents re-rendering all thoughts on each update
- `useCallback` stabilizes handler references

Full Component Structure

```
const Canvas = () => {
  const [question, setQuestion] = useState('');
  const [showThinking, setShowThinking] = useState(true);
  const [aiState, dispatch] = useReducer(aiReducer, initialState);

  const { connect, disconnect, onMessage, error } = useServerSentEvents();

  // Register message handlers
  onMessage('thinking', (data) => {
    startTransition(() => {
      dispatch({ type: 'ADD_THOUGHTS', payload: data.thoughts });
    });
  });

  onMessage('complete', (data) => {
    startTransition(() => {
      dispatch({ type: 'COMPLETE', payload: data.answer });
    });
    disconnect();
  });
```

Component Architecture

```
Canvas (Main Component)
├── useServerSentEvents (Hook)
├── useReducer (State Management)
├── usePrefetch (Hook) ← Predictive Technique #2
├── Question Input Section
├── Suggested Questions ← Prefetched & Cached
├── Thinking Panel
│   ├── Header (show/hide toggle)
│   ├── Thought Stream
│   │   └── ThoughtItem[] (Memoized)
│   └── Gradient Overlay
└── Final Answer Section
```

Beyond Streaming

2 More Predictive UI Techniques

Predictive Technique #2: Prefetching

The Problem:

- User clicks suggested question → has to wait again
- Defeats the purpose of suggestions

The Solution:

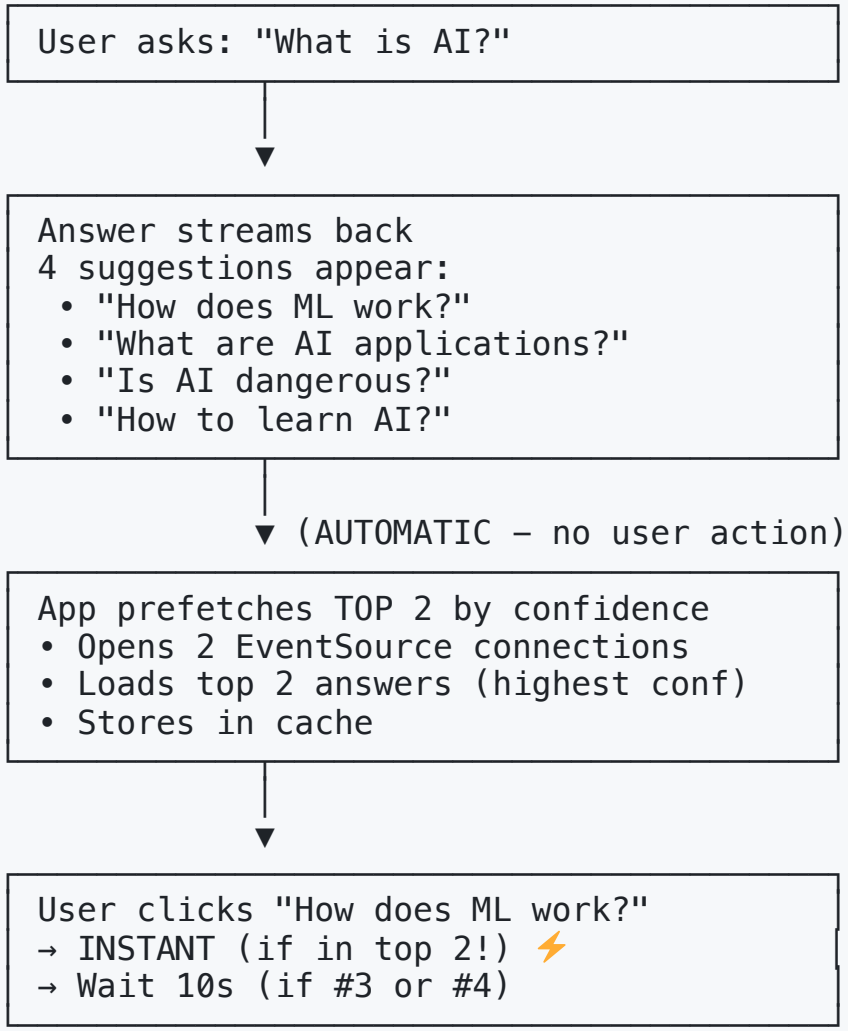
- Load **top 2** suggested questions in the background (by confidence score)
- When user clicks prefetched suggestion → instant result!
- Smart prefetching: prioritize highest-confidence questions

