

Title

Author

Early Access

‹packt›

Building Ambient AI Agents

Create autonomous always-on product experiences that move beyond chat



Frederick Joseph Patton | Leman Pinar Patton

Building Ambient AI Agents

Create autonomous always-on product experiences that move beyond chat

Frederick Joseph Patton
Leman Pinar Patton



Building Ambient AI Agents

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.
Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Early Access Publication: Building Ambient AI Agents

Early Access Production reference: B38467

First published: Mar 2027

Production reference: ---------

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83588-250-4
www.packtpub.com

Table of Contents

Welcome to Packt Early Access

Building Ambient AI Agents, First Edition: Create autonomous always-on product experiences that move beyond chat

Chapter 1: Beyond Chat: The Emergence of Ambient Agents

Traditional AI Agents: Lacking Initiative

Failure 1: Cognitive Load •

Failure 2: Context Reset •

Failure 3: Missed Timing •

Failure 4: Unknown Unknowns •

Failure 5: Interaction Burden •

Why These Failure Modes Are Now Solvable •

The Architecture Will Outlast the Model •

Ambient Agents: Observation, Memory, Restraint

Case Example: Deploy-Time Code Review •

1. Observation •

Three Principles of Observation •

What the Agent Can Monitor •

An Observation Example •

Observation and Memory •

2. Memory •

3. Restraint •

The Dance of Three: Observation, Memory, Restraint •

Is Your Application a Good Candidate for an Ambient Agent?

Do you need to understand the full spectrum of user behavior? •

Do you need this understanding in real time? •

Do you need to take action on what you learn? •

Summary

Chapter 2: Core Technologies: Building Blocks for Ambient Agents

- The Ambient Agent Five-Phase Loop
  - Sense •
  - Interpret •
  - Decide •
  - Act •
  - Learn •
  - The Loop •
- Neo4j: Memory and Application Context
  - The Application Context Graph •
  - Heuristic Configuration •
  - Outcome Storage •
- LangGraph: The Decision Spine
  - Why a Graph? •
  - State as the Agent's Memory •
  - The Graph Topology •
- CopilotKit and AG-UI: The Human-Agent Collaboration Layer
  - The AG-UI Protocol •
  - Frontend Tools with useCopilotAction •
  - The Provider Pattern •
- Deepgram: Voice as a Sensing Channel
  - Challenge of voice processing •
  - The Backend Proxy Pattern •
- ElevenLabs: Voice as an Act Channel
  - The Streaming Proxy and MediaSource Playback •
- Voice Orchestration: Turn-Taking and Interruption
  - Voice Activity Detection •
  - The Turn-Taking State Machine •
- What the Technology Stack Gives You
  - Stateful session memory across episodes •
  - Frontend tool execution from backend logic •
  - Sub-second voice I/O •
  - Learnable application context •
  - Backend-neutral architecture •
- Summary

Chapter 3: Observation: Sensing User Behavior

- Understanding Our Sensing System
- Implementing the Sensing Model
- Raw Signals
- Classification Primitives
- Patterns
- Candidate Intent
- Interaction Episodes
  - The Base Episode Class •
    - *Accumulation* •
    - *Analysis* •
    - *Inference* •
    - *Serialization* •
    - *Finalization* •
  - Episode Types •
    - *Motion Episodes* •
    - *Reading Episodes* •
    - *Navigation Episodes* •
    - *Form Episodes* •
- Summary

Welcome to Packt Early Access

Building Ambient AI Agents, First Edition: Create autonomous always-on product experiences that move beyond chat

We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: Beyond Chat: The Emergence of Ambient Agents
2. Chapter 2: Core Technologies: Building Blocks for Ambient Agents
3. Chapter 3: Observation: Sensing User Behavior
4. Chapter 4: Interpretation: From Signals to Understanding
5. Chapter 5: Memory: Accumulating Context Across Sessions
6. Chapter 6: Restraint: Knowing When Not to Help
7. Chapter 7: Intervention Design: Choosing How to Help
8. Chapter 8: Security: Building Trustworthy Agents
9. Chapter 9: Onboarding and Activation: Guiding New Users
10. Chapter 10: Complex Configuration: Preventing Downstream Problems
11. Chapter 11: Pre-Escalation Support
12.

1

Beyond Chat: The Emergence of Ambient Agents

Think of 007.

Before the mission, M gives him the briefest of briefings. Just a name, a coordinate, a clue. James Bond figures out the rest himself. He reads the environment. He notices patterns. He sees the danger before you do. And at exactly the right moment, with exactly the right move, he steps in. Nobody tells him, "run left now." He's already there.

Traditional AI agents have performed more like customer service agents. Your problem doesn't get worked on until you pick up the phone and call. Ambient agents, on the other hand, aspire to be like Bond. They work with a stream of clues extracted from noise and seek to act at the perfect moment, solving the problem before the user even has to ask, and in some cases, before the problem is even known to them.

We call this chapter *Beyond Chat* for exactly this reason. Tools like Chat GPT or Claude are powerful and impressive, with millions of daily users, but they all operate within the same fundamental paradigm: they wait until you type something. Sure, they have memory, and they know you, but they don't speak until you ask. They don't act until you make a request and tell them how to act. Even with the screen open in front of you, they're silent until spoken to.

The ambient agent breaks that silence. It doesn't need to be reminded you're there. It's always observing you. Though you may ignore it, it doesn't ignore you. It notices the change in your calendar, a recurring behavioral pattern, a silent friction point, and steps in before you've even formulated the question. The difference isn't memory. The difference is initiative.

This chapter lays the groundwork for everything that follows in this book. We live in a moment where AI is everywhere, and yet most AI products still wait to be spoken to. That gap between what users need and what they think to ask for is where products quietly fail, and it is exactly the gap that ambient agents are position to close. Understanding why that gap exists, and how to architect a system that bridges it, is what this chapter is about.

The goal of this chapter is not simply to introduce a new category of AI systems. It is to reframe how you think about the relationship between software and its users, starting from a model in which users drive every interaction, to one in which a well-designed system can meet users where they are, before they ask. That shift has profound implications for product design, engineering architecture, and user trust

Q: "Sometimes you have to pull the trigger."

Bond: "Or not. It's hard to tell the difference in your pajamas."

As Bond reminds us in Skyfall, knowing when not to pull the trigger is the hardest part. Time to put on our ambient agent suit and find out. This chapter covers:

- Traditional AI Agents: Lacking Initiative
- Ambient Agents: Observation, Memory, Restraint
- Is Your Application a Good Candidate?

Traditional AI Agents: Lacking Initiative

Imagine it's 2023 and a product team has just integrated an AI chat feature into their application. Months of development, carefully crafted prompts, thorough testing, and a launch day that went without a hitch. Yet the metrics never moved. Support tickets kept coming. Activation curves stayed flat.

The reason finally became clear: nobody was asking.

Users don't like to ask. They get stuck, lose confidence, and quietly leave, because a chat window assumes they will stop mid-task, switch contexts, and articulate their problem clearly. That's not how people behave under friction.

The chat window relies on one assumption: the user notices, then stops, then asks. This loop must work every time. In reality, it doesn't. The user might not notice, might not want to stop, or might not know how to ask, and every failure mode stems from this single point of collapse: help only arrives when the user speaks up.

We write this section because every day new AI chat features launch and the same mistakes repeat. Teams think they can fix it with better prompts, bigger models, or more testing, but the problem isn't technical. It's structural. Without naming these failure modes and distinguishing them from one another, designing proactive systems is impossible, and understanding each one separately is the first step toward the solution.

In this section, we examine five structural failure modes of reactive AI interfaces:

- Cognitive Load: the user carries a burden the system should be carrying
- Context Reset: every session starts from scratch, even when it shouldn't
- Missed Timing: help arrives after the moment it would have mattered
- Unknown Unknowns: users can't ask for what they don't know they need
- Interaction Burden: the act of seeking help is itself a source of friction

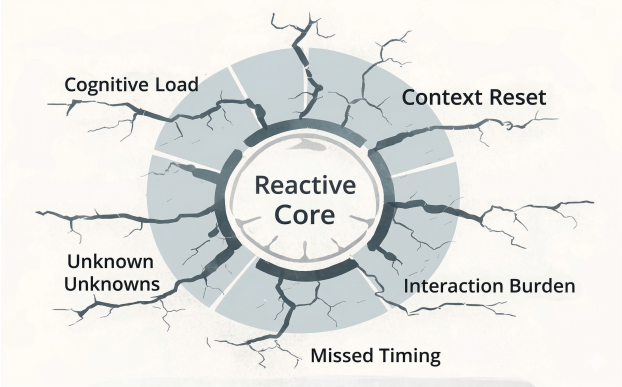


Figure 1.1: Structural failure modes

Failure 1: Cognitive Load

The first failure mode is cognitive load. Reactive AI systems create an invisible cost by placing full responsibility on the user to identify and articulate problems before receiving help. In the request-response model, the system is only responsible for producing an answer. Everything that happens before that answer falls entirely on the user. This includes realizing that something is wrong, deciding whether to interrupt the current task, and clearly formulating the problem.

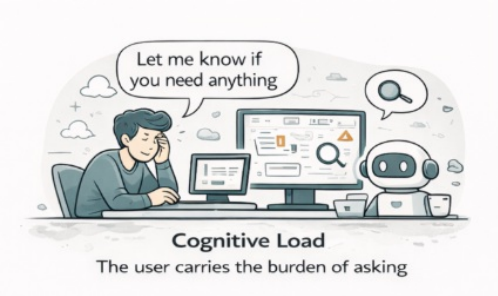


Figure 1.2: Cognitive load

A typical example of this is when a customer struggles to complete an online purchase. They encounter an error message, hesitate, then open a support chat. Before receiving any assistance, they must recognize that the error is solvable, decide whether breaking their payment flow is worth it, and explain the problem in terms the AI will understand. All the while, their shopping cart sits abandoned and their frustration mounts.

Research in cognitive psychology confirms what product teams observe in practice. Kahneman's work on cognitive effort (1973) and Johnson's research on interface friction (2010) reveal that when users are forced to expend mental energy navigating help systems rather than advancing toward their goals, both efficiency and satisfaction drop measurably. The original task suffers as working memory gets diverted to managing the system itself.

We can work towards fixing this failure mode by designing ambient agents that reverse this burden. Rather than forcing users to articulate what is wrong, the system continuously observes behavioral patterns and offers help proactively. The system tracks signals such as hesitation, repeated clicks, and abandoned forms. The user does not need to formulate the problem because the system already understands it through behavioral signals. This reduces those four cognitive steps to zero.

Failure 2: Context Reset

The second failure mode is context reset. In typical reactive systems, every new interaction begins disconnected from previous context. The system does not remember what the user did in the past or where they struggled.



Figure 1.3: Context reset

An example of this is when a user returns to a support chat three days after their last query about a billing discrepancy. Despite having exchanged ten messages with the system previously, they must re-explain account details, restate the original problem, and reconstruct the timeline of events. The system possesses all this information but keeps it hidden unless explicitly requested.

Clark and Brennan's work on grounding in communication (1991) and Gao et al.'s studies on dialogue systems (2019) demonstrate that repeated context-building carries significant social and cognitive costs. These studies reveal that users abandon interactions rather than reconstruct their narratives.

We can work towards fixing this failure mode by designing ambient agents that sustain persistent behavioral and conversational memory. The system immediately recognizes returning users, surfaces relevant history without prompting, and connects current friction to past patterns. This transforms every interaction from a cold start into the continuation of an ongoing relationship.

Failure 3: Missed Timing

The third failure mode is missed timing. Because reactive systems can only respond to requests already made, they are structurally unable to intervene at the moment when help would have maximum impact. By the time the user asks, the optimal window has usually closed.



Figure 1.4: Missed timing

An example of this is when a project management tool user spends twenty minutes trying to find

a feature, attempts multiple menu paths, and fails repeatedly. When they finally open the help chat, they receive guidance on a task they no longer have the patience to complete. The help is technically correct but emotionally useless because the moment of motivation has passed.

Karabenick and Newman's research on academic help-seeking behavior (2013) reveals that delay increases the probability of task abandonment exponentially. Similarly, Pu and Hearst's studies on user frustration in software interfaces (2018) show that intervention effectiveness drops sharply after the first thirty seconds of struggle.

