Introduction to AI SDK and Ink

Building Web and Terminal Apps with TypeScript

15 May 2026

Yong Cao, Phastar

phastar

Quick Intro to React and Ink

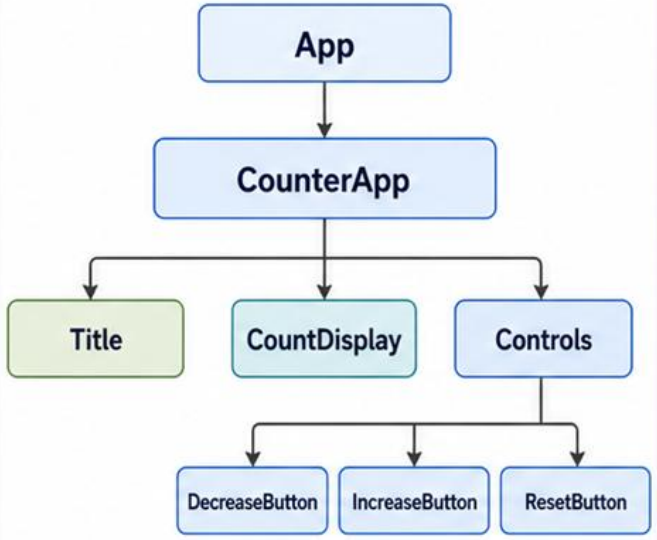
No Frontend Knowledge Assumed

phastar

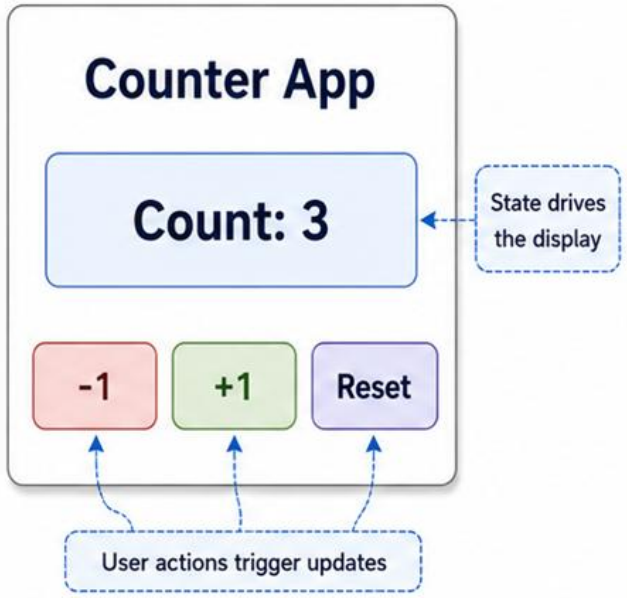
React/Ink Through A SAS Lens

1. React Classic Example: Counter App

Component Structure



Page Preview