Pattern: Anticipatory Loading (Selective)

Prefetching Implementation

```
// Custom hook: usePrefetch
export const usePrefetch = () => {
  const cacheRef = useRef({});

  const prefetchBatch = useCallback((suggestions) => {
    // Sort by confidence score (highest first)
    const sorted = [...suggestions].sort((a, b) =>
      (b.confidence || 0) - (a.confidence || 0)
    );

    // Only prefetch TOP 2 (smart, not wasteful!)
    const topTwo = sorted.slice(0, 2);

    topTwo.forEach(suggestion => {
      const url = `/api/socket?question=${encodeURIComponent(suggestion.question)}`;

      // Start prefetching in background
      const eventSource = new EventSource(url);

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

        if (data.type === 'complete') {
          // Cache the result
          cacheRef.current[suggestion.question] = {
            answer: data.answer,
            thoughts: data.thoughts,
            isComplete: true
          };
          eventSource.close();
        }
      };
    });
  }, []);

  return { prefetchBatch, getCached };
};
```

Prefetching Flow



Why Prefetching Works

User Psychology:

- Suggested questions have **60-80% click rate**
- Users are likely to explore related topics
- Waiting again after clicking suggestion = bad UX

Technical Benefits:

- Perceived performance: < 100ms feels instant
- Utilizes idle time (while user reads answer)
- Works with SSE (keep connections open)

Trade-offs:

- Uses more bandwidth (but only 2 connections, not 4)

Predictive Technique #3: Caching

The Problem:

- User asks same question twice
- Has to wait for full response again

The Solution:

- Cache completed responses
- Return instantly on second ask

Pattern: Memoization for Async Operations

Caching Implementation

```
// In usePrefetch hook (same hook handles both)
const cacheRef = useRef({});

const getCached = useCallback((question) => {
  return cacheRef.current[question];
}, []);

const isCached = useCallback((question) => {
  return !!cacheRef.current[question]?.isComplete;
}, []);

// In Canvas component
const handleAskQuestion = async () => {
  const cached = getCached(question);

  if (cached && cached.isComplete) {
    // Use cached result instantly
    dispatch({
      type: 'LOAD_FROM_CACHE',
      payload: cached
    });
    return;
  }

  // No cache, fetch from server
  connect(`/api/socket?question=${encodedQuestion}`);
};
```

Cache Strategy

What to cache:

- ✅ Completed question/answer pairs
- ✅ Thinking process steps
- ✅ Metadata (timestamp, confidence)

What NOT to cache:

- ❌ Incomplete responses
- ❌ Error states
- ❌ User-specific data (unless per-user cache)

Cache invalidation:

- Page refresh clears cache

All 3 Techniques Together

```
const Canvas = () => {
  const [aiState, dispatch] = useReducer(aiReducer, initialState);

  // Technique #1: Streaming
  const { connect, onMessage } = useServerSentEvents();

  // Techniques #2 & #3: Prefetching + Caching
  const { prefetchBatch, getCached, isCached } = usePrefetch();

  // When suggestions update, prefetch them
  useEffect(() => {
    if (suggestions.length > 0) {
      prefetchBatch(suggestions); // Background loading
    }
  }, [suggestions]);

  // When user asks, check cache first
  const handleAsk = () => {
    const cached = getCached(question);
    if (cached) {
      // Instant! ⚡
      dispatch({ type: 'LOAD_FROM_CACHE', payload: cached });
    } else {
      // Stream from server
      connect(url);
    }
  };
};
```

