@NGAF/CHAT · ENTERPRISE GUIDE

The Enterprise Guide to Agent Chat Interfaces in Angular

Production agent chat UI in days, not sprints

cacheplane.ai · 2026

Contents

CHAPTER 1

The Sprint Tax

The Sprint Tax

Every team building an Angular agent application eventually builds the same chat UI from scratch. Message list, input box, streaming token display, auto-scroll, loading states, error handling. It takes 4-6 weeks. Then they iterate on it for another 4-6 weeks. Meanwhile, the agent backend is ready and waiting.

This isn't a skill gap. It's a structural inefficiency baked into how we approach agent interfaces.

The Inventory

Here's what every production chat UI actually needs:

Message rendering: Not just text. Markdown with code highlighting, tables, lists, inline formatting. Messages that stream in token-by-token. Messages that arrive complete. Messages from the user, from the agent, from tools. Each with different visual treatment.

Streaming display: Tokens arriving at variable rates. Buffering strategies that balance responsiveness against flicker. Cursor indicators. Graceful handling when the stream errors mid-message. Reconnection logic when it drops.

Tool call cards: Tools that are pending, executing, complete, or failed. Tools that return structured data you need to render contextually. Tools that run for 30 seconds and need progress indication. Tools that spawn sub-agents with their own message streams.

Interrupt panels: Human-in-the-loop flows where the agent needs input before proceeding. Multiple choice, free text, confirmation dialogs. State that persists if the user navigates away. Resume logic that picks up where it left off.

Auto-scroll: Sounds simple. Scroll to bottom as new content arrives. Except: don't scroll if the user scrolled up to read history. Do scroll if they scroll back down. Handle window resize. Handle virtual scrolling for conversations with thousands of messages. Handle images that load asynchronously and shift layout.

Accessibility: Screen reader announcements for new messages. Focus management that doesn't trap users. Keyboard navigation through the conversation. ARIA live regions that don't spam. High contrast support. Reduced motion support.

Mobile layout: Viewport that shifts when the keyboard appears. Touch targets that meet minimum sizes. Gesture handling that doesn't conflict with browser gestures.

The Hidden Costs

Accessibility is harder than it looks. Your first implementation will fail an audit. Your second implementation will annoy screen reader users. Your third implementation will work, but you'll have spent two weeks on it.

Streaming token display has edge cases. What happens when you receive a partial UTF-8 sequence? What about markdown that's valid mid-stream but invalid once complete? What about code blocks that open but haven't closed yet?

Tool call state machines are complex. A tool can be requested, approved, executing, streaming results, complete, or failed. It can timeout. It can be cancelled. The user can interrupt. The agent can retry. Each state transition needs UI treatment.

"Good Enough for Demo" vs. Production

Demo chat UI: Messages appear. Input works. It scrolls.

Production chat UI: Messages stream smoothly. Tool calls show real-time progress. Errors recover gracefully. History loads incrementally. Mobile works. Accessibility audits pass. Performance stays stable at 10,000 messages.

The gap between these isn't a weekend. It's eight weeks of senior engineer time, plus ongoing maintenance.

The Opportunity Cost

Here's the actual problem: you're paying senior Angular engineers to solve problems that have already been solved. Problems that have nothing to do with what makes your agent application valuable.

While your team is debugging auto-scroll edge cases, your agent backend is ready and waiting. While they're implementing tool call state machines, your differentiating features aren't getting built. While they're fixing accessibility audit failures, your competitors are shipping.

The @ngaf/chat Thesis

`@ngaf/chat` ships the complete inventory: ``, ``, ``, ``, ``. Production-grade. Accessible. Mobile-ready. Streaming-optimized.

The thesis is simple: ship the chat UI on day one. Spend the sprints on what differentiates your product—the agent logic, the tool integrations, the domain-specific features that

your competitors can't copy.

The sprint tax is optional. Stop paying it.

CHAPTER 2

Batteries-Included Components

Batteries-Included Components

@ngaf/chat ships two component tiers: headless primitives that encapsulate behavior without styling opinions, and prebuilt compositions that deliver production-ready interfaces with minimal configuration. The separation lets teams adopt complete solutions immediately while preserving escape hatches for custom requirements.

The Headless Tier

Headless components own behavior and state management but emit no styled markup. They expose content projection slots and structural directives for complete template control.