Component Pseudocode

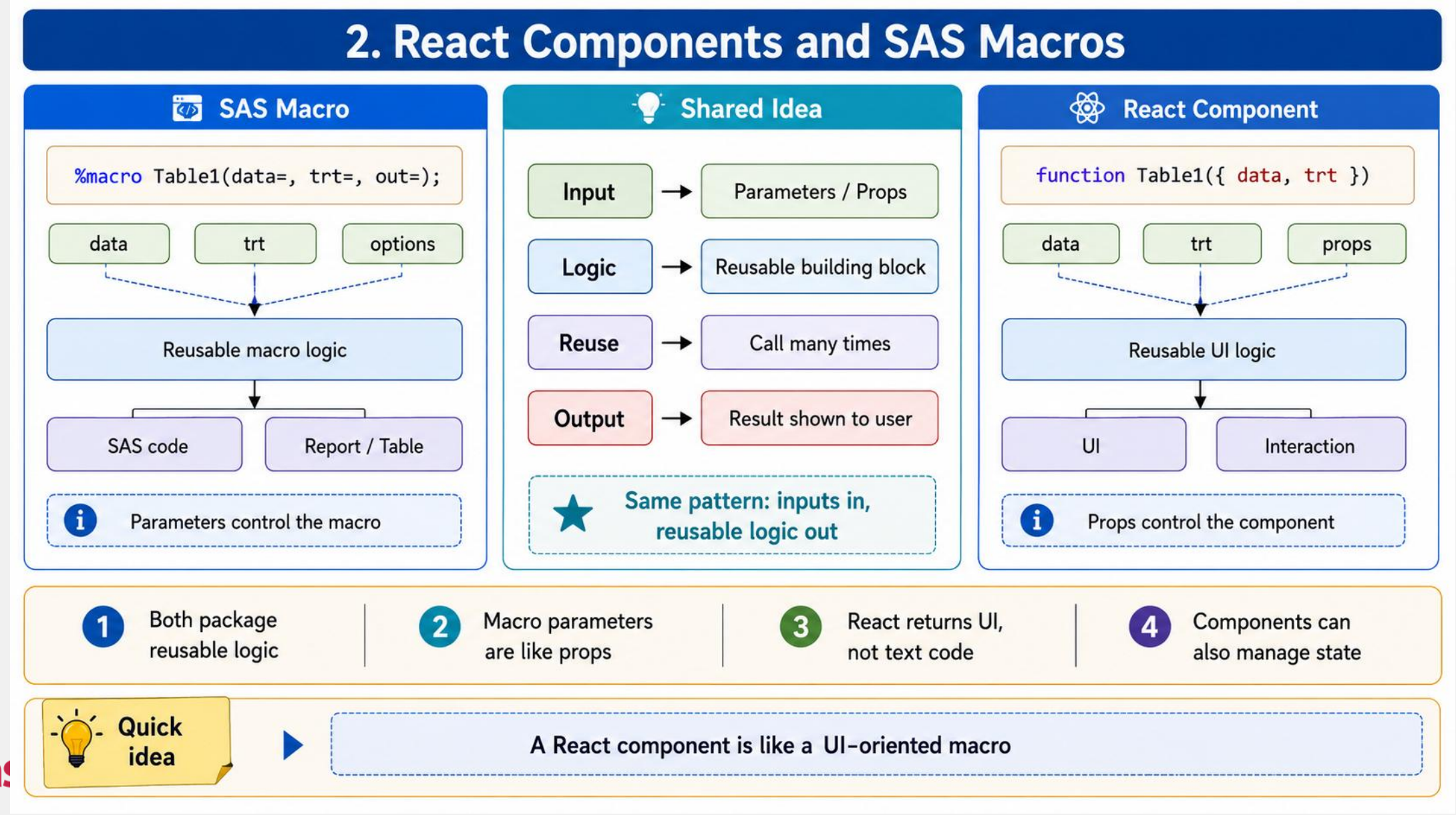
```
function CounterApp() {
  const [count, setCount] = useState(0)
  return (
    <Card>
      <Title />
      <CountDisplay count={count} />
      <Controls
        onInc={() => setCount(count + 1)}
        onDec={() => setCount(count - 1)}
        onReset={() => setCount(0)}
      />
    </Card>
  )
}
```