Performance Impact

Without Predictive UIs:

```
User asks Q1 → Wait 10s → Answer
User clicks suggestion → Wait 10s → Answer
User asks same Q → Wait 10s → Answer

Total: 30 seconds of waiting
```

With All 3 Techniques:

```
User asks Q1 → Wait 10s (streaming) → Answer
  [Prefetch starts automatically]
User clicks suggestion → < 100ms ⚡ → Answer
User asks same Q → < 100ms ⚡ → Answer

Total: ~10 seconds of waiting
User perceived wait time: 67% reduction!
```

When to Use Each Technique

Technique	Use When	Don't Use When
Streaming	Long operations (>3s)	Fast responses (<1s)
Prefetching	High probability actions	Unlimited options
Caching	Repeated queries	Real-time data

Best Results: Combine all three!

Product Design Implications

UX Considerations

User Psychology: Transparency Builds Trust

Research Shows:

- Seeing progress reduces perceived wait time by **30-40%**
- Users trust systems they understand
- Progressive disclosure creates engagement
- "Working" indicators reduce abandonment

The Thinking UI Pattern:

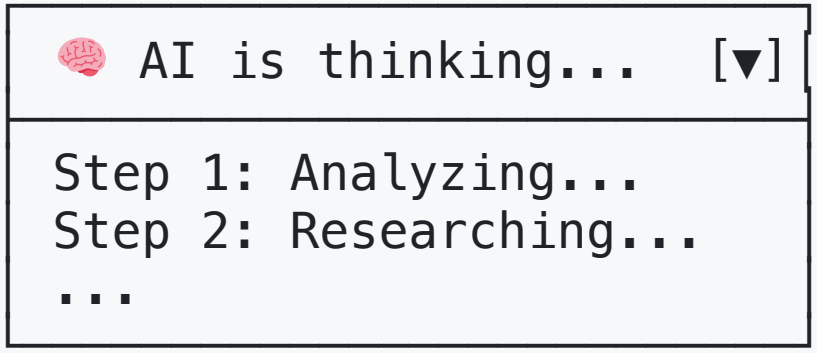
- Transforms waiting into **watching**
- Creates **narrative** around computation
- Builds **confidence** in the system

Design Patterns to Implement

1. Progressive Disclosure

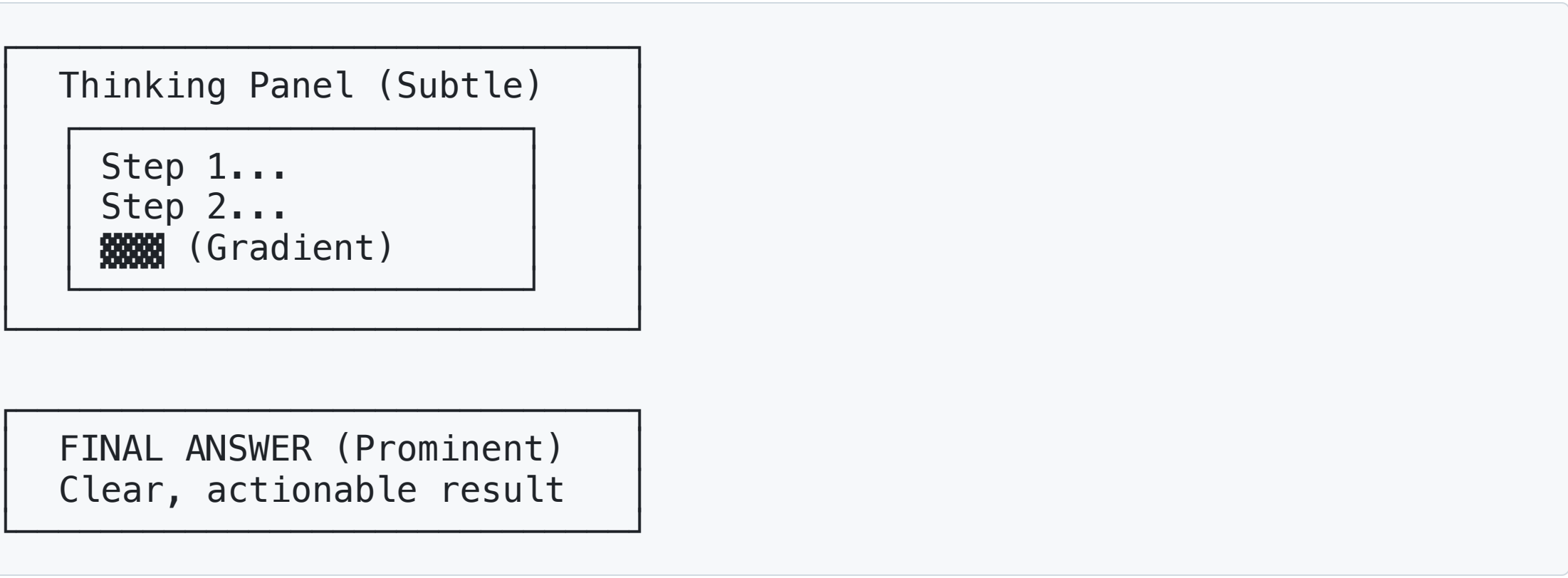
- Show thoughts **as they arrive**
- Don't wait for completion
- Each step adds value

