@NGAF/LANGGRAPH · ENTERPRISE GUIDE

The Enterprise Guide to Agent Streaming in Angular

Ship LangGraph agents in Angular — without building the plumbing

cacheplane.ai · 2026

Contents

CHAPTER 1

The Last-Mile Problem

The Last-Mile Problem

You've built the backend. The LangGraph agent handles multi-step reasoning, calls tools, maintains conversation memory, and streams responses token by token. You've tested it with curl, watched it work in LangGraph Studio, maybe even built a quick React prototype. The agent architecture is solid.

Then you integrate it with your Angular application.

Zone Pollution Is Architectural, Not Configurable

The first symptom appears quickly: performance degradation during streaming. Every SSE event triggers zone.js change detection. A typical LLM response generates hundreds of token events over several seconds. Each event runs through `Zone.wrap()`, schedules a microtask, and triggers a full change detection cycle. Your application becomes unresponsive while the agent is responding.

The instinctive fix—running the EventSource outside the zone—creates new problems. Updates don't reach templates. Manual `ChangeDetectorRef.detectChanges()` calls scatter through your codebase. You're now maintaining zone-aware and zone-unaware code paths for the same data flow.

This isn't a configuration problem you can solve with `NgZone.runOutsideAngular()`. It's a fundamental mismatch between SSE's event model and Angular's zone-based change detection architecture.

Synchronous Templates, Asynchronous Tokens

Angular's template binding model expects synchronous state reads. Signals improved this, but the core assumption remains: when a template renders, it reads current values and completes. LLM token streams don't work this way. Tokens arrive continuously, accumulate into partial content, and may include control events (tool calls, interrupts) interleaved with text.

The naive implementation—updating a signal on every token—violates Angular's expectation of stable reads during change detection. You get ExpressionChangedAfterItHasBeenChecked errors, visual flickering, or worse: dropped tokens during rapid updates.

Batching tokens into animation frames helps but introduces its own complexity. You're now managing accumulation buffers, flush timing, and ensuring final state consistency when streams complete or error.

Push vs. Pull: The Reactivity Mismatch

RxJS Observables are push-based. Angular signals are pull-based. LLM streams are push-based with ordering guarantees. Bridging these models requires careful coordination.

Your REST-era patterns don't transfer. An HTTP response completes atomically; you handle loading, success, or error states. A streaming agent response is loading *and* partially successful *and* potentially errored, simultaneously. Tool calls arrive mid-stream. Human interrupts pause processing indefinitely. Partial content is valid content.

The standard `toSignal()` approach gives you the latest emission but loses the accumulated message history. Building that accumulation logic—correctly handling message append vs. replace semantics, tool call lifecycle states, and interrupt coordination—requires understanding LangGraph's event protocol, not just Angular's reactivity model.

The Real Cost

Teams solve these problems. They build zone-patch utilities, token accumulator services, retry-with-backoff wrappers, and error boundary

components. They write tests for partial stream failure, reconnection logic, and concurrent stream management.

Then the next project starts, and they build it again. Or they copy the code, discover edge cases the original didn't handle, and fork into divergent implementations.

The gap between a working demo and production-safe Angular integration is measured in weeks of engineering time, repeated across every team building agent-powered features. The backend streaming problem was solved. The frontend streaming problem keeps getting re-solved.

CHAPTER 2

The agent() API

The agent() API

The `agent()` function is the primary interface for streaming LangGraph agents into Angular components. It returns a `LangGraphAgent` instance containing reactive signals that update automatically as the agent stream progresses. No subscriptions. No cleanup. No zone gymnastics.

Signal Architecture

Calling `agent()` returns an object with typed signals covering the full agent lifecycle:

- **`messages()`** — The accumulated message history as `Message[]`, updated with each stream chunk
- **`isLoading()`** — Boolean signal indicating active stream processing
- **`error()`** — The current error state, or `undefined` when healthy
- **`interrupt()`** — `AgentInterrupt | undefined`, populated when the agent yields control for human input
- **`status()`** — Runtime lifecycle state: `'idle'` | `'running'` | `'error'`; use `isLoading()` for loading UI
- **`toolCalls()`** — Active tool invocations extracted from the message stream
- **`state()`** — The current agent state object from the LangGraph thread

For cases requiring access to the raw LangGraph protocol, additional signals like `langGraphMessages()` expose the unprocessed message format.