`` manages scroll position, virtualization hints, and message grouping logic. It consumes `Message[]` from the agent surface and handles the complexity of streaming message updates—partial content, role transitions, and optimistic UI states. Your templates define how each message renders.

`` handles submit semantics, keyboard shortcuts, disabled states during streaming, and multiline expansion. It exposes form control bindings without prescribing input styling.

`` and `` manage tool invocation display, including pending states, execution results, and error handling. `` surfaces human-in-the-loop decision points when the agent requires input to proceed.

These primitives compose freely. Use them when your design system mandates specific markup structures or when accessibility requirements demand particular ARIA patterns.

The Prebuilt Composition Tier

The `` component assembles headless primitives with production styling and sensible defaults. It accepts an agent instance and renders a complete interface:

```
@Component({
  selector: 'app-support',
  template: ``,
})
export class SupportComponent {
```

```
  supportAgent = agent({
    transport: new FetchStreamTransport(),
    assistantId: 'support-assistant',
    apiUrl: '/api/langgraph',
  });
}
```

Six lines deliver a functional chat interface with message history, streaming indicators, input handling, and tool call display. The component handles loading states, error presentation, and responsive layout without additional configuration.

Companion components—`ChatMessageListComponent`, `ChatInputComponent`, `ChatToolCallsComponent`, `ChatInterruptPanelComponent`, `ChatDebugComponent`—can be used independently when you need prebuilt styling for specific sections while customizing others.

Composing Tiers

The practical pattern: use prebuilt components for standard sections, drop to headless for custom requirements. A typical enterprise implementation might use the prebuilt message list and input while providing a custom tool call renderer that integrates with internal component libraries.

The `` component accepts content projection for this purpose. Override specific slots while retaining default behavior elsewhere. When projection proves insufficient, decompose to individual prebuilt components, then to headless primitives as customization needs escalate.

The Agent Contract

Both tiers consume the runtime-neutral `Agent` contract returned by `agent()` from @ngaf/langgraph. This contract exposes signals: `messages()` returns `Message[]` representing the conversation, `status()` indicates connection state, `isLoading()` reflects pending operations, `toolCalls()` surfaces invocations, and `state()` provides LangGraph checkpoint data.

Components bind directly to these signals. When the agent streams a response, `messages()` updates reactively. Components re-render affected sections without manual change detection. The `Message` type from @ngaf/chat provides the runtime-neutral representation—role, content, metadata—that components consume regardless of the underlying LangGraph message format.

Choosing Your Tier

Start with ``. It covers the common case and establishes baseline functionality in minutes. Drop to prebuilt companions when you need to rearrange layout or inject custom sections between standard elements. Move to headless primitives when your design system requires specific DOM structures or when you're building novel interaction patterns.

Migration between tiers is mechanical: prebuilt components are compositions of headless primitives with styling applied. Extracting customization points means identifying which primitive to expose and which styling to preserve. The agent contract remains constant across tiers—your backend integration stays unchanged regardless of which component tier renders the interface.

CHAPTER 3

Theming & Design System Integration

Theming & Design System Integration

A chat interface that looks like a demo is a liability in production. Users notice when components don't match the rest of the application—inconsistent border radius, wrong font stack, off-brand colors. These details erode trust in the product and in the AI features you're shipping.

@ngaf/chat exposes its visual design decisions through CSS custom properties, giving you control over appearance without touching component internals or maintaining forks.

The CSS Custom Property API

Every visual decision in @ngaf/chat maps to a custom property. The components read these values at runtime, so overriding them in your stylesheet changes the rendered output immediately.

The naming convention follows a predictable pattern: `--chat-{component}-{property}`. Component-level tokens reference global tokens, which reference your design system tokens. This layering lets you override at whatever granularity makes sense—change one button's border radius or change every border radius in the chat interface with a single line.