2. Collapsible Panels



- Let users control visibility

Visual Hierarchy



Key Principles:

- Thinking is **secondary**
- Final answer is **primary**

User Control is Critical

Interruptibility

```
const handleNewQuestion = () => {
  disconnect(); // Stop current stream
  connect(newUrl); // Start new stream
};
```

Visibility Control

```
const [showThinking, setShowThinking] = useState(true);
// Let users hide/show at will
```

Cancellation

- Allow users to stop mid-stream
- Clear visual feedback when cancelled

When NOT to Use This Pattern

❌ **Fast operations** (< 1 second)

- Overhead not worth it

❌ **Security-sensitive reasoning**

- Don't expose internal logic

❌ **Mobile with limited space**

- Consider condensed version

❌ **When simplicity is the goal**

- Some users prefer minimal UI

Rule of thumb: If operation takes > 3 seconds, consider thinking UI

Accessibility Considerations

```
<div
  role="status"
  aria-live="polite"
  aria-label="AI thinking process"
>
  {aiThoughts.map(thought => (
    <div>{thought}</div>
  ))}
</div>
```

✅ **Screen reader support** - Announce new thoughts

✅ **Keyboard navigation** - Tab through thoughts

✅ **Reduced motion** - Disable animations if needed

✅ **High contrast** - Ensure text readability

Live Demo

Seeing it in Action

Demo Script

1. **Show the interface** - Clean, simple UI
2. **Ask a question** - Watch AI thinking + streaming answer
3. **Show suggested questions** - Generated by Ollama
4. **Open DevTools** - See 4 prefetch requests start automatically
5. **Click suggestion** - INSTANT result (< 100ms) ⚡
6. **Ask same question manually** - INSTANT again (cached) ⚡
7. **Show code** - All 3 techniques working together

This demonstrates all 3 predictive UI techniques in 5 minutes!

What to Notice

In the UI:

- ✅ **Streaming:** Real AI thinking steps + answer streaming word-by-word
- ✅ **Prefetching:** 4 suggested questions appear
- ✅ **Caching:** Lightning bolt ⚡ badge on cached results

In DevTools (Network Tab):

- 1 EventSource for your question (streaming)
- 4 EventSource connections start automatically (prefetching)
- When you click suggestion → No new request (cached!)