We can work towards fixing this failure mode by designing ambient agents that detect struggle in real-time through behavioral signals. Cursor hesitation, repeated backtracking, and error cascade patterns are among these signals. When the user first encounters friction, we intervene exactly before frustration converts to abandonment. This transforms help from a belated reaction into a proactive presence.

Failure 4: Unknown Unknowns

The fourth failure mode is unknown unknowns. Reactive systems are limited to what users know how to ask. Because users cannot formulate requests for things they do not know exist, vast territories of unrealized value remain unexplored.

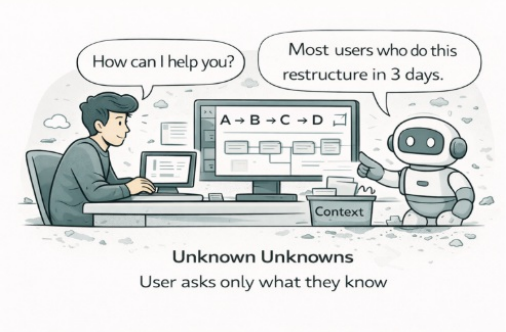


Figure 1.5: Unknown unknowns

An example of this is when a spreadsheet user manually formats hundreds of cells over hours. The user is unaware that conditional formatting automation would complete the same task in seconds. They never open the help chat because they lack the language for this problem. They experience only tedious labor, not a solvable technical challenge. The system has the solution but remains silent because no question was asked.

Norman's work on the technology paradox (2013) and Wiese et al.'s research on software learnability (2017) reveal that hidden capabilities effectively do not exist for users who lack the conceptual models to imagine them. This leads to chronic underutilization and inefficiency.

We can work towards fixing this failure mode by designing ambient agents that surface relevant capabilities based on observed behavioral patterns rather than explicit queries. The system recognizes manual, repetitive actions and proactively suggests alternatives. This transforms unknown unknowns into discovered possibilities without requiring pre-existing knowledge of the solution space.

Failure 5: Interaction Burden

The fifth failure mode is interaction burden. The mechanisms designed to deliver help become

additional sources of resistance themselves, creating a paradox where accessing help requires more effort than tolerating the original problem.

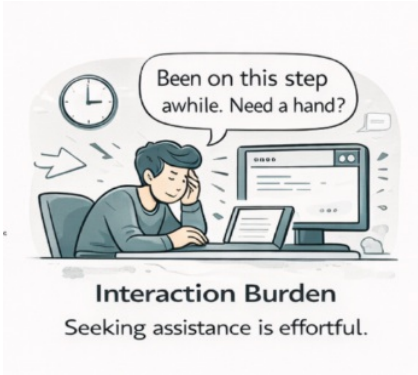


Figure 1.6: Interaction burden

An example of this is when a mobile banking user encounters a transaction error. They must navigate three menu layers, authenticate twice, wait in a chat queue, craft a problem description within character limits, and parse generic automated responses before reaching human assistance. Every step in the help-seeking chain filters users. Only the most persistent or desperate remain by the final stage.

Dixon et al.'s research on customer effort at the Corporate Executive Board (2010) and Meyer and Schwager's work on customer experience management (2007) reveal that the effort required to obtain help predicts satisfaction and loyalty more reliably than the quality of the solution itself. This makes the journey to help as critical as the help delivered.

We can work towards fixing this failure mode by designing ambient agents that eliminate the journey entirely. Help emerges within the workflow at the point of need, requiring no navigation, authentication burden, queue waiting, or formulation. This reduces help cost from a multi-step discovery to a seamless, frictionless presence.

Why These Failure Modes Are Now Solvable

For a long time, these failure modes were understood but not fixable. Continuous behavioral observation, semantic interpretation of user intent, persistent cross-session memory, and calibrated restraint each demanded custom machine learning pipelines and months of engineering work. Most teams could not afford to build all four, so they defaulted to the chat window instead.

This is not a historical coincidence. Research confirms it is still the norm. A NeurIPS 2024 study found that the vast majority of current agent systems remain reactive, waiting for explicit instruction before acting, and identified continuous observation and proactive task suggestion as the key capabilities needed to move beyond tool-like behavior (Ananthram et al., 2024). The gap between what reactive systems offer and what users actually need has been visible for years. What was missing was not awareness of the problem, but the infrastructure to address it economically.

LLMs changed that. Today, a system can interpret behavioral streams semantically, maintain structured memory across sessions, and reason about when and whether to intervene, without training a model from scratch. The frameworks are available. The economics have shifted. But

solvable does not mean simple. Research shows that proactive intervention is not automatically better than reactive. Its value depends entirely on timing. An intervention at the wrong moment damages trust as much as a missed one (Pinski & Benlian, 2025). This is why restraint is as architecturally significant as observation or memory, and why each component must be designed with the others in mind. What has been missing is a clear playbook for putting these components together. That is what this book provides.

The Architecture Will Outlast the Model

The specific technology enabling this architecture will not remain fixed. LLMs opened the door, but the framework described in this book is not bound to them. Smaller on-device models, SLMs, are already making continuous behavioral observation faster, lower-latency, and more privacy-preserving, bringing the ambient quality of these systems closer to its true meaning: present, invisible, and always on.

Beyond SLMs, a new generation of world models points toward something more fundamental. Systems like V-JEPA 2 and the world models that will follow do not learn from text alone. They learn from observation: how things move, how states change, how actions connect to outcomes. Applied to user behavior, this means an Interpretation Layer that reads hesitation, navigation rhythm, and interaction patterns the way a skilled observer reads a room, forming a working hypothesis before a single word is exchanged. The underlying models will keep changing. The need for observation, memory, and restraint will not.

Ambient Agents: Observation, Memory, Restraint

Traditional AI operates on a request-response model: the assistant waits to be asked. Ambient agents reverse this dynamic entirely. They maintain continuous awareness in the background, monitoring event streams, interpreting what they perceive, and acting proactively within governed boundaries—never waiting to be pulled, yet never overstepping their authority.

Harrison Chase, CEO of LangChain, coined the term in 2025 to describe this architecture: event-driven systems that operate in parallel, include humans when necessary, and respond within policy constraints. Enterprise implementations like Microsoft's Copilot ecosystem and Salesforce's Einstein AI demonstrate this paradigm in practice.

In its simplest form, we can define an ambient agent as a system that proactively perceives and interprets, remembers past context, and acts within a secure framework. These three core capabilities—Observation, Memory, and Restraint—distinguish ambient agents from both traditional automation and passive AI assistants.

The following sections examine each of these building blocks in turn.

Case Example: Deploy-Time Code Review

Say a user deploys a new API endpoint, the ambient agent silently perceives not just the commit, but build anomalies, bundle size changes, and staging latency. It retrieves similar past deployments from memory—three months ago, this same pattern caused a memory leak and an emergency midnight hotfix. Before the user leaves the terminal, the agent acts within its governed framework: it does not auto-rollback (it lacks that authority), but leaves a Slack message: "This deployment carries a pattern that caused issues in the past. Shall we check?"

This is a case of the ambient agent observing the present and looking at the past to put the current moment in content. By always observing, the agent can discover opportunities to anticipate the need for action and respond proactively. When we talk about the ambient agent loop, we will describe the first phase as "sensing", is truer to the active work of extracting signals to observe. In talking about observation, memory, and restraint here, we are really talking about the qualities that define an ambient agent. Let's consider the first now, observation.

1. Observation

An ambient agent continuously observes events. When the agent decides that an event is meaningful, that event becomes a signal. In other words, the event begins to acquire value from the agent's perspective.

An ambient agent loop has three core characteristics: continuous listening, using an LLM to determine what is meaningful, and triggering by observing the user interacting with the environment.

Three Principles of Observation

When an ambient agent operates in a web application, three things happen continuously during the observation process:

1. Listening: Monitoring the Event Stream: The agent works like an ear that is always open. It receives everything that comes from the system: user behaviors, API calls, commits, log entries. At this stage, the agent just listens and does not yet take action. These are called events, meaning raw occurrences whose significance has not yet been evaluated.
2. Filtering: Event or Signal? For every incoming event, the agent asks: Is this meaningful to me? Most events cannot pass this question, they are noise and pass through without action. But some events cross the threshold. For example, a repeated search query, an unexpected error rate, or a pattern not seen in a long time. These events become signals. A signal is an event that has acquired meaning. Filtering runs on rules, it is fast and computationally inexpensive.
3. Interpretation: Making Sense of the Signal: What has become a signal is sent to the LLM. But not just the signal goes, context goes too: past patterns, the user's previous behaviors, the system's current state. The LLM answers the question: What does this signal mean and does it require action? The result is a structured decision.

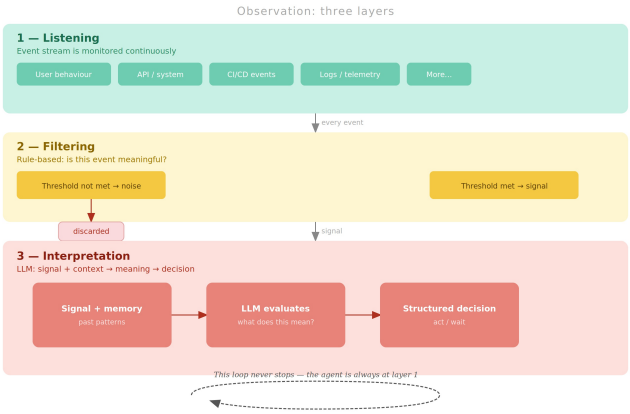


Figure 1.7: Three layers of observation

What the Agent Can Monitor

In a web application, an ambient agent can monitor:

- User behaviors: which pages were visited, how long was spent, which searches returned no results, which features were not used.
- System telemetry: API response times, error rates, endpoints under unexpected load.
- Workflow events: new commits, opened pull requests, completed deployments.
- Document streams: updated KB entries, new support tickets, documents visited repeatedly.

An Observation Example

Let's consider a web developer doing a frontend deployment. The agent continuously observes the user as they work.

First, the user enters the deployment panel. This is a routine event. The agent notes this, but does not act. Then the user refreshes the build status page three times in two minutes. This pattern crosses the threshold and becomes a signal. The agent remembers the past: this user had a build error at the same stage yesterday and the issue was resolved with a dependency update. The agent interprets: that the user has probably encountered the same error again.

The agent offers a suggestion: Your last build failed at the dependency installation. Would you like to see the updated lockfile? The user accepts, solves the problem, and completes the deployment.

What the agent ignored: forty different API calls, mouse movements, background system checks. These remained as events and passed without disturbing the user. Only the meaningful pattern became a signal, and only the signal triggered action.

Observation and Memory

An observation stripped of context is useless. Detecting that a user searched for the same term three times is good. But if you do not know that someone had the same problem last month and that a documentation page helped at that time, you cannot provide the right assistance.

Observation and memory work together. Observation identifies what is happening now; memory determines whether that observation has a history, a pattern, a known application meaning, or an action policy attached to it. Without memory, each signal is interpreted in isolation. With memory, the agent can connect the present moment to prior sessions, user-specific evidence, application context, population patterns, and intervention rules.

This brings us to the second principle: memory.

2. Memory

Observation tells the agent what is happening now. Memory helps the agent understand whether the present moment is ordinary, unusual, recurring, resolved, or worth acting on.

Traditional systems store data. Ambient agents need something more specific: context that can change future interpretation and future action. A page visit, a hesitation, or a failed task attempt is not automatically memory. Most events should disappear with the session. Memory forms only when an observation is meaningful enough to help the agent make a better decision later.

In this book, memory is not treated as a single database or a simple transcript of past interactions. It is a structured system that answers several different questions:

- What is happening in the current session?
- What has this user experienced before?
- What does the application or workflow require?
- What tends to happen across users?
- What should the agent do, defer, suppress, or avoid?

Those questions matter because an ambient agent must distinguish between different kinds of context. The state of the current session is not the same as a durable belief about the user. A past struggle is not the same as a current struggle. A population-level friction point is not proof that this particular user is confused. A policy about when to intervene is not the same thing as a memory of what happened.

For that reason, the memory architecture used later in this book separates memory across four implementation objects. The Session State Model holds the live context of the current session. The User Model preserves selected evidence and revisable beliefs about an individual user. The Application Domain Model stores knowledge about the application, its workflows, and population-level behavior. The Procedural Policy Layer governs how remembered context should influence action.