Provider Configuration

Before `agent()` can connect, configure the transport layer with `provideAgent()`:

```
provideAgent({
  transport: new FetchStreamTransport()
})
```

This registers the stream transport globally. Individual `agent()` calls then specify their endpoint:

```
readonly chat = agent({
  assistantId: 'support_agent',
  apiUrl: 'https://api.example.com/langgraph',
  threadId: this.threadId,
  onThreadId: id => this.threadId.set(id)
});
```

The `assistantId` identifies the deployed agent. The `apiUrl` points to your LangGraph API endpoint. Thread management is handled through the `threadId` input and `onThreadId` callback.

Why Signals Work with OnPush

Angular's `OnPush` change detection strategy only triggers updates when input references change or when signals read in the template emit new values. Because `agent()` returns signals—not observables requiring `async` pipes—the framework detects changes automatically when stream chunks arrive.

No `markForCheck()`. No `ChangeDetectorRef` injection. The signals integrate with Angular's reactivity system at the primitive level.

Template Binding

Bind agent state directly in templates without ceremony:

```
@Component({
  template: `
    @if (chat.isLoading()) {
```

```
    }
    @for (message of chat.messages(); track message.id) {

    }
    @if (chat.error(); as error) {

    }
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChatComponent {
  readonly chat = agent({ assistantId: 'chat_agent' });
}
```

Ten lines. The stream connects, messages accumulate, loading state toggles, errors surface—all reactive, all type-safe.

The Alternative

Without `agent()`, the equivalent implementation requires manual stream handling:

```
// Manual approach: ~60 lines of subscription management,
// token accumulation, error handling, cleanup logic,
// and change detection triggers
```

With `agent()`:

```
readonly chat = agent({ assistantId: 'chat_agent' });
```

The signal-native design eliminates the subscription lifecycle entirely. When the component destroys, the signals become inert. No `takeUntilDestroyed()`. No `ngOnDestroy`. The framework handles it.

CHAPTER 3

Thread Persistence & Memory

Thread Persistence & Memory

Production agent applications are stateful. Users expect to close a browser tab, return hours later, and resume exactly where they left off. This requires tight coordination between LangGraph's checkpoint system and your Angular frontend's thread management.

The Thread Lifecycle

LangGraph's `MemorySaver` backend persists conversation state against a `threadId`. Every message, tool call, and state mutation is checkpointed. The frontend's job is simple: track which `threadId` the user is working with and ensure it survives page reloads.

The `agent()` surface exposes this through two mechanisms. First, `threadId` accepts a signal containing the current thread identifier—pass `undefined` to create a new conversation. Second, `onThreadId` fires when LangGraph assigns an ID to a newly created thread.

```
@Component({
  template: ``,
  providers: [provideAgent()]
})
export class ChatPage {
  private readonly storage = inject(ThreadStorageService);

  readonly threadId = signal(
    this.storage.getActiveThreadId()
  );

  readonly chat = agent({
    assistantId: 'support_agent',
    threadId: this.threadId,
```

```
      onThreadId: id => {
        this.storage.setActiveThreadId(id);
        this.storage.addToThreadList(id);
        this.threadId.set(id);
      }
    });
  }
```

When `threadId` is `undefined`, the first `submit()` call triggers thread creation on the backend. LangGraph responds with the assigned ID, which flows through `onThreadId`. You persist it, update your signal, and subsequent messages automatically route to the correct checkpoint.

Restoring State on Mount

When a user returns with an existing `threadId`, LangGraph's checkpoint system handles restoration automatically. The backend loads the conversation history from `MemorySaver`, and the frontend receives the full message stream during the initial connection handshake.