Quick idea

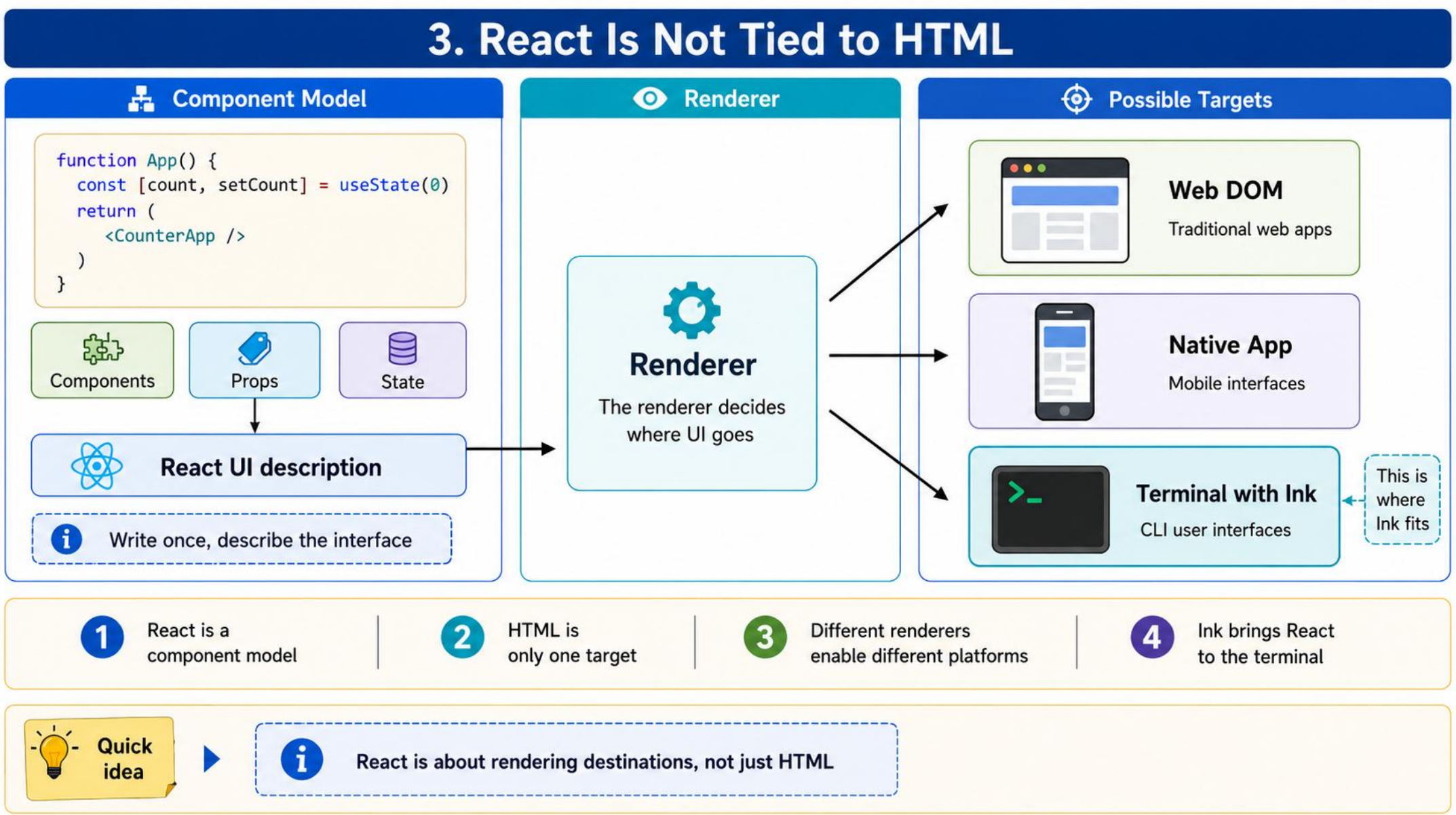
React = components + state + events

1. A component is a reusable UI building block
2. State stores changing values
3. Props pass data and behavior
4. User actions update the UI