At this stage, the important point is simple: memory gives the agent continuity. It lets the agent notice that a user has returned to a workflow they abandoned last week, that a difficulty has already been resolved, that a screen is a known friction point for many users, or that the user has explicitly asked not to receive a certain kind of help.

Consider a developer returning to a deployment panel after a failed release. A reactive assistant sees only the current question. An ambient agent can recognize the larger situation: the current route, the previous failure, the user's past resolution path, the known risks in this workflow, and the policies that determine whether an intervention is appropriate. It may decide to offer a quiet suggestion. It may decide to wait. It may decide that the user has already mastered this issue and remain silent.

That selectivity is crucial. A useful memory system does not preserve everything. It preserves what can improve future interpretation, future restraint, or future assistance. Memory makes the agent more capable, but only when it remains typed, selective, revisable, and accountable.

Chapter 5 returns to this topic in detail. There, we will make the memory architecture operational: how session evidence becomes durable memory, how user beliefs are formed and corrected, how application context grounds interpretation, how population patterns inform but do not override personal evidence, and how procedural memory governs restraint.

3. Restraint

In reactive systems, every input triggers an immediate output. But an ambient agent operates under a continuous stream of signals, and converting every observation into an action overwhelms the user. Restraint is the filtering mechanism that prevents this.

Restraint rests on three core justifications:

1. Attention economics: excessive notifications cause fatigue and erode signal quality
2. Autonomy: users must retain a sense of control over their own workflow
3. Trust-building: users learn to trust proactive systems when interventions are relevant, explainable, and appropriately timed

Restraint answers four core questions:

1. Is this signal truly worth acting on?
2. Is this the right moment to intervene
3. With what mechanism should I intervene
4. Is this decision within my authority, or should I defer to a human?

These questions move restraint from a general design principle into an operational system. The agent cannot rely on good judgment in the abstract; it needs mechanisms that decide which signals matter, when intervention is appropriate, how often the user should be interrupted, what contextual state should suppress action, what data should be retained, and which actions require human approval.

The following layers translate restraint into concrete system behavior:

Layer	Mechanism	Example
Signal Filtering	The system filters out sub-threshold signals and applies relevance scoring to prioritize what warrants attention.	Alert only when 7-day average exceeds 15%
Temporal Gating	The system manages intervention timing based on user availability and contextual appropriateness.	Suppress notifications during meetings
Rate Limiting	The system caps the frequency of proactive messages and batches low-priority communications.	Max 3 proactive suggestions per day
Context Awareness	The system monitors user state indicators including do-not-disturb settings, active sessions, and estimated cognitive load.	All alerts deferred during sleep hours
Privacy / Retention	The system collects only necessary data and enforces automatic deletion according to predefined retention schedules.	Location history auto-deleted after 30 days
HITL Boundary	The system defines boundaries for autonomous action, requiring human approval when decisions exceed confidence thresholds or involve sensitive operations.	Financial actions always require approval

Table 1.1: The Layers of Restraint

The Dance of Three: Observation, Memory, Restraint

Observation, memory, and restraint do not operate as separate stages. They work together

continuously.

Observation gives the agent awareness of the present moment. It detects events, filters noise, and identifies signals that may matter. Memory gives those signals context by connecting them to prior sessions, user-specific evidence, application knowledge, population patterns, and procedural rules. Restraint determines whether that understanding should become an intervention, a deferred action, a quiet update, or no action at all.

Consider a system monitoring a new API endpoint deployment. Observation detects consistent signals across several sources: a merged pull request, a deployment event, a Jira ticket, and an updated architecture note. Memory connects those signals to the application domain: the endpoint belongs to the payments workflow, affects several dependent services, and resembles a prior deployment that required documentation updates. Restraint then governs the response. If the user is in a meeting, the agent waits. If the daily notification limit has already been reached, the update may be queued. If the action is additive and low-risk, the agent may update the knowledge base automatically. If the action would overwrite existing documentation or affect a production workflow, it should defer to a human.

The resulting behavior is not simply proactive. It is selective. The agent does not act because something happened. It acts because the signal is meaningful, the context supports action, and the intervention is appropriate within the system's boundaries.

Without observation, the agent is blind. Without memory, it lacks context. Without restraint, it becomes noisy, intrusive, or unsafe. Together, these three principles make ambient agents useful without making them overwhelming.

These principles make ambient agents powerful, but they also make them demanding. Continuous observation requires access to behavioral signals. Memory requires a way to preserve and retrieve context responsibly. Restraint requires policies that determine when action is justified and when silence is better. Together, these capabilities introduce architectural, operational, and trust considerations that not every product needs.

Before building an ambient agent, then, the first question is not how to implement one. The first question is whether the application has a problem that ambient agency is uniquely suited to solve.

Is Your Application a Good Candidate for an Ambient Agent?

Not every application needs continuous behavioral intelligence. Before investing in the infrastructure and complexity of an ambient agent, work through the following questions.

Do you need to understand the full spectrum of user behavior?

Traditional analytics capture completed actions: clicks, page views, conversions. They tell you what users did but miss everything between those discrete events: hesitations before a click, cursor movements that suggest uncertainty, scroll patterns that indicate skimming versus reading.

If completed actions tell you enough, traditional analytics may serve you. If you need to

understand the texture of user behavior, the moments of confusion and consideration that precede action, you need the continuous observation that ambient sensing provides.

Do you need this understanding in real time?

Rich behavioral data can be collected and analyzed after the fact. Session recordings and detailed event logs let you study user behavior without acting on it in the moment.

If batch analysis serves your needs, you can capture full behavioral detail without the complexity of real-time interpretation using session recording and offline behavioral analysis.

Do you need to take action on what you learn?

The next question is what kind of action the system needs to take once it understands the behavior.

In some applications, the goal is only to learn from patterns across users. The system may detect that many users hesitate on the same screen, abandon the same step, or ignore the same feature. That information can guide product decisions, documentation updates, or workflow redesigns, but the product still behaves the same way for every user. In this case, you need observation and interpretation, but not the full ambient agent loop.

Other applications need user-level adaptation without direct interruption. The system may quietly adjust the experience based on what it has learned about a specific user. It might reorder guidance, simplify a workflow, suppress an unnecessary explanation, or surface a more relevant default. The user does not experience this as a conversation with an agent. The interface simply becomes more responsive to their behavior. In this case, you need observation, interpretation, and decision-making, but action is expressed through subtle environmental change.

The fullest version of ambient agency appears when the system needs to intervene directly. Here, the agent surfaces itself to the user with a suggestion, warning, clarification, or offer of help. This is appropriate when the moment is time-sensitive, the risk of abandonment is high, or the user would benefit from guidance before they explicitly ask for it. In this case, you need the complete loop: sensing, interpretation, decision, action, and learning.

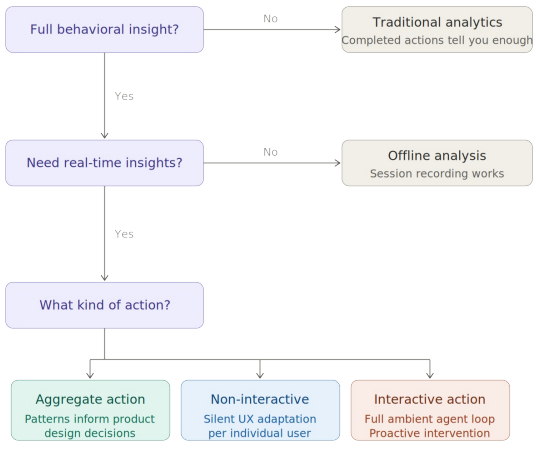


Figure 1.8: Ambient Agent Suitability Decision Tree

Summary

This chapter established the core argument of the book: reactive AI interfaces fail not because they are poorly built, but because they are structurally unable to meet users where they are. We examined five structural failure modes (cognitive load, context reset, missed timing, unknown unknowns, and interaction burden), each stemming from the same root cause: help arrives only when the user asks for it. We then introduced the three behavioral design principles that ambient agents use to close this gap: Observation, which continuously monitors and interprets behavioral signals; Memory, which accumulates context across sessions and distills it into actionable knowledge; and Restraint, which governs when and how to act so that proactive assistance builds trust rather than eroding it.

The failure modes described here are not edge cases. They are the default outcome of every chat-first AI integration shipped without proactive architecture. Understanding them by name, and knowing the principles that address each one, is what separates an ambient agent from a more sophisticated chat window. The shift from reactive to ambient is not primarily a technical upgrade; it is a design philosophy, and this chapter is the lens through which the rest of the book should be read.

The rest of this book turns that conceptual model into an implementation path. Observation, memory, and restraint explain what makes ambient agents different from reactive AI systems, but those principles need concrete machinery behind them: a loop for processing behavioral signals, persistence for session and long-term context, a decision framework for choosing whether to act, a collaboration layer for delivering help in the user experience, and voice infrastructure for multimodal interaction. The next chapter begins that translation from principle to architecture by introducing the core technologies that make ambient agents practical to build.

2

Core Technologies: Building Blocks for Ambient Agents

In this chapter, we'll explore the core building blocks for implementing an ambient agent.

First, there needs to be some sort of agent framework, even if we end up building our own, because each session requires a dedicated processing loop. At the same time, aggregate processing will be desirable for feedback and adaptive learning. Second, there will also be a need to save state, both short term and longer term, so one or more sources of persistence will be wanted. Third, we'll want some sort of framework in order to readily materialize the effects of proactive assistance in the UX with minimal latency.

We could stop here, and have a viable ambient UX agent. However, multi-modal functionality is particularly useful when it comes to ambient agents, because it allows users to speak to an agent in conjunction to demonstrating what they are trying to do. And while we could just provide speech-to-text (STT), just as we are focused on preserving the activity that proceeds the actions traditionally recognized by and shared with backend APIs, we want to preserve the expressions provided in the medium of voice that get lost when we flatten to text.

Therefore, we also to be able to take proactive action on the signals manifest in speech as well. We're adding two additional technologies, the first of which enables us to efficiently stream speech to the backend and convert it to text, and the second, which allows to talk back to the user, converting the agent response into realistic voice.

This chapter introduces the five core technologies that, together, make all of that possible: Neo4j for memory and context, LangGraph for the AI-powered decision graph, CopilotKit for the human-agent collaboration layer, Deepgram for voice input, and ElevenLabs for voice output.

By the end of this chapter, you'll understand how these technologies work together to build an ambient agent. Here are the sections we'll traverse along theway:

- The Five-Phase Loop
- Neo4j: Memory and Application Context
- LangGraph: The Decision Spine
- CopilotKit and AG-UI: The Human-Agent Collaboration Layer
- Deepgram: Voice as a Sensing Channel
- ElevenLabs: Voice as an Act Channel
- Voice Orchestration: Turn-Taking and Interruption

The Ambient Agent Five-Phase Loop

Before we examine each technology individually, we need a frame for how they relate to one another. That frame is the ambient agent loop. The ambient agent system we will discuss cycles through five phases: Sense – Interpret – Decide – Act – Learn. Together they form a closed feedback loop that runs continuously as long as the user is present, which makes it session-based.

> NOTE
>
> An ambient agent could be a long-running process that does things in the background on behalf of the user or the system, regardless of an active user session. This could lead to an alternative loop, such as Observe – Decide – Act. Any feedback received by the system could be taken into account during Decide, in this case.

Let's take a quick look at each stage of our session-based loop.

Sense

Our system needs to observe what the user is doing. Not raw keystrokes and coordinates, but derived behavioral signals: a hesitation before a difficult field, a scroll reversal, a voice question half-formed and abandoned. *Chapters 3* covers the sensing phase in depth, but for now the key point is that sensing produces a structured behavioral report called an episode, and forwards it to interpretation, which is covered in *Chapter 4*. Raw signals are discarded; the episode is the sole output of sensing. Typically we'd call this phase Observe, but because of the work we are doing to make things observable, Sense is closer to the truth.

Interpret

The system explains what the behavior means. A sequence of hesitations on a form field might mean confusion; the same pattern at a different point in the session, after several successful completions, might mean nothing at all. Interpretation is where sensing signals acquire meaning by being placed in context. It could be the context of the current screen, the task in progress, or anything the system has observed about this user so far.