This means your `messages()` signal populates with historical content without additional API calls. The `langGraphCheckpoint()` signal exposes metadata about the restored state—useful for debugging or displaying "last active" timestamps.

Building a Thread List

Most applications need more than single-thread persistence. Users expect to manage multiple conversations:

```
@Component({
  template: `


      @for (id of threadIds(); track id) {
        
          {{ id | slice:0:8 }}...
        
      }
      New Conversation
```

```
    `
  })
  export class MultiThreadChat {
    readonly threadIds =
  signal(this.storage.getAllThreadIds());
    readonly threadId =
  signal(this.storage.getActiveThreadId());

    switchThread(id: string) {
      this.threadId.set(id);
      this.storage.setActiveThreadId(id);
    }

    newThread() {
      this.threadId.set(undefined);
    }
  }
```

Switching threads is a signal update. The `agent()` reactive system handles reconnection, state restoration, and UI synchronization.

Production Considerations

Server-side thread expiration creates a failure mode your UI must handle. `MemorySaver` configurations often include TTLs—threads expire after periods of inactivity. When a user selects a stale `threadId`, the backend returns an error rather than conversation history.

Watch the `error()` signal for thread-not-found conditions. Your recovery logic should remove the invalid ID from local storage, notify the user, and optionally create a fresh thread automatically.

Production checklist:

- Does your thread list handle deleted or expired server-side threads gracefully?
- Are you cleaning up localStorage when threads fail to load?
- Do you display meaningful state when restoration is in progress versus complete?

- Have you considered thread metadata (titles, timestamps) beyond raw IDs?

The `MemorySaver` backend and Angular's signal-based reactivity create a clean separation of concerns. The backend owns durability; the frontend owns navigation. Keep that boundary crisp.

CHAPTER 4

Interrupt & Approval Flows

Interrupt & Approval Flows

Agents that modify external systems—sending emails, executing database writes, triggering deployments—require human oversight. Autonomous execution without checkpoints creates liability, compliance violations, and irreversible mistakes. LangGraph's `interrupt()` primitive solves this at the graph level, pausing execution mid-stream until a human provides explicit authorization. `@ngaf/langgraph` surfaces this as a reactive signal, making approval workflows native to Angular's change detection without polling, websockets, or custom resume endpoints.

How LangGraph Interrupt Works

When a LangGraph node calls `interrupt()`, the graph halts execution and persists its current state to the configured checkpointer. The interrupt payload—containing context about the pending action—is sent to the client as part of the stream. Execution remains suspended until the client sends a resume command with one of three directives: approve the action as-is, provide edited parameters, or cancel entirely.

The resume payload structure is straightforward:

```
{ action: 'approve' }                    // Proceed with
original parameters
{ action: 'edit', args: { ... } }        // Proceed with
modified parameters
{ action: 'cancel', reason?: string }    // Abort the
pending action
```

LangGraph's `Command.RESUME` handles the routing. The graph receives the payload and either continues execution, re-executes with new arguments, or terminates gracefully.

The interrupt() Signal

The agent surface exposes `interrupt()` as a signal that transitions from `undefined` to an `AgentInterrupt` object when the graph pauses:

```
interface AgentInterrupt {
  id: string;
  type: string;
  payload: unknown;
  timestamp: number;
}
```

This signal integrates directly with Angular's reactivity model. Components re-render automatically when an interrupt arrives—no subscription management, no manual change detection triggers. When the user responds and execution resumes, the signal returns to `undefined`.

Prebuilt Approval UI

`@ngaf/chat` provides ``, a ready-to-use approval interface:

```
@Component({
  selector: 'app-agent',
  template: `


  `
})
export class AgentComponent {
  readonly chat = agent({ assistantId: 'ops_agent' });
}
```

The panel renders contextual information from the interrupt payload, displays approve/cancel actions, and handles the resume flow. For custom requirements, bind the signal directly:

```
@Component({
  template: `
    @if (chat.interrupt(); as int) {


{{ int.payload.description }}


        Approve
        Cancel


    }
  `
})
export class CustomApprovalComponent {
  readonly chat = agent({ assistantId: 'ops_agent' });
}
```

Calling `interrupt()` with a resume payload sends the command and clears the signal.