Design Token Mapping

If your team already maintains design tokens, integration is direct assignment. Map your existing tokens to the chat component tokens in a single stylesheet:

```
:root {
  /* Typography */
  --chat-font-family: var(--ds-font-family-body);
  --chat-font-size-base: var(--ds-font-size-md);
  --chat-line-height: var(--ds-line-height-normal);
/* Colors */
  --chat-surface-primary: var(--ds-color-surface-elevated);
  --chat-surface-secondary: var(--ds-color-surface-sunken);
  --chat-text-primary: var(--ds-color-text-primary);
```

```
  --chat-text-secondary: var(--ds-color-text-muted);
  --chat-accent: var(--ds-color-brand-primary);

/* Shape */
  --chat-border-radius: var(--ds-radius-md);
  --chat-spacing-unit: var(--ds-spacing-base);


/* State colors */
  --chat-color-error: var(--ds-color-feedback-error);
  --chat-color-success: var(--ds-color-feedback-success);
}
```

This mapping becomes your single source of truth. When design updates the brand's border radius, the chat components update automatically because they reference your tokens, not hardcoded values.

Typography Integration

Chat interfaces are text-heavy. Typography consistency matters more here than in most UI contexts.

@ngaf/chat exposes tokens for font family, size scale (base, small, large), line height, and font weight. Message content, timestamps, tool call labels, and input placeholders all reference these tokens. Set them once, and the entire typographic hierarchy follows your design system.

Color System Integration

Surface colors control backgrounds—the chat container, message bubbles, input field. Text colors handle primary content and secondary metadata. Accent colors drive interactive elements and visual emphasis. Semantic state colors handle error states, success confirmations, and loading indicators.

The token structure assumes you have these categories in your design system. If you don't, use literal values.

Dark Mode Support

The token system supports dark mode without component changes. Override the custom properties inside a dark mode selector:

```
[data-theme="dark"] {
  --chat-surface-primary: var(--ds-color-surface-elevated-dark);
  --chat-text-primary: var(--ds-color-text-primary-dark);
}
```

Components don't contain theme-switching logic. They read current token values. Your application controls when those values change.

Limitations of Token-Based Theming

CSS custom properties control visual properties—colors, spacing, typography, borders. They don't control structure.

If you need different DOM layout, custom animations, or component composition that diverges from the default, tokens won't help. This is where the headless pattern applies: use `agent()` directly with your own components, keeping the streaming infrastructure while owning the entire render layer.

Tokens handle 80% of enterprise theming needs. The headless tier handles the rest.

CHAPTER 4

Generative UI in Chat

Generative UI in Chat

Text is a bottleneck. When a financial agent needs to present quarterly results, streaming prose about revenue figures wastes cognitive load. When a scheduling agent confirms a booking, a wall of text obscures the actionable details. The most capable agent interfaces solve this by rendering structured UI directly in the message stream—tables, forms, approval cards—alongside conversational text.

Structured UI in the Message Stream

@ngaf/chat treats generative UI as a first-class message type. When an agent emits a UI specification instead of text, the chat renders it inline using @ngaf/render. From the user's perspective, the conversation flows naturally: the agent explains context in prose, then presents a rendered component for interaction.

This works because the Agent contract exposes messages as a heterogeneous stream. Text messages render as text. UI messages resolve to Angular components through a registry. The chat component handles both transparently.

The json-render Spec

The json-render specification defines a minimal contract for declarative UI. An agent emits a JSON object with a `type` field identifying the component and a `props` field containing its inputs:

```
{"type": "data-table", "props": {"columns": ["Quarter", "Revenue"], "rows": [...]}}
```

The renderer resolves `data-table` to an Angular component, binds `props` to its inputs, and inserts it into the DOM. No custom parsing, no message-type switches in templates. The specification stays minimal intentionally—agents describe *what* to render, not *how*.

A2UI: Agent-Specific Patterns

Google's A2UI specification extends json-render with patterns specific to agent interactions: approval workflows, structured actions, and rich data displays. Where json-render handles arbitrary components, A2UI codifies the common cases—confirmation dialogs, multi-step forms, action buttons with pending states.

@ngaf/render supports both specifications. You can mix A2UI's structured patterns with custom json-render components in the same message stream.

Registry Integration

Component resolution flows through a registry. You define which component handles each type, then provide it to the chat context:

```
import { defineAngularRegistry, provideRender } from '@ngaf/render';
import { DataTableComponent } from './data-table.component';
import { BookingFormComponent } from './booking-form.component';
import { ApprovalCardComponent } from './approval-card.component';
const registry = defineAngularRegistry({
  'data-table': DataTableComponent,
  'booking-form': BookingFormComponent,
  'approval-card': ApprovalCardComponent,
});


// In your providers array
provideRender({ registry })
```

The chat component picks up the registry through dependency injection. When a message contains a render spec, it resolves and instantiates the component automatically.