Decide

Given an interpreted behavioral state, the system determines whether to act, how urgently, and in what modality. The decision is conservative by design: an ambient agent that interrupts unnecessarily is worse than one that stays quiet. Most episodes end with a decision to do nothing. We'll see how this works in *Chapters 6* and *7*.

Act

When the system decides to intervene, it builds a response and delivers it. That response might be a message in a chat panel, a field highlighted in the UI, a navigation shortcut, or a spoken sentence. The act phase includes execution because the agent's response must actually reach the user to provide assistance. We'll see how this works in *Chapters 6* and *7*.

Learn

Over time, the system accumulates evidence about which interventions helped and which didn't. That evidence feeds back into the decide phase, adjusting thresholds and shaping when and how

future interventions happen. Learning closes the loop. We'll see how this works in *Chapters 6* and *7*.

The Loop

Here's how the five technologies we'll examine map to the five phases:

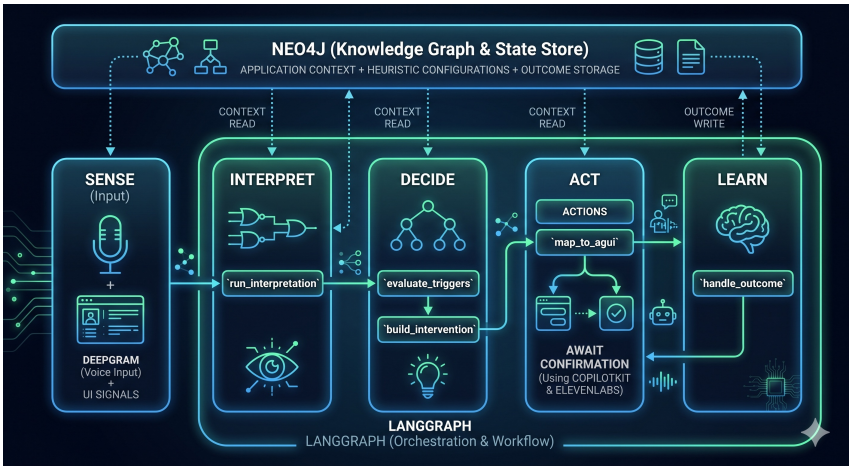


Figure 2.1: 5 Phases Mapped to 5 Technologies

Neo4j threads through multiple phases because it serves three distinct roles:

1. It supplies the application knowledge that makes interpretation meaningful.
2. It stores the heuristic thresholds that govern decisions.
3. It persists the outcome data that feeds learning.

LangGraph spans the remaining phases, where its graph nodes implement interpretation, decision, act, and outcome tracking as a single persistent, resumable workflow. That span is what makes it foundational: a stateless function could handle a single decision, but only a stateful graph can maintain session memory across dozens of episodes, suspend mid-intervention to wait for user confirmation, and feed outcome signals back into the same session context that produced the intervention.

Deepgram sits squarely in the Sense phase, while ElevenLabs resides in the Act phase. CopilotKit is the mechanism we use to connect the front-end to the ambient agent, which is important for the Sense, Interpret, Act and Learn phases.

Now let's examine each of these technologies in turn.

Neo4j: Memory and Application Context

When a user hesitates on a form field for eight seconds, what does that mean? The answer depends entirely on which field. Eight seconds on a postal code field probably signals a minor distraction, for example, the user may have gone to look up the extra four digits for their zip code. Eight seconds on a "debt-to-income ratio" field is a different matter: it is a technical concept that many mortgage applicants may have never encountered, so hesitation there is a strong signal that guidance would help.

Enabling this distinction – that the same behavioral pattern can mean something different depending on where it occurs – is where Neo4J comes in. Neo4j is a graph database, and for

ambient agents, it plays the role of institutional knowledge: a structured representation of the application's layout, the tasks users are trying to accomplish, the steps those tasks require, and critically, where users tend to get stuck.

The Application Context Graph

The graph that supports an ambient agent is organized around a small set of node types with clear relationships between them. A `Domain` node represents the overall application, while `Screen` nodes represent the distinct views a user can visit, each annotated with metadata that the interpretation layer needs at runtime:

```
interface ScreenNode {
  route_pattern:       string;   // e.g., '/apply/mortgage/income'
  label:               string;   // 'Monthly Income Entry'
  screen_type:         'form' | 'review' | 'dashboard' | 'onboarding' | 'informational';
  typical_duration_ms: number;
  struggle_rate:       number;   // historical: fraction of sessions with friction here
  embedding?:          number[]; // optional vector for semantic screen matching
}
```

Moving on, `Task` nodes represent the goals users pursue: "complete income verification," "upload supporting documents," "review loan terms." `TaskStep` nodes break tasks into ordered stages. And `DifficultyPoint` nodes mark the specific fields or interactions that historical data has shown to be trouble spots, annotated with typical failure modes.

Here we see how a mapping between a `Screen` node and a `DifficultyPoint` node through the `HAS_DIFFICULTY_POINT` relationship. And by see, I mean we can see through the user of arrows, how the graph flows, how it is traversed. In a graph database, that traversal is expressed naturally. In a relational model, it requires joins across multiple tables that grow in complexity as the schema evolves:

```
(:Screen { route_pattern: '/apply/mortgage/income' })
  -[:HAS_DIFFICULTY_POINT]->
(:DifficultyPoint {
  field_id:     'dti_ratio',
  label:        'Debt-to-Income Ratio',
  struggle_rate: 0.34,
  failure_modes: ['definition_unknown', 'calculation_unclear']
})
```

When the interpretation layer processes an episode, it queries this graph to answer a question the episode alone cannot answer: "Is the user's struggle consistent with known difficulty at this point?" Thinking about the debt-to-income ratio again, a hesitation on the field is not generic confusion. In our example, it is a specific, well-documented pattern that the system has seen in 34% of sessions at this step. That knowledge changes what the agent says and how urgently it says it.

Graph relationships are the reason Neo4j is the right tool here rather than a relational or document database. The question "Which tasks involve this screen, and which steps of those tasks are users currently on?" requires a traversal across four node types with three relationship types.

Heuristic Configuration

Neo4j serves a second role that is easy to overlook but becomes critical as the system matures: it stores the numeric thresholds that govern the agent's behavior. An ambient agent's decision engine evaluates conditions like "Is friction above threshold?" and "Have retry loops exceeded the limit?" In early development, these thresholds are hardcoded constants, or better yet, initial values.

But different applications, and different customer deployments of the same application, will have radically different baselines. A healthcare enrollment workflow attracts users who are anxious and careful; they produce high friction signals even when they are not actually struggling. A developer tooling workflow attracts users who move fast and backtrack deliberately. Applying the same thresholds to both will produce the wrong answer in one of them.

The solution is to externalize thresholds as graph nodes, like so:

```
(:HeuristicConfig {
  scope:      'screen_type',
  scope_key:  'form',
  key:        'friction_high_threshold',
  value:      0.65,
  updated_at: datetime(),
  updated_by: 'seed'
})
```

The `updated_by` field is what makes this interesting. When the system is first deployed, values are written by `'seed'`, a startup script that loads sensible defaults. As the earn phase accumulates outcome data and an optimizer (essentially an optimization function) adjusts thresholds to reduce false positives, it writes values with `updated_by: 'optimizer'`, which is to differentiate updates by a human, where `updated_by` is set to `'manual'`. The seeder is designed to never overwrite optimizer or manual values, so the learning cycle is non-destructive — automated learning cannot erase deliberate operator decisions.

At every decision point in the agent code, threshold resolution follows the same pattern:

```
// The hardcoded constant is kept as a safe fallback.
const FRICTION_HIGH = 0.65;

// At runtime, the graph value takes precedence if available.
const frictionHigh =
  appCtx.getHeuristicConfig('friction_high_threshold') ?? FRICTION_HIGH;

if (sessionFriction >=frictionHigh&&progress.forward_momentum<= 0) {
  candidates.push({ type: 'guide', priority: 0.75, rationale: '...' });
}
```

This pattern (hardcoded default as fallback, graph value as override) means the system behaves safely even when Neo4j is unreachable, and learns continuously when it is.

Outcome Storage

The third role Neo4j plays is persistence for the learn loop. When an agent intervenes, the system records the result: did friction drop afterward? Did the user continue, complete the task, abandon it, or show no measurable change? This intervention outcome record is stored in Neo4j and becomes the training signal for threshold adjustment:

```
interface InterventionOutcome {
```

```
  intervention_id:        string;
  session_id:             string;
  intervention_type:       'inform' | 'guide', 'assist' | 'confirm' | 'escalate';
  result:                 'completed' | 'continued' | 'abandoned' | 'no_change';
  delta_friction:         number;
  delta_progress:         number;
  episodes_to_resolution: number;
  resolved_at:            number;
}
```

Let's break this code down:

Result Type	Result Meaning
`abandoned`	User left and returned after long gap
`completed`	Task reached completion status
`continued`	Friction dropped significantly within the window
`no_change`	Episode limit reached without resolution

Table 2.1

We will return to the learn phase in *Chapters 6* and *7*. For now, the key point is that Neo4j is not just a query-time dependency — it is the system's memory across sessions, customers, and deployments. Application context and heuristic config can be loaded into memory at startup; outcome data accumulates continuously. Over time the graph becomes a record of what the system has tried and what worked.

LangGraph: The Decision Spine

Every ambient agent needs to answer a fundamental question each time a new behavioral episode arrives: "Given everything I know about this user right now, should I act?" That question sounds simple, but it is not. The answer depends on session history that may span dozens of episodes. It may require human confirmation before acting. If the user accepts the intervention, the agent needs to remember that it intervened and watch for the outcome. If the user dismisses the intervention three times in a row, the agent should suppress future interventions for a while.

All of this requires stateful, persistent, resumable execution. That is exactly what LangGraph provides.

Why a Graph?

The earliest agent frameworks represented agent behavior as a linear chain: process input > call tools > return output. Chains work well for request-response interactions, however they fail for ambient agents because ambient agents are not request-response systems. They run continuously. Their behavior at step N depends on what happened at steps N-1 through N-10. They sometimes need to pause and wait for a signal from the user before they can continue. Chains have no mechanism for any of this.

LangGraph represents agent behavior as a directed graph of nodes and edges, compiled with a checkpointer that persists state between invocations. When a new episode arrives, the graph picks up exactly where it left off for that session. State is not rebuilt from scratch. Instead, it is

loaded from the checkpoint, updated, and checkpointed again at the end of the run.

Think of it like a persistent workflow engine: the graph describes what can happen and in what order; the checkpointer remembers where each session is in that workflow and what it has observed so far.

State as the Agent's Memory

Before a LangGraph graph can be defined, its state type must be defined. State is the complete memory of the agent for a given session — everything a node can read and write. For an ambient agent, state includes the most recent user state model (the output of the interpretation layer), the history of past interventions as a message thread, the current interaction modality, suppression signals accumulated from prior dismissed interventions, and whatever intervention is currently awaiting user confirmation:

```
export const AgentStateAnnotation = Annotation.Root({
  messages:
    Annotation<BaseMessage[]>({
      reducer: messagesStateReducer, default: () => []
    }),
  sessionId:
    Annotation<string>({
      reducer: (_, b) => b, default: () => ''
    }),
  modality:
    Annotation<'ui' | 'voice' | 'mixed'>({
      reducer: (_, b) => b, default: () => 'ui'
    }),
  usm:
    Annotation<UserStateModel | null>({
      reducer: (_, b) => b, default: () => null
    }),
  fusedEpisodes:
    Annotation<FusedEpisode[]>({
      reducer: fusedEpisodesReducer, default: () => []
    }),
  adjustmentState:
    Annotation<AdjustmentState>({
      reducer: (_, b) => b, default: () => ({ suppressionStreak: 0, confidencePenalty: 0 })
    }),
  pendingInterventionObject:
    Annotation<Intervention | null>({
      reducer: (_, b) => b, default: () => null
    }),
  decisionOutput:
    Annotation<DecisionOutput | null>({
      reducer: (_, b) => b, default: () => null
    }),
});
```

Let's break this code down:

Field	Description
`messages`	Holds one `AIMessage` per intervention the agent has produced, and one `HumanMessage` per outcome signal the user has returned; with the help of the `MessageState` reducer, a full conversational history is accumulated with deduplication

`usm`	Corresponds to the User State Model, which gets written by the `run_interpretation` node and read by the `evaluate_triggers` and `build_intervention` nodes; only the latest value is retained
`fusedEpisodes`	Is the behavioral history (time) window, which keeps the 15 most recent fused UI and voice episodes
`adjustmentState`	Carries the suppression streak and confidence penalty accumulated from prior dismissed interventions, so the agent becomes progressively more restrained if the user keeps refusing its suggestions
`pendingInterventionObject`	Holds whatever intervention is currently waiting on user confirmation
`decisionOutput`	Consists of the ephemeral routing signal produced by `evaluate_triggers` for the current run only

Table 2.2

Most fields simply take the latest value with `(_, b) => b`. This declarative approach means there is no boilerplate for state merging. The framework handles it.