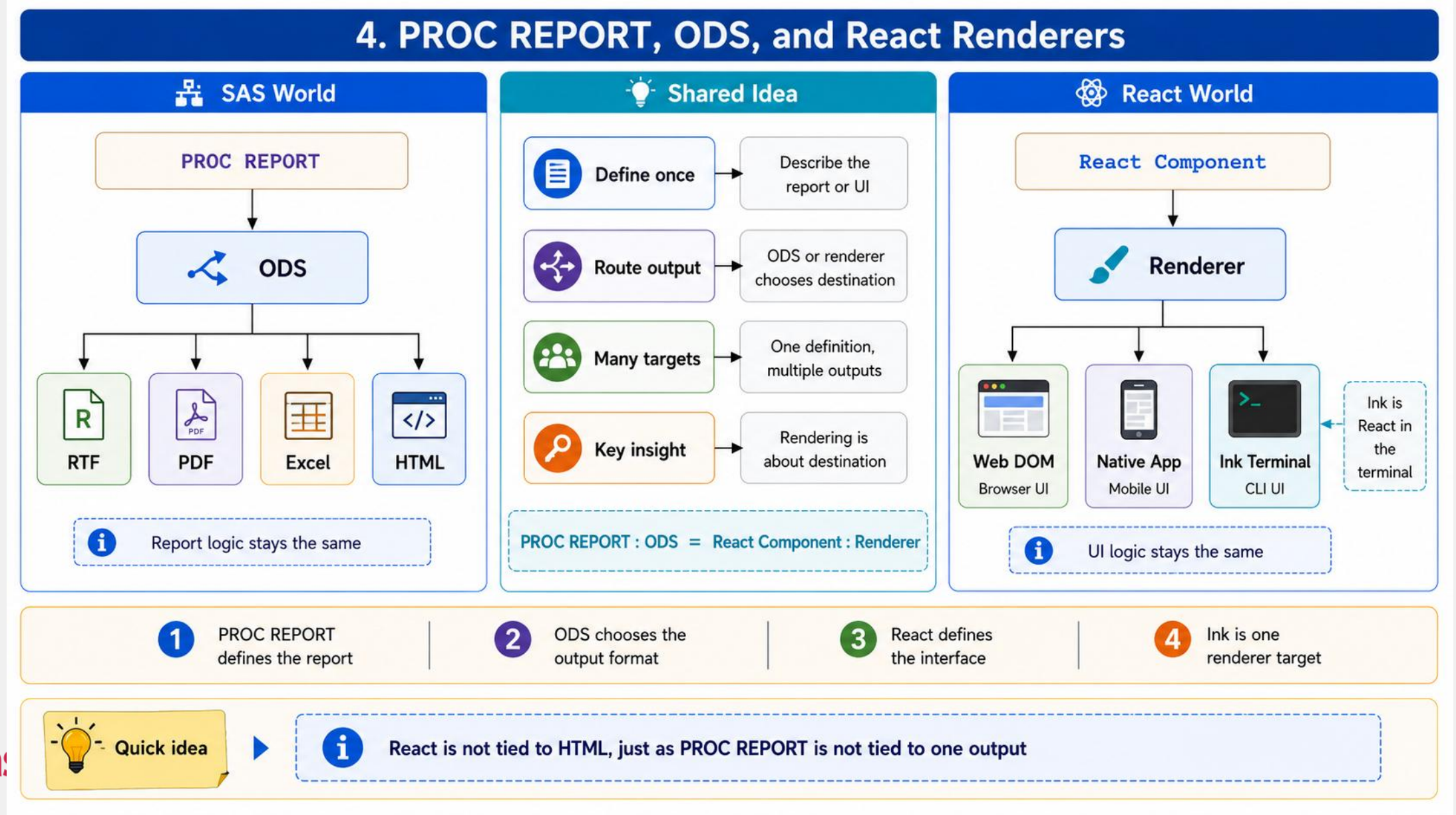
React/Ink Through A SAS Lens



React/Ink Through A SAS Lens



React/Ink Through A SAS Lens



React/Ink Through A SAS Lens

Ink capabilities: actual terminal UI + core API

A practical bridge between React, AI agents, and statistical-programming workflows

</> Core API surface

React components → Ink renderer → Terminal frame → Keyboard input → State update

Components

- **<Text>** — Style text, colors, bold, wrapping
- **<Box>** — Flexbox-style layout, padding, borders
- **<Static>** — Keep completed output visible
- **<Transform>** — Post-process rendered lines

Input hooks

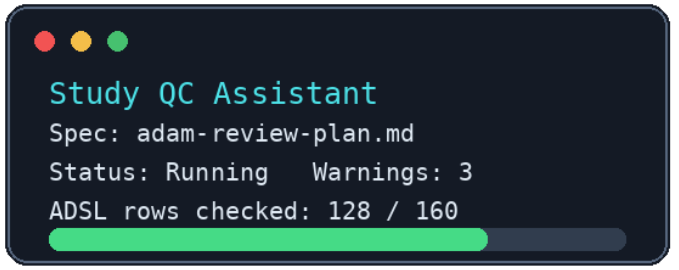
- **useInput()** — Typed chars, arrows, Enter, q
- **useFocus()** — Focus-aware controls
- **useFocusManager()** — Move focus between inputs
- **usePaste()** — Handle pasted text as one event

Render lifecycle

- **render()** — Mount React tree to terminal
- **useApp()** — exit(), waitUntilRenderFlush()
- **rerender()** — Update the root component
- **unmount()** — Clean up terminal state

Statistical programming fit: file-based, status-heavy, keyboard-friendly, and easy to run beside SAS/R/Python files and logs.