Edge Cases

Navigation during interrupt: LangGraph persists interrupt state server-side. If the user navigates away, the interrupt remains active. Re-initializing the agent with the same `threadId` restores the pending interrupt automatically.

Session expiration: Checkpointed state survives session boundaries. The interrupt signal repopulates on reconnection, though your application should handle re-authentication before allowing resume actions.

Cancel with partial state: Cancellation doesn't roll back prior node executions. If three nodes completed before the interrupt, those effects persist. Design graphs with compensation logic for actions that require atomicity, or structure interrupts to occur before side effects rather than after.

Multiple pending interrupts: LangGraph supports sequential interrupts within a single run. The `interrupt()` signal reflects the current pending interrupt; each resume advances to the next pause point or completion.

Human-in-the-loop isn't optional for production agents. `@ngaf/langgraph` makes it reactive, type-safe, and compatible with Angular's rendering model—approval flows become UI state, not infrastructure problems.

CHAPTER 5

Full LangGraph Feature Coverage

Full LangGraph Feature Coverage

Most Angular LLM integrations handle the basics: send a message, stream tokens, render a response. The moment you need tool calls, subgraphs, or multi-agent coordination, you're writing raw SSE parsers and manually reconciling state. @ngaf/langgraph exists specifically to avoid that cliff—every LangGraph feature surfaces through the same reactive signals your components already consume.

Tool Call Streaming

LangGraph emits tool invocations as structured events mid-stream. Rather than parsing `tool_call` chunks yourself, the agent ref exposes them directly:

```
readonly chat = agent({
  assistantId: 'research_agent',
  apiUrl: 'https://api.smith.langchain.com'
});
// In your template
@for (call of chat.toolCalls(); track call.id) {

}
```

The `toolCalls()` signal updates as invocations arrive, complete as the agent processes results, and clear when the turn ends. No manual event filtering. Tool call arguments stream incrementally—useful for showing users what data the agent is requesting before results return.

Subgraph Support

Nested graphs emit events with their own namespaces. @ngaf/langgraph flattens these into the primary stream while preserving hierarchy through the `subagents()` signal:

```
// Parent agent spawns child graphs for specialized tasks
const activeSubagents = this.chat.subagents();
// Returns SubagentInfo[] with id, name, status for each
active subgraph
```

Child graph messages, tool calls, and state updates flow through the same signals. When a subgraph completes, its final state merges into the parent. Your components don't need conditional logic for nested versus top-level events—they render identically.

Time Travel

LangGraph checkpoints graph state at each node. Rewinding means re-streaming from a prior checkpoint:

```
// Rewind to a previous state and continue from there
this.chat.interrupt({
  checkpoint_id: 'abc123',
  action: 'rewind'
});
```

The `langGraphCheckpoints()` signal exposes available restore points. After rewinding, `messages()` reflects the restored state and streaming continues from that node. This enables "undo" flows, A/B comparison of agent paths, and debugging without replaying the entire conversation.

DeepAgent Multi-Agent Coordination

DeepAgent orchestrates multiple specialized agents through a supervisor pattern. At the stream level, this means interleaved events from distinct agents with coordination metadata. The agent ref normalizes this:

```
// Each agent's output tagged with origin
const messages = this.chat.messages();
// Message.metadata.agent identifies the source agent
// Coordination state available through
const graphState = this.chat.langGraphState();
// Includes active_agent, delegation_history,
shared_context
```

Your UI can render agent-specific styling, show delegation chains, or visualize the coordination graph—all from signals that update as events arrive.

The onCustomEvent Hook

Agents emit structured events beyond messages: progress indicators, analytics payloads, generative UI specs. The `onCustomEvent` callback captures these without polluting the message stream:

```
readonly chat = agent({
  assistantId: 'ui_agent',
  onCustomEvent: (event) => {
    if (event.type === 'render_component') {
      this.dynamicUI.set(event.payload);
    }
  }
});
```

This separates concerns cleanly: messages render in the chat, custom events drive application-specific behavior.