The Graph Topology

The graph itself is assembled by connecting nodes with edges. Conditional edges implement decision points, which are branches where the graph routes differently depending on the current state. The topology for an ambient agent that handles both proactive interventions and human-in-the-loop confirmation could look like this:

```
__start__
  -> ingest_episode          (parse episode, derive UI and voice slices, fuse them)
  -> run_interpretation      (run pipeline, write updated usm to state)
  -> evaluate_triggers       (consult usm + Neo4j thresholds -> decisionOutput)
  -> conditional: should_intervene?
  ├─ false -> __end__
  └─ true
       -> build_intervention  (call LLM -> Intervention object)
       -> map_to_agui         (format as AG-UI events -> stream to browser)
       -> conditional: requires_confirmation?
       ├─ false -> __end__
       └─ true
            -> await_confirmation  <- graph suspends here (interrupt)
            -> handle_outcome      (record result, update adjustmentState)
            -> __end__
```

In code, this can be assembled using LangGraph's builder API:

```
const workflow = new StateGraph(AgentStateAnnotation)
  .addNode('ingest_episode',    ingestEpisodeNode)
  .addNode('run_interpretation', makeRunInterpretationNode(adapter))
  .addNode('evaluate_triggers', evaluateTriggersNode)
  .addNode('build_intervention', buildInterventionNode)
  .addNode('map_to_agui',       mapToAGUINode)
  .addNode('await_confirmation', awaitConfirmationNode)
  .addNode('handle_outcome',    handleOutcomeNode)
  .addEdge(START, 'ingest_episode')
  .addEdge('ingest_episode', 'run_interpretation')
```

```
  .addEdge('run_interpretation', 'evaluate_triggers')
  .addConditionalEdges('evaluate_triggers', shouldIntervene, {
    build_intervention: 'build_intervention',
    [END]: END,
  })
  .addEdge('build_intervention', 'map_to_agui')
  .addConditionalEdges('map_to_agui', requiresConfirmation, {
    await_confirmation: 'await_confirmation',
    [END]: END,
  })
  .addEdge('await_confirmation', 'handle_outcome')
  .addEdge('handle_outcome', END);

const graph = workflow.compile({
  checkpointer,
  interruptAfter: ['await_confirmation'],
});
```

The graph's policy is visible at a glance. Most episodes flow through `ingest_episode` > `run_interpretation` > `evaluate_triggers` > `END`. The episodes are processed and silently discarded. When `shouldIntervene` returns true, the graph continues to `build_intervention`, formats the response for the browser, and either ends immediately or suspends at `await_confirmation` if the intervention requires the user to respond before proceeding.

The `interruptAfter: ['await_confirmation']` configuration is how LangGraph implements human-in-the-loop. After executing the `await_confirmation` node, the graph suspends and persists its current state to the checkpointer. It does not resume until the application calls it again with the user's response as input. This is not polling because the graph is truly suspended, consuming no resources, until it receives the signal to continue.

The conditional routing functions are plain TypeScript that reads state and returns the name of the next node:

```
function shouldIntervene(state: AgentState): 'build_intervention' | typeof END {
  return state.decisionOutput?.should_intervene ? 'build_intervention' : END;
}
```

The key insight about LangGraph is that the graph is the agent's policy. The routing logic is not buried in if-statements scattered across service classes; it is explicit, inspectable, and modifiable. To change when the agent intervenes, you change the graph topology or the routing functions. To add a new kind of intervention, you add a node and connect it. To support a new confirmation flow, you add a new conditional edge. The graph makes the policy legible in a way that procedural code cannot.

CopilotKit and AG-UI: The Human-Agent Collaboration Layer

The agent has decided to act. It has built an intervention. Now it needs to deliver it — not as a response to a question the user asked, but as a proactive message that appears in the right place, at the right time, in the right form. And in some cases it needs to do more than display text: it needs to move the user's attention to a specific field, highlight an element, or trigger a navigation to a help resource.

This is the problem CopilotKit solves. CopilotKit is a framework for building human-agent

collaboration interfaces. At its core, it manages the channel between a running agent and a user's browser — including the ability for the agent to execute actions in the browser on the user's behalf.

The AG-UI Protocol

Between the agent and the browser, structured events flow over a Server-Sent Events stream. This stream carries a typed protocol called AG-UI: a standard event format that CopilotKit and other compatible frontends consume. AG-UI events represent everything that can happen in an agent turn: a text message starting, chunks of that message arriving as they are generated, a tool call the agent wants to make, and the result of that tool call.

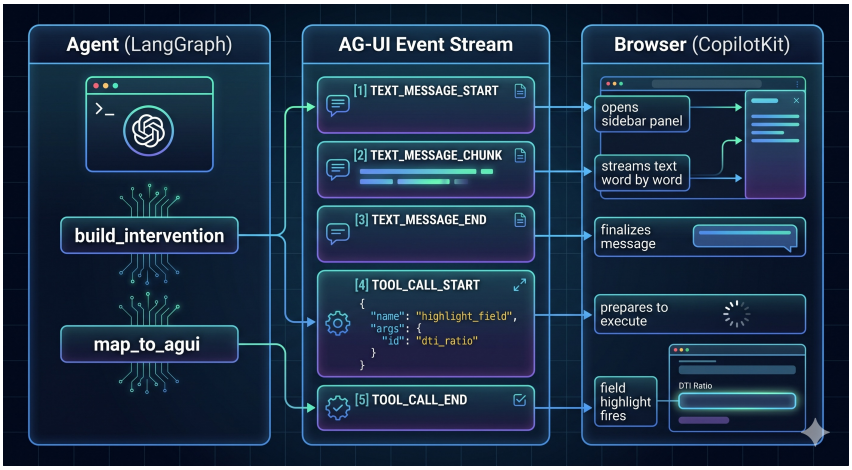


Figure 2.2: AG-UI Event Flow

What makes this powerful is the tool call mechanism. When the agent emits a `TOOL_CALL_START` event with a tool name and arguments, CopilotKit executes the corresponding handler registered in the browser. The agent has no direct access to the DOM, but thanks to CopilotKit's tool protocol, it doesn't need to. It issues instructions, and the browser executes them.

Frontend Tools with useCopilotAction

Tools are registered in the browser using the `useCopilotAction` hook. Tools are simply user-defined functions. In this case, functions that will run within the browser to realize a given action. Each tool has a name, a description the agent's LLM uses to decide when to call it, typed parameters, and a handler function that runs in the browser when the agent invokes the tool. Because handlers run in React component scope, they have access to the application's router, local state, and the full DOM.

Here we have a function that highlights a UI element to draw a user's attention to it; a function to navigate to a different section of the application; and a function to open a help dialog.

```
useCopilotAction({
  name: 'highlight_field',
  description: 'Temporarily highlight a UI element to draw user attention',
  parameters: [
    { name: 'id',         type: 'string', required: true  },
    { name: 'durationMs', type: 'number', required: false },
  ],
  handler: ({ id, durationMs }) => {
```

```
    highlightField(id, durationMs ?? 3000);
  },
});

useCopilotAction({
  name: 'navigate_to',
  description: 'Navigate to a different route in the application',
  parameters: [
    { name: 'path', type: 'string', required: true },
  ],
  handler: ({ path }) => {
    router.push(path);
  },
});

useCopilotAction({
  name: 'open_help_panel',
  description: 'Open the contextual help panel for the current view',
  parameters: [],
  handler: () => {
    window.dispatchEvent(new CustomEvent('cogstream:open-help-panel'));
  },
});
```

From the agent's perspective, these are API calls it makes by returning structured output from a LangGraph node. The user sees the result through browser effects: a field lit up, the page navigated, a panel appeared. The agent's reach extends into the DOM.

The Provider Pattern

CopilotKit is integrated via a provider component that wraps the application. The provider establishes the connection to the backend agent, manages the AG-UI SSE stream, and makes context available throughout the React component tree. Tools registered with `useCopilotAction` anywhere inside the provider are automatically available for the agent to call:

```
export function App() {
  return (
    <CopilotKit runtimeUrl="/api/copilotkit" agent="ambient-guide">
      <MortgageApplicationForm />
    </CopilotKit>
  );
}
```

The `runtimeUrl` points to a backend endpoint that proxies agent requests and manages the LangGraph session. CopilotKit handles the SSE connection lifecycle, reconnection, and the rendering of streaming messages, none of which the application code needs to worry about.

The key insight about CopilotKit is that it collapses the gap between agent decision and the resulting browser change. Without CopilotKit, you would need to build your own protocol: a custom SSE format, a custom tool invocation layer with type checking and error handling, a custom message streaming and rendering system. With CopilotKit, the agent's decisions propagate to the browser through a standardized channel that both sides understand natively.

Deepgram: Voice as a Sensing Channel

For our ambient agent, and with a nod to the oldies Alexa and Siri, we won't make voice input an afterthought. It is, in many ways, the most direct signal available, so we'll make use of it. A user

who says "wait, what does debt-to-income mean?" is expressing intent, confusion, and a specific question simultaneously, all in one utterance, before any UI interaction has occurred. Voice closes the gap between what the user is thinking and what the system can observe. Of course, we don't want to turn this into surveillance, so this feature needs to be readily activated through an intercom feature.

Challenge of voice processing

The challenge is that useful voice processing for ambient UX agents has requirements that traditional speech-to-text approaches don't satisfy. Think about it: we wanted to capture the behavioral happenings that don't usually make it to the backend via a recognizable event. Now imagine all that can get lost when immediately transferring speech to text and sending it up for processing. We lose all the audible nuance. So we need streaming transcription, partial transcripts arriving as the user speaks, so that the system can begin reasoning about intent before the utterance ends. And we need a model that performs well on conversational speech in real-world conditions, where users hesitate, restart sentences, and speak at varying volumes.

Deepgram's Nova-2 model is built for this use case. Nova-2 supports streaming transcription via a WebSocket connection, produces interim results on each audio frame, and applies smart formatting to handle common patterns like phone numbers, dollar amounts, and percentages that appear frequently in form-filling conversations.