What it looks like



Minimal Ink code

Box · Text · useInput · useApp · render

```
function App() {
  const {exit} = useApp();
  useInput((input, key) => {
    if (input === 'q') exit();
    if (key.return) runQcStep();
  });

  return <Box borderStyle="round">
    <Text color="cyan">Study QC Assistant</Text>
  </Box>;
}
render(<App />);
```

Key idea Ink gives AI SDK / tool-calling workflows a real terminal front-end: status, input, approvals, and re-rendering.

- **Live dashboards** — Progress, warnings, status and streamed output
- **Interactive input** — Menus, forms, confirmations and keyboard shortcuts
- **Local workflow** — Read specs, SAS logs, datasets and generated files
- **Agent approvals** — Show tool intent before writing files or running actions

Why Terminal App?

The terminal is a good home for internal tools

Text-first workflows, direct file access, and rapid internal tool development

Direct Access to Study Assets

Work directly with specs, SAS datasets, logs, metadata, and existing CLI tools on local or server files.

- No upload / download loop
- No browser or CORS friction
- Fits file-based study workflows

Focused, Text-First UI

Many statistical programming tasks are table-heavy, status-heavy, and workflow-oriented. Terminal UI is often enough.

- Great for tables and status
- Fewer layout decisions
- Focus on function, not styling

Fast Iteration for Internal Tools

Prototype utilities quickly, test them on real study folders, and share them with colleagues without building a full web app.

- Idea → prototype quickly
- Prototype → teammate feedback
- Same-day iteration

Typical use cases

- Spec-to-code assistants
- QC and validation helpers
- Log and output reviewers
- Metadata and study utilities

Example workflow

A terminal **agent** reads a **local spec**, generates a SAS program beside it, and reports progress live — no network required.

phastar

AI SDK

phastar

AI SDK Introduction

AI SDK by Vercel

AI SDK: TypeScript toolkit for AI applications

A unified way to build AI features across models, tools, and interfaces

What it is

An open-source TypeScript library from Vercel for building AI-powered applications and agents.

Open source | TypeScript-first | Provider-agnostic

Works with

- React / Next.js
- Vue / Svelte
- Node.js / server
- Custom clients and CLIs

Core problem

LLM integration can become provider-specific and repetitive. AI SDK standardizes the plumbing so teams can focus on the workflow.

- Call many model providers through one API
- Stream responses without custom protocol work
- Define tools with typed schemas
- Validate structured outputs for downstream code
- Keep application logic separate from provider details

Where it fits

Use it when AI behavior needs to become part of an application, tool, or operational workflow.

- Chatbots and copilots
- AI agents with tools
- Internal workflow tools
- RAG and knowledge apps
- Structured data extraction
- Web, server, and CLI apps

Quick idea

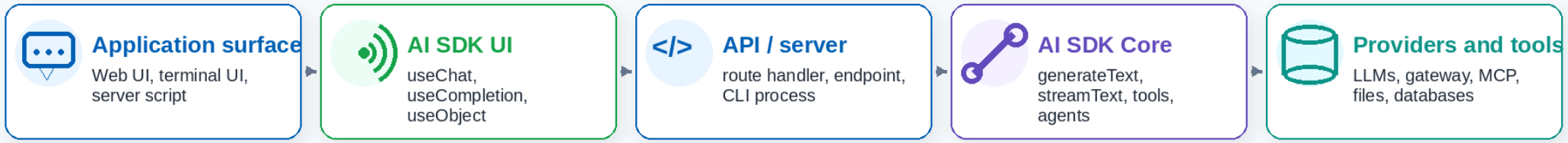
AI SDK separates workflow logic from provider plumbing, streaming protocols, and tool orchestration.

AI SDK Architecture

AI SDK architecture

How AI SDK UI and AI SDK Core fit together

AI SDK UI manages product interactions; AI SDK Core talks to models, tools, and providers



AI SDK Core (package: ai)

Low-level, provider-agnostic primitives for model calls in TypeScript environments.