Streaming Patches

Agents rarely emit complete UI specifications in one shot. A data table streams rows as they arrive. A form populates fields progressively. @ngaf/render handles this through JSON Patch streaming—the agent emits an initial skeleton, then streams RFC 6902 patches that mutate the specification incrementally.

In the chat context, this creates the live update effect users expect. A table appears with headers, then rows populate one by one. A summary card fills in as the agent processes. The render layer applies patches reactively; bound components update without re-instantiation.

This matters for perceived latency. Users see structure immediately, then watch it fill in—progress feels continuous rather than blocked on complete responses.

CHAPTER 5

Debug Tooling

Debug Tooling

Debugging agent chat is hard. The message stream is opaque, tool call state transitions happen in milliseconds, and interrupt flows have timing edge cases that only surface under load. You can't step through a streaming conversation the way you step through synchronous code.

`` is @ngaf/chat's built-in debug panel—a developer overlay that surfaces agent state, raw message events, tool call history, and interrupt state in real time. One component, zero configuration.

What chat-debug Shows

The debug panel exposes four primary views:

Message State. The current `Message[]` array, rendered as expandable JSON. You see every message the agent has processed—user inputs, assistant responses, tool results—with full metadata intact.

Streaming Event Log. A chronological log of every event received from the transport layer. This includes partial tokens during streaming, state updates, tool call initiations, and completion signals. The log timestamps each event to the millisecond.

Tool Call State Machine. Each tool call passes through distinct states: pending, executing, completed, or failed. The debug panel visualizes this state machine for every tool call in the current conversation, showing transitions as they happen.

Interrupt Payload. When the agent requests human intervention, the full interrupt payload appears in the panel—the interrupt type, the data the agent is requesting, and any context it provided. After user action, you see both the original payload and the response.

Adding chat-debug

Drop it into any chat interface:

```
import { ChatDebugComponent } from '@ngaf/chat';
```

```
@Component({
  selector: 'app-support-chat',
  imports: [ChatComponent, ChatDebugComponent],
  template: `


  `
})
export class SupportChatComponent {
  agent = agent({
    transport: new FetchStreamTransport(),
    assistantId: 'support-agent',
    apiUrl: '/api/langgraph'
  });
}
```

No configuration required. The panel renders in the bottom-right corner with a toggle to expand and collapse.

Inspecting Individual Messages

Click any message in the Message State view to expand it. You'll see:

- **Message type**: user, assistant, tool, or system
- **Content**: the full text or structured content
- **Metadata**: timestamps, message IDs, and any custom properties your agent attaches
- **Token count**: if your transport provides it, the token usage for that message

For assistant messages during active streaming, content updates live as tokens arrive.

Tool Call Debugging

The tool call view shows each invocation with:

- **Tool name**: the function the agent called
- **Input payload**: the exact JSON the agent passed to the tool
- **Output**: the tool's response, once execution completes
- **Execution timing**: start time, end time, and duration in milliseconds

When a tool call fails, the panel displays the error message and stack trace. This is where you'll catch malformed inputs, timeout issues, and permission errors.

Interrupt State Inspection

Interrupts are the hardest flow to debug without tooling. The agent pauses, waits for user input, then resumes—but what exactly is it waiting for?

The interrupt view shows the full payload the agent sent when requesting the interrupt. After the user responds, you see both sides: what was asked and what was provided. This catches mismatches between what your interrupt UI collects and what the agent expects.

Integration with Angular DevTools

Agent signals created by `agent()` appear in Angular DevTools like any other signal. In the component tree, you'll see `messages`, `status`, `isLoading`, `toolCalls`, and `state` as inspectable signals with their current values.

This means standard Angular debugging workflows apply. Set a breakpoint, inspect signal values, trace reactivity—the agent surface behaves like any other signal-based state.

Production Safety

`` uses Angular's `isDevMode()` check internally. In production builds, the component renders nothing—no DOM nodes, no event listeners, no performance overhead. Leave it in your templates; the build process handles the rest.

For teams that need debug capabilities in staging environments, pass `[forceEnable]="true"` to override the dev mode check. Use this sparingly and gate it behind feature flags.