The Backend Proxy Pattern

Browser applications cannot open WebSocket connections to Deepgram directly, because doing so would expose the API key in client-side code. The solution is a backend proxy: the browser streams audio to a server route, which opens the Deepgram WebSocket and forwards transcripts back to the browser as a newline-delimited JSON stream (NDJSON). At a high-level it shows Deepgram setup, transcript serialization, response streaming, and browser audio forwarding:

```
// POST /api/deepgram
export async function POST(req: NextRequest): Promise<Response> {
  const apiKey = getDeepgramApiKey();
  const connection = createDeepgramConnection(apiKey);
  const transcriptStream = createTranscriptStream();

  forwardDeepgramTranscripts(connection, transcriptStream.writer);
  forwardBrowserAudio(req, connection);

  return createNdjsonResponse(transcriptStream.readable);
}
```

First, the function retrieves tries to retrieve the API key from the environment variable:

```
function getDeepgramApiKey(): string {
  const apiKey = process.env.DEEPGRAM_API_KEY;
  if (!apiKey) {
    throw new Error('Missing DEEPGRAM_API_KEY');
  }
  return apiKey;
}
```

If all goes well, it next establishes a WebSocket connection to Deepgram, by creating a client and then listening and transcoding audio:

```
function createDeepgramConnection(apiKey: string) {
  const deepgram = createClient(apiKey);

  return deepgram.listen.live({
    model: 'nova-2',
    language: 'en',
    smart_format: true,
    interim_results: true,
    encoding: 'webm-opus',
    sample_rate: 48000,
  });
}
```

Then the transcoding is accomplished via a transform stream:

```
function createTranscriptStream() {
  const stream = new TransformStream<Uint8Array, Uint8Array>();

  return {
    readable: stream.readable,
    writer: stream.writable.getWriter(),
  };
}
```

Next, `ForwardDeepgramTranscripts` listens for transcript events from Deepgram and writes them to the browser response stream. It converts Deepgram's provider-specific event into the application's simpler voice-event format, usually `speech_partial` or `speech_final`:

```
function forwardDeepgramTranscripts(
  connection: LiveConnection,
  writer: WritableStreamDefaultWriter<Uint8Array>
): void {
  connection.on(LiveTranscriptionEvents.Transcript, (data) => {
    const transcript =
      data.channel?.alternatives?.[0]?.transcript?.trim() ?? '';

    if (!transcript) return;

    const event = {
      type: data.is_final ? 'speech_final' : 'speech_partial',
      timestamp: Date.now(),
      transcript,
    };

    void writer.write(new TextEncoder().encode(JSON.stringify(event) + '\n'));
  });

  connection.on(LiveTranscriptionEvents.Close, () => {
    void writer.close();
  });

  connection.on(LiveTranscriptionEvents.Error, (error) => {
    void writer.abort(error);
  });
}
```

`forwardBrowserAudio` does what its name says. It waits until the Deepgram WebSocket is open, then starts forwarding the browser's incoming audio stream to Deepgram. It does not read the audio itself. It only connects the Deepgram Open event to the lower-level audio pumping function:

```
function forwardBrowserAudio(
  req: NextRequest,
  connection: LiveConnection
): void {
  connection.on(LiveTranscriptionEvents.Open, () => {
    void pumpRequestBodyToDeepgram(req, connection);
  });
}
```

`pumpRequestBodyToDeepgram` reads audio chunks from the browser's streaming POST request body and sends each chunk into the Deepgram WebSocket. This is the actual loop that moves audio from the browser to Deepgram:

```
async function pumpRequestBodyToDeepgram(
  req: NextRequest,
  connection: LiveConnection
): Promise<void> {
  if (!req.body) {
    connection.requestClose();
    return;
  }

  const reader = req.body.getReader();

  try {
    while (true) {
      const { done, value } = await reader.read();

      if (done) {
        connection.requestClose();
        return;
      }

      connection.send(value.buffer);
    }
  } catch (error) {
    connection.requestClose();
    throw error;
  }
}
```

Lastly, createNdjsonResponse creates the HTTP response that streams transcript events back to the browser. It wraps the readable stream in a Response object and sets the content type to `application/x-ndjson`:

```
function createNdjsonResponse(readable: ReadableStream<Uint8Array>): Response {
  return new Response(readable, {
    headers: {
      'Content-Type': 'application/x-ndjson',
      'Cache-Control': 'no-cache',
    },
  });
}
```

It's worthwhile discussing the browser-side event contract a bit. The browser sends audio as a streaming POST body. The route simultaneously pipes it to Deepgram and streams transcript events back as NDJSON. The browser reads the response line by line, parsing each event into a RawVoiceSignal:

```
interface RawVoiceSignal {
  type:         'speech_start' | 'speech_partial' | 'speech_final'
                | 'speech_end' | 'silence' | 'interrupt';
  timestamp:    number;
  transcript?:  string;
  duration_ms?: number;
}
```

The `speech_partial` events carry the transcript as it evolves, word by word. The `speech_final` event carries the complete, stable transcript after Deepgram's model has finalized its output. The sensing layer forwards both to the voice primitive detector: partial transcripts allow intent to be detected early, while final transcripts confirm it. A user who says "um, how do I—" has already revealed hesitation and the beginning of a question before they finish the sentence.

The key insight about Deepgram is that streaming transcription means the sensing layer gets context before the user finishes speaking. By the time the final transcript arrives, interpretation has already begun. This is what allows the system to feel responsive rather than reactive. The agent isn't waiting for the user to finish and then processing; it is processing in parallel with the first utterance.

ElevenLabs: Voice as an Act Channel

In our use case, if Deepgram is the agent's ears, ElevenLabs is its voice. It should be noted that each can do basically what the other does, but we want to utilise Deepgram's super-efficient STT, and Eleven Labs' hi-fidelity voice synthesis.

The relationship between text-to-speech and ambient UX is subtler than it might appear. Chat applications offer TTS as a convenience: you can choose to read or listen. In an ambient agent, voice is often the better output channel because the user's eyes are already occupied: they are looking at a form, reviewing a document, navigating a workflow. Ambient voice lets the agent deliver guidance without competing with the visual interface.

That said, ambient voice imposes requirements that batch TTS does not:

- Latency matters: a two-second pause before audio begins feels like a malfunction in the middle of a conversation.
- Interruptibility matters: if the user starts speaking, the agent must stop immediately and listen.
- And quality matters: it must be high enough that a calm, helpful voice doesn't feel jarring in the middle of focused work.

ElevenLabs addresses all three requirements. Its streaming API produces the first audio chunk before the full text has been synthesized, bringing perceived latency to roughly 500 milliseconds for typical short responses. The eleven_multilingual_v2 model is designed for natural conversational speech across a wide range of accents and speaking styles.

The Streaming Proxy and MediaSource Playback

Like Deepgram, ElevenLabs is accessed through a backend proxy that protects the API key. The proxy converts ElevenLabs's Node.js Readable stream to a Web ReadableStream and returns it as audio/mpeg:

```
// POST /api/elevenlabs
```

```
export async function POST(req: NextRequest): Promise<Response> {
  const { text, voice } = await req.json();
  const voiceId = voice ?? process.env.ELEVENLABS_DEFAULT_VOICE ?? 'Rachel';
  const client = new ElevenLabsClient({ apiKey: process.env.ELEVENLABS_API_KEY });

  const nodeStream = await client.textToSpeech.convertAsStream(voiceId, {
    text,
    model_id:      'eleven_multilingual_v2',
    output_format: 'mp3_44100_128',
  });

  const webStream = new ReadableStream<Uint8Array>({
    start(controller) {
      nodeStream.on('data', (chunk: Buffer) => controller.enqueue(new Uint8Array(chunk)));
      nodeStream.on('end',  ()             => controller.close());
      nodeStream.on('error', (err)         => controller.error(err));
    },
  });

  return new Response(webStream, {
    headers: { 'Content-Type': 'audio/mpeg' },
  });
}
```

In the browser, audio is played back using the MediaSource API, which allows chunks to be appended to a playback buffer as they arrive rather than waiting for the full file. Playback begins as soon as the first chunk is buffered, so the user starts hearing the response within half a second of the agent deciding to speak.

The critical property of the voice output handle is `stop()`, which cancels both the fetch request and the audio element simultaneously:

```
function stop(): void {
  controller?.abort();   // Cancel the in-flight request
  audioElement?.pause(); // Halt playback immediately
  audioElement.src = ''; // Release the MediaSource buffer
  playing = false;
}
```

A `stop()` call takes effect in the same event loop tick. There is no waiting for a chunk boundary, no queued audio draining. The voice simply stops. This is not a nice-to-have: it is a fundamental requirement for ambient agents. If the user begins speaking while the agent is mid-sentence and the agent continues for two more seconds before it registers the interruption, the interaction feels combative. Instant interruptibility is what makes ambient voice feel collaborative rather than intrusive.

The key insight about ElevenLabs is that ambient agents speak when appropriate, not when asked, and they always yield the floor. Voice is not a feature the user activates; it is a channel the agent uses when it judges that spoken guidance is the right medium for the moment. But the moment the user speaks, the agent needs to stop.

Voice Orchestration: Turn-Taking and Interruption

Deepgram and ElevenLabs are pipes: one delivers transcripts, the other delivers audio. Neither knows about the other. Something must sit between them and manage the conversation: whose

turn is it? When should the agent speak? When should it stop and listen?

This is voice orchestration, and it depends on a third voice component: Voice Activity Detection (VAD).

Voice Activity Detection

Push-to-talk is the simplest approach to voice input, where the user holds a button to speak and releases to stop. It works, but it doesn't feel ambient. An ambient agent should know when the user has finished speaking without requiring any explicit signal. VAD provides this.

VAD runs in the browser using an AudioWorklet that processes microphone audio inside the WebAudio pipeline. On each audio frame, the worklet computes the root mean square (RMS) energy, which serves as a straightforward proxy for speech activity. When energy exceeds a threshold, the user is likely speaking. When energy falls and stays below the threshold for 600 milliseconds, the utterance is likely complete:

```
const VAD_ENERGY_THRESHOLD  = 0.01;  // RMS level above which speech is active
const SPEECH_END_SILENCE_MS = 600;   // Silence duration to confirm utterance end
const PAUSE_WITHIN_SPEECH_MS = 250;  // Silence duration for mid-utterance pause
```

This timing produces three events: `speech_start` when continuous energy begins, `speech_end` when silence has persisted long enough to confirm the utterance is finished, and `silence` for mid-utterance pauses, such as the natural hesitations that appear in conversational speech.

The constraint that `PAUSE_WITHIN_SPEECH_MS` must be less than `SPEECH_END_SILENCE_MS` is enforced by the constants themselves: if they were equal, both timers would fire simultaneously, double-counting pause signals for every utterance that ends naturally.

The Turn-Taking State Machine

The voice orchestration layer combines VAD output with knowledge of whether the agent is currently speaking to manage a simple state machine:

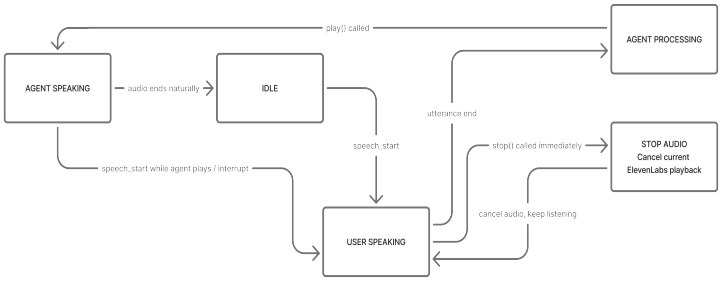


Figure 2.3: Turn-Taking State Machine

The interruption path is the most important. When the VAD detects the onset of user speech while the agent is in the `AGENT_SPEAKING` state, that onset is classified as an interruption before it is handled as an ordinary `speech_start`. The sensing layer emits an interrupt signal, calls `stop()` on the active voice output handle, and cuts off the agent's speech. It can then emit `speech_start` for the user's new utterance and process that utterance as the highest-priority input, triggering an

immediate graph invocation with `interaction.user_speaking = true`.

VAD is what makes voice feel ambient rather than walkie-talkie. Without it, users must explicitly signal the end of each utterance. With it, the conversation flows naturally: the user finishes speaking, a short silence threshold passes, and the agent responds. The latency is felt as a brief thinking pause, and that is exactly what it is.

What the Technology Stack Gives You

Taken individually, each of these technologies solves a specific problem. Together, they provide capabilities that would require substantial custom engineering to replicate:

Stateful session memory across episodes

LangGraph's checkpointer means the agent doesn't start from scratch each time a new episode arrives. It remembers prior interventions, tracks suppression streaks, and accumulates the full behavioral context of the session. Building this without a checkpointer would require implementing your own session state store with exactly-once delivery semantics — non-trivial engineering that LangGraph handles as infrastructure.

Frontend tool execution from backend logic

CopilotKit's useCopilotAction mechanism lets the agent's decisions manifest as DOM manipulations, route changes, and custom events, without the agent having any direct access to the browser. Without CopilotKit, you would need to design a custom tool invocation protocol, handle registration lifecycle and scope, and build the streaming delivery channel yourself.

Sub-second voice I/O

Deepgram's streaming transcription and ElevenLabs's streaming synthesis bring end-to-end voice latency into the range where conversation feels natural. Again, you could choose to use one or the other for similarity functionality. Achieving this with batch APIs would require significant buffering, latency compensation, and audio management logic that the streaming approach simply avoids.

Learnable application context

Neo4j's graph structure lets you represent the relationships between screens, tasks, steps, difficulty points, and historical outcomes in a form that is queryable at inference time and writable by an optimizer at learn time. A flat configuration file cannot represent these relationships; a relational database can, but the queries become complex as the schema evolves. A graph database makes these relationships first-class.