Why Full Coverage Matters

The pattern we've seen repeatedly: teams adopt a library for basic chat, then bypass it entirely when requirements expand. They end up maintaining parallel implementations—the library for simple flows, raw

SSE handling for everything else. That's two mental models, two bug surfaces, two upgrade paths.

Full feature coverage eliminates that bifurcation. Tool calls, subgraphs, time travel, and multi-agent coordination all flow through the same `agent()` call. When LangGraph adds capabilities, they surface through existing signals rather than requiring new integration code. Your components stay declarative. Your state stays reactive. The complexity lives in the library, not your application.

CHAPTER 6

Deterministic Testing

Deterministic Testing

Agent UIs are notoriously difficult to test. Real LLM calls introduce latency measured in seconds, non-deterministic outputs, rate limits, and network dependencies that make CI pipelines slow and flaky. A test that passes locally might fail in CI because the model returned a slightly different response, or the API throttled your request, or the stream took longer than your timeout.

The solution is deterministic mocking at the transport layer. NGAF provides two complementary approaches: `MockAgentTransport` for scripting realistic SSE event sequences, and `mockLangGraphAgent()` for direct signal manipulation when you need fine-grained control.

MockAgentTransport: Scripted Event Sequences

`MockAgentTransport` replaces `FetchStreamTransport` in tests, emitting a predetermined sequence of SSE events without any network calls. You script exactly what the agent receives—message chunks, tool calls, interrupts, errors—and the transport replays them synchronously or with configurable delays.

```
import { TestBed } from '@angular/core/testing';
import { MockAgentTransport, provideAgent } from
'@ngaf/langgraph';
describe('ChatComponent', () => {
  let transport: MockAgentTransport;


beforeEach(() => {
    transport = new MockAgentTransport();
    TestBed.configureTestingModule({
```

```
      imports: [ChatComponent],
      providers: [provideAgent({ transport })]
    });
  });
});
```

This setup runs entirely offline. No HTTP interceptors, no mock servers, no environment configuration. The transport is synchronous by default, meaning your test assertions execute immediately after triggering a submit.

Testing Agent States

Every agent state your UI handles needs a corresponding test. `MockAgentTransport` lets you script each scenario explicitly:

Streaming in progress: Emit partial message events without a completion event. Assert that `isLoading()` returns true and the message list shows the partial content.

Stream complete: Emit the full event sequence including the completion marker. Assert that `isLoading()` returns false and messages contain the final content.

Interrupt pending: Script an interrupt event mid-stream. Assert that `interrupt()` returns the interrupt payload and your interrupt panel renders the expected options.

Error state: Emit an error event. Assert that `error()` contains the error details and your error UI appears.

Direct Signal Control with mockLangGraphAgent

When testing component rendering in isolation—without exercising the transport layer—use `mockLangGraphAgent()` to create an agent instance with directly controllable signals:

```
const mockAgent = mockLangGraphAgent({
  messages: signal([{ role: 'assistant', content: 'Test
response' }]),
  status: signal('complete'),
  toolCalls: signal([{ id: 'tc1', name: 'search', args: {
query: 'test' } }])
});
```

Pass this mock to components that accept an agent input. You control exactly what signals return, making it trivial to test tool call rendering, generative UI output, and edge cases like empty states or malformed data.

Testing Thread Switching

Thread switching tests verify that your component correctly handles `threadId` changes and `onThreadId` callbacks. Script a sequence where the transport emits a new thread ID, then assert that your `onThreadId` handler persisted the value and subsequent messages associate with the correct thread.

The Benchmark

Agent component tests should run offline and complete in under 100ms each. This isn't aspirational—it's achievable when you eliminate network calls and async delays. A test suite with 50 agent UI tests should finish in under 5 seconds, run identically on developer machines and CI, and produce the same results every time.

If your agent tests take longer or exhibit flakiness, you're either hitting real infrastructure or introducing unnecessary async delays in your mocks. Fix the mocking strategy, not the timeout thresholds.