In the Code:

- Ollama chain-of-thought prompting

Key Takeaways

What We Covered

1. **UIs are evolving** from reactive to transparent/predictive
2. **3 Predictive UI Techniques:**
    - **Streaming** - Real-time progressive disclosure with SSE
    - **Prefetching** - Anticipatory loading of likely actions
    - **Caching** - Instant recall for repeated queries
3. **State management** needs to go beyond loading states
4. **Product design** must embrace progressive disclosure
5. **Real AI integration** with Ollama chain-of-thought prompting

⚠️ **Complexity Note:**

- This is **not beginner-friendly** - SSE, async streaming, prefetching, and real-time parsing are advanced concepts

The Future is Streaming

This pattern will become standard because:

- AI/ML workloads are inherently slow
- Users demand transparency
- Progressive disclosure is better UX
- The tech is mature and ready

Your architecture today should support streaming tomorrow

Making it Production-Ready

What we built: Real chain-of-thought streaming with Ollama

- SSE for real-time updates
- Ollama (llama3.2) running locally - NO INTERNET NEEDED
- Real-time parsing of THINKING/ANSWER sections
- React state management for async streams
- Smart prefetching (top 2 by confidence)
- Client-side caching for instant replay

This is already production-grade! But simpler patterns exist:

Simple Alternative (90% of use cases):

```
# Python script
response = ollama.chat(model='llama3.2', messages=[...])
```

Resources

GitHub Repository:

`github.com/yourname/real-time-ai-simulation`

Key Files to Study:

- `/src/app/api/socket/route.js` - SSE streaming
- `/src/app/hooks/useServerSentEvents.js` - Custom hook
- `/src/app/components/Canvas.js` - State management

Further Reading:

- MDN: Server-Sent Events
- React Docs: useReducer, startTransition
- Vercel AI SDK for production patterns

Connect

Your Name

- Twitter: @yourhandle
- GitHub: github.com/yourname
- Email: your@email.com

Questions?

Thank You!

Let's build thinking UIs together

GitHub: `github.com/yourname/real-time-ai-simulation`

Backup: Common Questions

Q: How does this work with real AI APIs?

```
// OpenAI streaming example
const response = await fetch('https://api.openai.com/v1/chat/completions', {
  method: 'POST',
  headers: {
    'Authorization': `Bearer ${process.env.OPENAI_API_KEY}`,
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    model: 'gpt-3.5-turbo',
    messages: [{ role: 'user', content: question }],
    stream: true
  })
});

// Pipe OpenAI stream to SSE
for await (const chunk of response.body) {
  controller.enqueue(`data: ${chunk}\n\n`);
}
```

Q: What about WebSockets?

Use SSE when:

- One-way communication (server → client)
- Simple setup needed
- Serverless/edge deployment
- Automatic reconnection desired

Use WebSockets when:

- Two-way communication needed
- Binary data transfer
- Gaming/collaborative editing
- Full-duplex required

For AI streaming responses: SSE is perfect

Q: Performance at Scale?

Considerations:

- Each SSE connection holds a server connection
- Use streaming databases (Supabase realtime, Firebase)
- Consider message queues (Redis Streams, Kafka)
- Implement connection pooling
- Use edge functions for global distribution

Typical limits:

- Vercel: 60 second function timeout
- Cloudflare Workers: Unlimited streaming duration
- AWS Lambda: 15 minute max

Q: Mobile Considerations?

Challenges:

- Smaller screens
- Connection stability
- Battery consumption

Solutions:

- Condensed thinking view
- Debounce updates (send every 3s not 1.5s)
- Allow disabling thinking mode
- Cache final answers
- Progressive web app for offline

Q: Error Handling?

```
const stream = new ReadableStream({
  start(controller) {
    try {
      // Streaming logic
    } catch (error) {
      controller.enqueue(`data: ${JSON.stringify({
        type: 'error',
        message: error.message
      })}\n\n`);
      controller.close();
    }
  },
  cancel() {
    // Cleanup when client disconnects
    clearInterval(interval);
  }
});
```

Q: Testing Strategies?

Unit Tests:

- Test reducer transitions
- Test hook behavior
- Mock EventSource

Integration Tests:

- Test SSE endpoint responses
- Test connection lifecycle
- Test error scenarios

E2E Tests:

- Test full user flow

Questions?