- **Text APIs** — generateText, streamText
- **Structured output** — schema-valid data
- **Tool calling** — tool(), dynamic tools, MCP
- **Agents** — multi-step loops with stop conditions
- **Providers** — OpenAI, Anthropic, Google, custom
- **Testing** — mock providers and helpers

AI SDK UI (@ai-sdk/*)

Front-end hooks and stream-aware abstractions for building chat, completion, and assistant interfaces.

- **Hooks** — useChat, useCompletion, useObject
- **Managed state** — messages, status, error, tool parts
- **Streaming UI** — real-time updates as data arrives
- **Transport** — custom endpoints and protocols
- **Approval UX** — surface sensitive tool requests
- **Frameworks** — React, Svelte, Vue support

Mental model

AI SDK Core is the engine room. AI SDK UI is the cockpit. The same model call can feed a web chat, a dashboard, an API, or a terminal app.

Core concepts – text generation and tool calling

AI SDK Core concepts

Generation, streaming, tools, and approval

The minimum mental model for building assistants that can act safely

Text generation APIs

Use the API shape that matches the user experience.

generateText

Returns the complete result. Good for automation, summaries, and batch work.

streamText

Streams tokens, tool events, and progress. Good for chat and live interfaces.

Tool calling

A tool is a typed function the model may call when it needs external information or an action.

User request → Model chooses → Schema validates → Tool runs → Result returns

- Tools can read data, call APIs, run code, or write files
- Tool output can be fed back into the model for the final answer

How to define a tool

Give the model a name, a clear description, a typed input schema, and an execute function.

```
const runSas = tool({
  description: "Run a SAS program",
  inputSchema: z.object({
    path: z.string(),
    dryRun: z.boolean()
  }),
  execute: async input => runSasProgram(input)
})
```

Tool anatomy

- description
- inputSchema
- execute
- typed result

How to allow user approval

Require confirmation before sensitive tools run, such as file writes, shell commands, job triggers, or private data access.

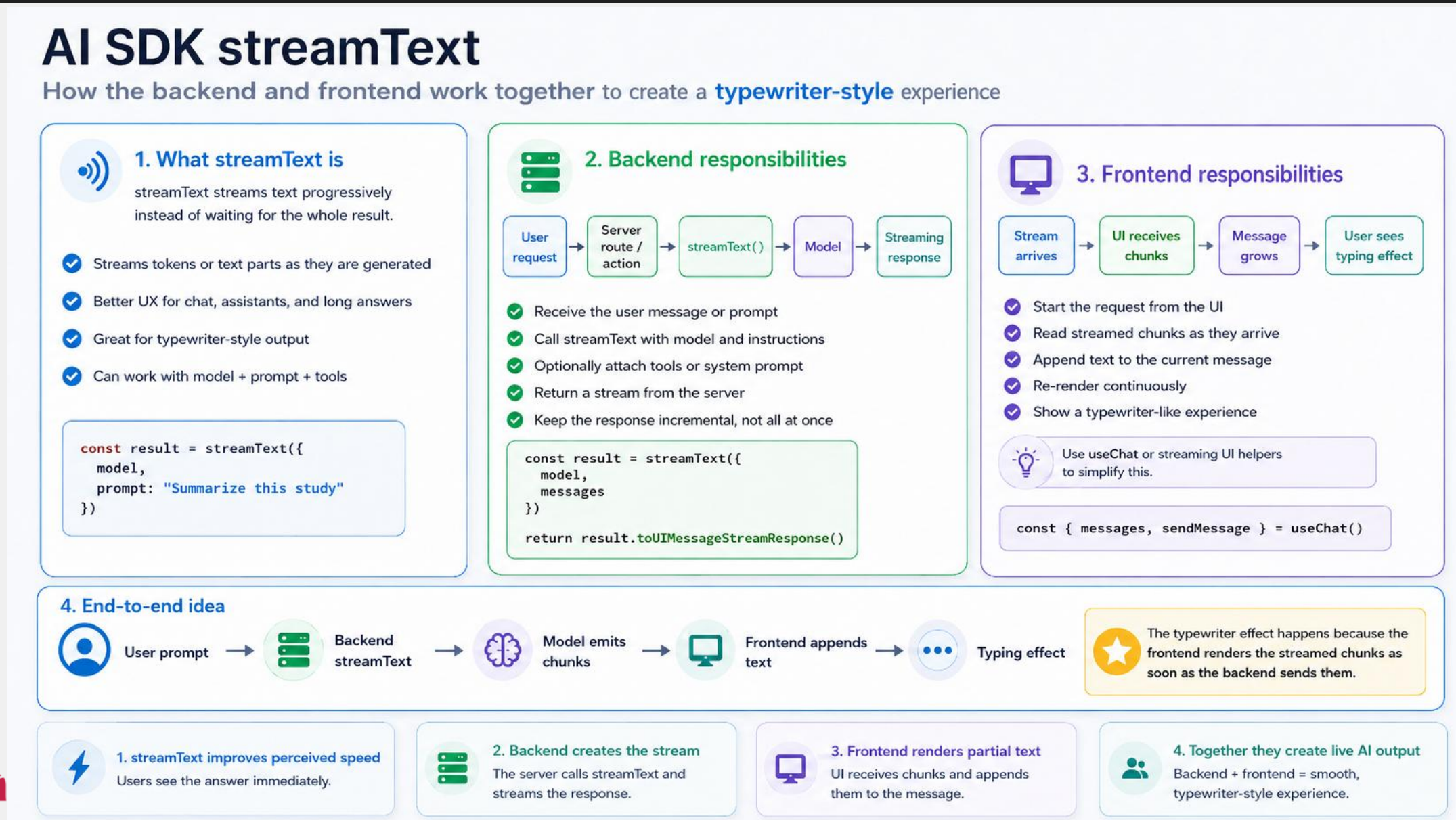
Tool call needs approval → Show inputs to user → Approve or deny → Execute or cancel

```
const writeFile = tool({
  description: "Write a .sas file",
  inputSchema,
  needsApproval: true,
  execute: async input => writeFile(input)
})
```