Backend-neutral architecture

None of these technologies require a specific web framework. LangGraph runs anywhere Node.js runs. CopilotKit's frontend is React, but the AG-UI protocol and the backend runtime are framework-neutral. The Deepgram and ElevenLabs proxy routes are plain HTTP handlers. The sensing and interpretation layers can be deployed independently of the frontend, tested in isolation, and migrated to a different runtime without touching the browser code.

What you don't get for free: the sensing logic that converts raw browser events into structured episodes, the interpretation pipeline that synthesizes episodes into a User State Model, and the

decision logic that evaluates intervention triggers against the interpreted state. Those are the layers that Chapters 3, 4, and 5 cover in depth. The technology stack gives you the infrastructure; the behavior is what you build on top of it.

In the next chapter, we turn to the sensing layer, the part of the system that transforms raw browser events into the structured episodes that all of this infrastructure depends on.

Summary

In this chapter, we introduced the core technologies that make ambient agents practical to build. We framed the system around the five-phase loop of Sense, Interpret, Decide, Act, and Learn, then mapped that loop to a working technology stack: Neo4j for memory and application context, LangGraph for stateful decision workflows, CopilotKit and AG-UI for browser-based human-agent collaboration, Deepgram for streaming voice input, and ElevenLabs for streaming voice output. We also examined how voice orchestration, turn-taking, and interruption allow the agent to participate naturally without taking control of the user's experience.

This matters because ambient agents are not simple request-response systems. They need to preserve session state, interpret behavior in application context, decide conservatively when to intervene, act through the interface or voice, and learn from the outcome. Together, these technologies provide a great infrastructure for ambient assistance, while the sensing, interpretation, and decision logic remain the system-specific behavior built on top.

In the next chapter, we will turn to the sensing layer. We will examine how raw browser activity is transformed into structured behavioral episodes, giving the rest of the ambient agent system something meaningful to interpret, reason over, and act upon.

3

Observation: Sensing User Behavior

In *Chapter 1*, we introduced the three broad capabilities an ambient agent needs: observation, memory, and restraint. Those are architectural concerns that recur throughout the system. In *Chapter 2*, we expanded this understanding, learning that our agent loop will be formed from a five-phase loop: sense, interpret, decide, act, and learn. The five-phase loop is the runtime process by which those concerns become actionable.

In this chapter, we will focus on the first phase: sense. The sensing layer observes user behavior and converts low-level interaction signals info structural behavioral evidence. We will look at a constrained set of signal types and show how they can be organized into behavioral episodes. These episodes are important because the meaning of signals crystalizes through context. A single mouse movement or scroll event reveals almost nothing about user intent. But a sequence of movements toward disabled UI elements, or repeated scrolling with reversals and pauses, can indicate confusion, uncertainty or struggle.

> **NOTE**
>
> Let's define what we mean by intent in this chapter. Intent does not mean a definitive claim about what the user wants. At the sensing layer, intent means a provisional behavioral hypothesis inferred from observable activity. The system cannot know the user's private goal from a mouse movement, scroll reversal, route change, or invalid form input. It can only produce evidence that suggests possible purposes. For that reason, we will use the term "candidate intent" for the sensing model's output. A candidate intent is a compact, tentative label, such as locate, compare, recover, inspect, or commit, attached to an episode with a confidence value. Later phases may confirm, revise, or ignore that hypothesis once more context is available.

To make the sensing architecture concrete, we will build a minimal sensing system that observes user interaction and aggregates those signals into distinct segments called behavioral episodes. Later chapters will build on this foundation to implement the remaining phases of the loop: interpret, decide, act, and learn. By the end of this chapter, you'll understand how to structure behavioral observations at the right level of abstraction, detailed enough to preserve meaning, but compact enough to enable efficient reasoning.

Here's what we'll will cover:

- Understanding Our Sensing System
- Implementing the Sensing Model
- Raw Signals
- Classification Primitives

- Patterns
- Candidate Intent
- Interaction Episodes

Understanding Our Sensing System

To start creating our ambient agent loop, we need a mechanism to sense user behavior and provide proactive assistance. We need to capture and process behavioral signals that may never become events in the conventional sense: hesitations, clicking cancel, clicking away, backtracking.

For this, we need more than user interface signals; we need to have them in context of the application UI so we can make sense of them. That context is the application the user is interacting with, the UX: how the elements are mapped on the screen and what those elements map to regarding APIs.

To create our sensing system, it's clear that low-level events such as mouse movements will be needed, but they are too raw and too frequent. Think of radio frequency identification (RFID) sensors spread throughout a hallway. Each sensor picks up a person's RFID tag at a high frequency once the tag is in range and until it is out of range. That is not the important information, rather it is the transition from sensor to sensor that reveals someone's movement through the corridor at different points in time. Low-level events are like sensor readings. To be meaningful, they must be rolled up into larger, more meaningful sub-events based on application UI context. These sub-events will coalesce onto an aggregate level that will enable the reconstruction of plausible user purpose.

The topology of the page where elements are mapped to the application function is the application UI context that will enable us to reconstitute user intent from low-level events. We don't want a series of mouse move coordinates, we want to know which UX elements are being moved toward or away from, or if the mouse is simply performing some kind of fidgeting, like moving in aimless circles.

When the mouse is moving, there will likely be a set of candidate targets. Imagine these targets having relative probabilities. In fact, it may be that the user will pass right over a target in favor of a different target further away that was initially less likely. At the same time, all elements might not be enabled at the same time, which is good because it reduces the solution space by eliminating irrelevant options, but the user may not realize this. A user trying to interact with elements that cannot yet be acted upon can provide insight into what operations need to be undertaken to enable those elements.

Implementing the Sensing Model

That's enough theorizing, let's jump into how we go about sensing user behavior so that our ambient agent can be proactive.

We could take the naive approach and register event listeners for every event we care about, then ship every event to the server as it occurs. That would be easy to implement, but it would flood the server with noise.

Throttling helps, but throttling by itself is not enough. It reduces volume but does not produce meaning. We need to compress low-level events into something that reflects what the user is

actually doing.

Figure 3.1 shows this compression pipeline from low-level browser activity to tenative candidate intent. The important thing to notice is that each stage reduces noise while preserving more behavioral meaning than the stage before it:



Figure 3.1: Four stages of the sensing model

Each level compresses information from the level that flows into it:

- Raw signals: low-level UI events such as mouse events and key presses
- Classification primitives: higher-level descriptors of the underlying action being performed in the UI, such as scanning up or down a document
- Patterns: groups of raw signals that combine to suggest a concerted semantic meaning
- Candidate intent: higher-level descriptors of plausible intent, given that intent at this point is hypothetical, and in this case, has not benefitted from the application UI context accessible in the interpretation phase

This pipeline does not make the browser "intelligent" by itself. It gives the rest of the ambient agent system a disciplined input format. Instead of reasoning over thousands of raw UI events, later phases receive compact behavioral evidence that has already been organized into primitives, patterns, and candidate intent.

We will discuss each level in the following sections.

Raw Signals

With just a small set of browser events to capture raw signals, we can get pretty far in capturing candidate intent. In this way, these raw signals serve as a vocabulary we can use to express intent in wide-ranging scenarios. While we could cover additional browser events to encompass things like viewport awareness, errors, and media events, we'll focus on just six signals here:

- Route changes
- Scroll events
- Mouse movement
- Focus events
- Input events
- Clicks

We can represent our selected signal set with the following union:

```
export type RawSignal =
   | { type: 'route'; timestamp: number; route: string }
   | { type: 'scroll'; timestamp: number; scrollY: number; previousScrollY: number }
   | { type: 'mousemove'; timestamp: number; x: number; y: number }
   | { type: 'focus'; timestamp: number; formId?: string; fieldId?: string }
   | { type: 'input'; timestamp: number; formId?: string; fieldId?: string;
       valueState: 'empty' | 'partial' | 'filled'; invalid: boolean; }
   | { type: 'click'; timestamp: number; role?: string; elementType?: string
```

```
};
```

We are not capturing everything we get from the low-level browser events. For example, for input events, rather than capturing the actual field values, we are only recording coarse state such as `empty`, `partial` or `filled`, plus whether the input is invalidated. The sensing layer needs to preserve behavioral meaning, not raw payloads.

The same principle is even more important for `mousemove` events. A stream of `mousemove` events is too large and too noisy to be useful in raw form. So, although `mousemove` is one of our signal types, we continuously drop events that don't offer much interpretative value in the way we described with RFID sensors. Put another way, we want to aggressively compact or summarize `mousemove` state.

Classification Primitives

Signals are still quite close to the browser's JavaScript events. Classification primitives get us closer to intent by providing small behavioral fragments that recur across many interactions.

We will define them like this:

```
export type Primitive =
    | 'scan' | 'dwell' | 'reversal'| 'retry'| 'commit' | 'approach';
```

These are much more meaningful than raw browser events:

- `scan` suggests forward movement through content
- `dwell` suggests a pause with attention
- `reversal` suggests a change of direction
- `retry` suggests repeated effort
- `commit` suggests a decisive action
- `approach` suggests movement toward a possible target

The important shift here is that classification primitives are not browser events. They are already an interpretive compression of browser events. They are still local and limited, but they begin to describe behavior rather than mechanics. The notable thing here is that like the raw signals, classification primitives are application-agnostic, taking no account of the application domain.

Let's trace what happens when a user fills out a form. First, we have the raw signals, what the browser sees:

```
{ type: 'input', timestamp: 1500, fieldId: 'email', invalid: true }
{ type: 'input', timestamp: 2000, fieldId: 'email', invalid: true }
{ type: 'input', timestamp: 2500, fieldId: 'email', invalid: false }
```

Next, we have the classification primitives, what the behavior means:

```
primitives: ['dwell', 'retry', 'retry', 'commit']
```

The classification primitives layer does the hard work of temporal reasoning, pattern matching, and signal correlation.

Patterns

Classification primitives tell us what happened, but not what it means. That's where patterns come in. Classification primitives alone are still too granular to drive an ambient agent. We need to be able to speak about larger behavioral structures. Patterns describe a shape of behavior, while candidate intent is the system's provisional guess at the user's possible goal.

Let's look at how `Pattern` is defined in the code:

```
export type Pattern =
 | 'hesitation' | 'backtracking' | 'exploration' | 'field-struggle'
 | 'read-reread' | 'fidgeting';
```

The `Pattern` type definition is intentionally small. Each value names an episode-level behavioral construct, not a browser event and not a user goal. For example, `field-struggle` does not mean the system knows what the user wants. It means the sensing layer has seen enough form-related evidence, such as repeated invalid inputs or retries, to summarize the episode as a struggle around a field.

Let's continue our form example, where the classification primitives shed light on plausible behavior. Now with patterns, we can get into the possible meaning:

```
patterns: ['field-struggle']
```

Classification primitives are local behavioral fragments, while patterns are higher-level behavioral shapes.

Patterns exist to answer this question: *What larger behavioral shape emerges when several primitives are considered together inside one episode?*

Consider these examples:

- Several `reversal` primitives in a navigation episode may produce `backtracking`
- `reversal+` dwell in a reading episode may produce `read-reread`
- Repeated retry plus invalid inputs in a form episode may produce field-struggle
- Contradictory motion segments may produce `fidgeting` or `hesitation`

Patterns are therefore episode-level interpretations of primitive combinations. They are necessary because they stabilize the evidence before intent inference. Without patterns, intent inference would have to operate directly on noisy, low-level fragments every time.

Candidate Intent

With patterns, we see what happened, but we don't know why. To understand the why, we introduce another type, candidate intent. We define it as follows:

```
export type CandidateIntent =
   | 'inspect' | 'locate' | 'compare' | 'recover' | 'commit';
```

Candidate intents are tentative semantic hypotheses about user goals. We need them because patterns still do not tell us what the user is trying to do. A pattern like backtracking is descriptive, not goal-oriented. The pattern says this behavior looks like revisiting prior states, but it does not say whether the user is:

- Trying to locate something

- Comparing options
- Recovering from a mistake
- Exploring

Candidate intent exists to bridge from behavioral description to possible user purpose.

The "candidate" part is important. The sensing layer is not claiming certainty. It is saying, "given what we have seen so far, the most plausible explanation is X." That gives later phases of the ambient agent loop something to reason over, without assuming the sensing layer knows the truth.