UI responds: approve or deny

e.g. addToolApprovalResponse(...)

A closer look at stream text



A closer look at tool calling

AI SDK Tool Calling

What a tool is, how to register it, how approval works, and how the model decides to use it

What is ia tool?

A tool is a structured function the model can call to do something beyond plain text generation.

- Has a name, description, input schema, and execute logic
- Lets the model read data or trigger actions
- Tool results become extra context for the final answer
- Common examples: read a file, query data, write output

Typical tool parts

Name	unique identifier
Description	helps the model know when to use it
Input schema	expected parameters
needsApproval	require user approval before execution
execute(...)	runs the tool and returns a result

How to register tools

Define tools

```
const readSpec = tool({
  description: "Read an ADaM spec file",
  inputSchema: z.object({ path: z.string() }),
  needsApproval: false,
  execute: async ({ path }) => { ... }
})

const generateReview = tool({
  description: "Generate review comments file",
  inputSchema: z.object({ specPath: z.string() }),
  needsApproval: true,
  execute: async ({ specPath }) => { ... }
})

const writeSasProgram = tool({
  description: "Write a draft SAS program",
  inputSchema: z.object({ outPath: z.string() }),
  needsApproval: true,
  execute: async ({ outPath }) => { ... }
})
```

Pass tools to the model call

```
const result = streamText({
  model,
  messages,
  tools: {
    readSpec,
    generateReview,
    writeSasProgram
})
```

- Define each tool with schema + execute
- Pass tools as a dictionary in tools: { ... }
- Descriptions and schemas guide tool selection
- Use needsApproval for sensitive actions

How the model uses tools

1. User asks a question
2. Model reads the request
3. Model inspects available tools
4. Model decides whether a tool can help
5. If selected tool needs approval, app asks user first
6. Approved tool executes
7. Tool result returns
8. Model continues and answers the user

- The model does not use every tool every time
- It chooses based on the user request + tool descriptions
- Better names, descriptions, and schemas improve selection
- Approval is ideal for file writes, external actions, or sensitive access

Approval step

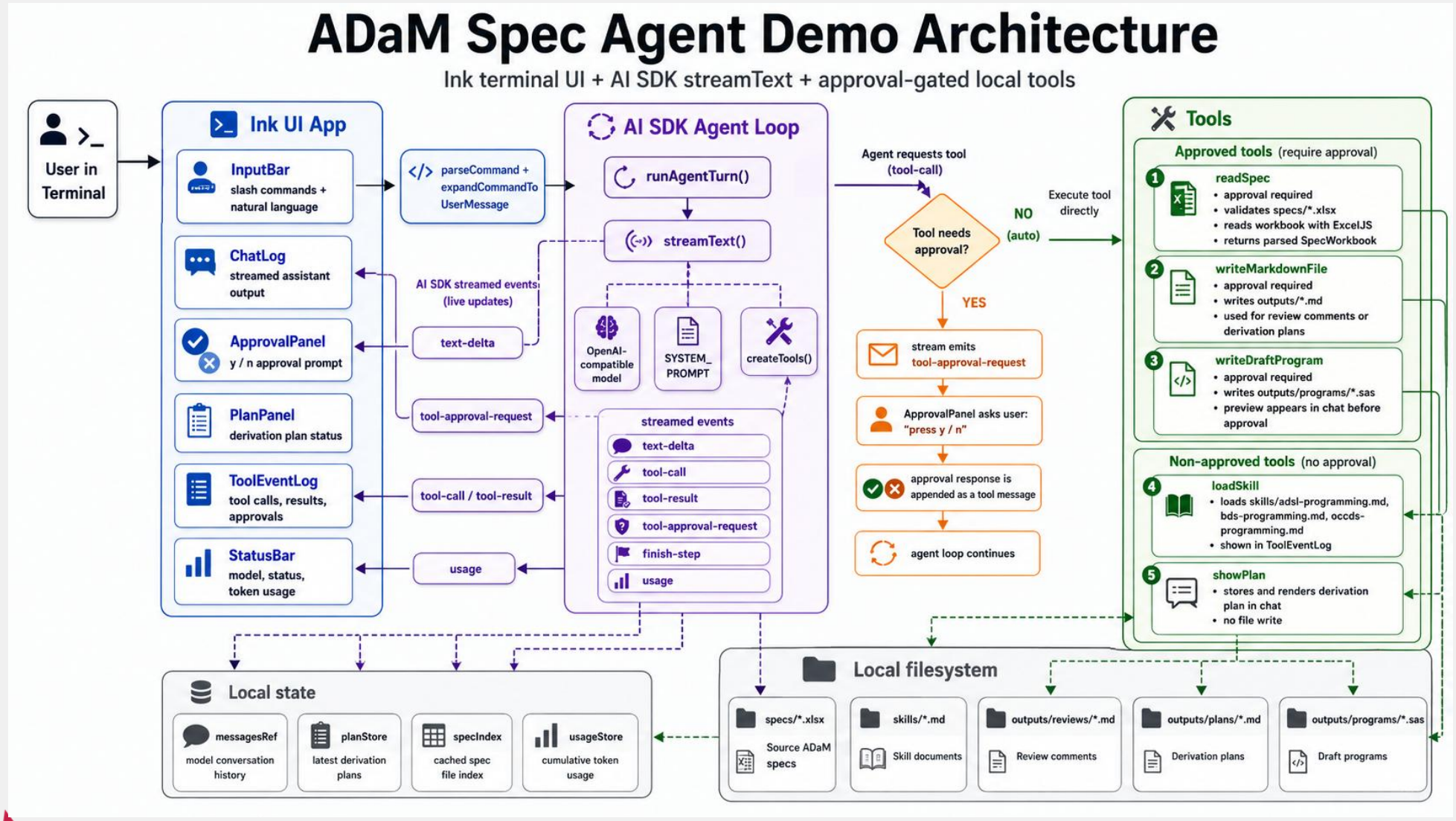
Key idea

Tool calling = model reasoning + tools dictionary + optional approval

1. Tools extend the model beyond text
2. Register multiple tools in tools: { ... }
3. needsApproval adds a user safety check
4. Tool results help produce better answers

Demo App

Demo App



https://github.com/kenwdcao/ink-ai-sdk-demo

Thank you

phastar