To continue the previous example, where the pattern was `field-struggle`, the corresponding candidate intent could be `recover`. But let's also look at another example where both cases exhibit similar patterns:

```
// Case A
{ episodeType: 'form', patterns: ['field-struggle'] }
{ episodeType: 'motion', patterns: ['hesitation', 'fidgeting'] }
{ episodeType: 'navigation', patterns: ['backtracking'] }

// Case B
{ episodeType: 'navigation', patterns: ['backtracking'] }
{ episodeType: 'reading', patterns: ['hesitation'] }
{ episodeType: 'motion', patterns: ['fidgeting'] }
```

In the first case, the user may be attempting to `recover`, while in the second case, the user may be attempting to `inspect`.

This gives us the full vocabulary of the sensing model: raw signals, primitives, patterns, and candidate intent. But vocabulary is not enough. The sensing layer also needs a unit composition that decides which signals belong together. That unit is the interaction episode.

Interaction Episodes

Now we can introduce the real structural unit of the sensing layer: interaction episodes.

Signals, classification primitives, patterns, and candidate intents give us the vocabulary of sensing. Interaction episodes give us the grammar. An interaction episode is a bounded sequence of related UI signals that, taken together, represent one coherent slice of user activity. Instead of storing raw events directly, we collect them into episodes so that the sensing layer can describe behavior at the right level of abstraction.

This matters because behavior does not arrive prepackaged. The browser gives us a stream of events. The sensing layer has to decide which events belong together strongly enough to count as one thing. Episodes are the answer to that problem.

To keep the example focused, we will limit ourselves to four kinds of episodes:

- Motion episodes: reveal how a user moves through the visible interface
- Navigation episodes: reveal how a user moves through the application
- Reading episodes: reveal how a user consumes page content
- Form episodes: reveal how a user attempts to complete structured input

These are intentionally simple, but together they provide substantial explanatory power. A burst

of mouse movement can indicate approach, hesitation, or fidgeting. A route sequence can indicate exploration, comparison, or backtracking. Scroll bursts and pauses can indicate scanning or rereading. Repeated invalid form inputs can indicate struggle and recovery.

An episode is therefore the unit in which raw interface activity becomes a behavioral interaction. You can think of episodes as the behavioral equivalent of phrases or sentences in a narrative. A single signal is too small to tell us much. A bounded sequence of signals, interpreted together, can begin to tell us what kind of interaction is unfolding.

Before we look at the individual episode types, it helps to define the shared structure they all follow. Motion, reading, navigation, and form episodes capture different kinds of behavior, but they all need to do some basic work: accept raw signals while the episode is open, derive patterns when the episode closes, infer candidate intent, and serialize the result into a compact record.

We will represent that shared structure with an abstract base class called `Episode<TSummary>`. The generic type parameter, `TSummary`, allows each episode type to define its own summary shape while still following the same lifecycle.

The Base Episode Class

The `Episode<TSummary>` abstract base class establishes a contract that all episode types must fulfill:

```
abstract class Episode<TSummary> {
  abstract process(signal: RawSignal): void;
  protected abstract finalizePatterns(): void;
  abstract inferIntent(): { type: CandidateIntent; confidence: number } | undefined;
  abstract summarize(): TSummary;
}
```

Each method corresponds to a distinct phase in the episode lifecycle:

- `process()` handles accumulation while the episode is open. As raw signals arrive, this method extracts primitives and records episode-specific features.
- `finalizePatterns()` handles analysis as the episode closes. It examines the accumulated primitives and summary features to identify higher-level behavioral patterns.
- `inferIntent()` handles inference after patterns have been finalized. It proposes a candidate intent only when the completed episode provides enough evidence.
- `summarize()` handles serialization. It produces the compact episode-specific summary that will be included in the emitted episode record.

Figure 3.2 shows the lifecycle every episode follows. While the episode is open, it accumulates signals. When the episode closes, it finalizes patterns, infers candidate intent, and serializes the compact record:

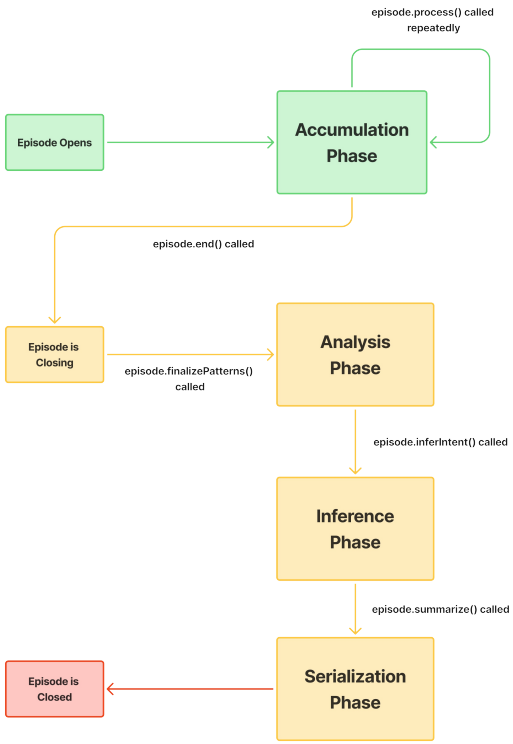


Figure 3.2: Episode Lifecycle

The ordering is important. Pattern detection happens only after the episode has collected enough evidence, and candidate intent is inferred only after those patterns have been finalized. That sequence prevents the sensing layer from making premature claims based on partial behavior.

Accumulation

As raw signals arrive, they are passed into the episode's `process()` method. This method performs the first layer of behavioral compression. It extracts classification primitives such as `scan`, `reversal`, `retry`, `dwell`, `commit`, or `approach`, and records any episode-specific features needed later. At this stage, the sensing layer is no longer merely observing browser mechanics; it is beginning to describe behavior in compact, reusable fragments.

Analysis

When the episode ends, `finalizePatterns()` is invoked on the collected primitives and summary features are consolidated into patterns. This step is deferred until termination because many patterns depend on the overall shape of the episode rather than on any single event. A route revisit may contribute to backtracking, a dwell plus reversal may contribute to rereading, and contradictory motion segments may contribute to fidgeting or hesitation.

Inference

Only after the episode closes does the sensing layer attempt to infer what the user might have been trying to do. This is handled in `inferIntent()`. Candidate intent is therefore based on the completed episode rather than on a partial stream of events, which reduces premature

interpretation and makes the inference more stable.

The important detail is that intent is inferred only after the episode boundary, not during the streaming of corresponding events. An episode therefore acts as a behavioral buffer between raw signals and higher-level understanding.

Serialization

Each concrete episode class, such as `MotionEpisode`, `ReadingEpisode`, `NavigationEpisode`, or `FormEpisode`, inherits shared storage for primitives and patterns from the base class. The base class keeps primitive counts in a map and pattern order in an array.

```
protected primitives: Map<Primitive, number> = new Map();
protected patterns: Pattern[] = [];

protected addPrimitive(value: Primitive): void {
  this.primitives.set(value, (this.primitives.get(value) ?? 0) + 1);
}

protected addPattern(value: Pattern): void {
  this.patterns.push(value);
}
```

The `primitives` field is a map because the sensing layer needs to preserve frequency. If retry occurs three times in an episode, that count may matter to downstream interpretation. The `patterns` field is an array because patterns should be preserved as discrete findings in the order they were finalized.

When `summarize()` is called, the emitted episode record includes the primitive counts, finalized patterns, candidate intent, and episode-specific summary. That record becomes the handoff point between client-side sensing and server=side interpretation.

Finalization

Episodes are continuous behavior sessions. They start automatically but need to end explicitly when:

- User completes a task
- User switches context
- Enough time passes without activity
- User navigates away

The `end()` method is especially important because it shows the boundary between sensing and later phases. An episode emits a compact behavioral record containing the following vital information:

- Timing
- Starting and ending context
- Primitives
- Patterns
- Summary metrics
- Candidate intent

The `end()` method brings these pieces together. It marks the episode boundary, captures the

ending context, finalizes patterns, infers candidate intent, and returns the emitted record. The ordering inside this method is important: candidate intent is inferred only after patterns have been finalized.

```
  end(reason: TerminationReason): EpisodeEmission<TSummary> {
    this.endTime = Date.now();
    this.endContext = getCurrentContext();

    // Finalize derived patterns before intent inference and serialization.
    this.finalizePatterns();
    const candidateIntent = this.inferIntent();

    return {
      episodeId: this.id,
      episodeType: this.type,
      sessionId: this.sessionId,
      pageViewId: this.pageViewId,
      sequenceNumber: this.sequenceNumber,
      startTime: this.startTime,
      endTime: this.endTime,
      duration: this.endTime - this.startTime,
      terminationReason: reason,
      uiContext: {
        start: this.startContext,
        end: this.endContext,
      },
      primitives: Object.fromEntries(this.primitives),
      patterns: Array.from(this.patterns),
      candidateIntent,
      summary: this.summarize(),
      clientVersion: '1.0',
    };
  }
```

This method is the boundary between sensing and interpretation. While the episode is open, the sensing layer accumulates evidence. Once `end()` is called, the episode becomes a compact behavioral record that can be sent downstream. Later phases do not need the full raw event stream; they receive the structured evidence needed to reason about what may be happening.

Episode Types

Now that we've looked at the abstract base class for episodes, let's examine some concrete specializations. Each episode type captures a distinct category of user behavior, defining what primitives to track, how to detect episode boundaries, and which patterns emerge from the accumulated data.

To keep the implementation focused, we will look at four core episode types. four core episode types:

- Motion Episodes: Track cursor movement and reveal spatial navigation patterns
- Reading Episodes: Capture text engagement through scroll position and viewport stability
- Navigation Episodes: Monitor route changes and page transitions
- Form Episodes: Record input interactions and form completion behavior

Each episode type follows the same lifecycle, but each one defines its own accumulation logic, summary fields, pattern rules, and candidate intent heuristics.

Motion Episodes

Motion episodes demonstrate why an ambient sensing layer is necessary. `mousemove` events fire at the browser's refresh rate, many times per second. Capturing raw coordinates would overwhelm the sensing pipeline while failing to produce meaningful behavioral insight. The motion episode solves this through intelligent compaction, transforming a high-frequency event stream into a compact behavioral representation.

The episode performs a single critical transformation. It groups cursor movement into directional segments. Rather than storing individual coordinates, it tracks runs of movement in consistent directions. A segment continues while the cursor moves in roughly the same orientation. When direction changes materially, the segment closes and a new one begins.

Each motion segment captures four characteristics that preserve the behavioral shape of the movement without storing every cursor coordinate:

```
interface MotionSegment {
  orientation: string;        // Compass direction: N, NE, E, SE, S, SW, W, NW
  distance: number;           // Total pixels traveled
  duration: number;           // Milliseconds elapsed
  acceleration: 'accelerating' | 'decelerating' | 'steady';
}
```

These four fields tell us where the cursor was generally moving, how far it traveled, how long the movement lasted, and whether the motion was gaining or losing speed. That is enough to distinguish smooth directed movement from fragmented or uncertain movement, while avoiding a large stream of raw coordinates.

The episode summary then exposes the collected segments along with two derived metrics:

```
interface MotionSummary {
  segments: MotionSegment[];
  contradictorySegments: number;  // Count of directional reversals
  fidgetScore: number;            // 0-1 measure of movement fragmentation
}
```

The `contradictorySegments` value records how often the user reversed direction. The `fidgetScore` compresses movement fragmentation into a value between 0 and 1, where higher values suggest more short, contradictory motion.

Those summary fields do not appear automatically. The episode needs a small amount of internal state to build them while mouse movement is still arriving. The class skeleton below is focused on that state rather than the full processing logic:

```
class MotionEpisode extends Episode<MotionSummary> {
   private segments: MotionSegment[] = [];
   private contradictorySegments = 0;
   private lastPoint: { x: number; y: number; t: number } | null = null;
   private currentSegment:
       | {
            orientation: string;
            distance: number;
            duration: number;
            firstSpeed: number;
            lastSpeed: number;
         }
```

```
        | null = null;
    constructor(sessionId: string, pageViewId: string, sequenceNumber: number) {
         super('motion', sessionId, pageViewId, sequenceNumber);
    }
```

The `segments` array stores the compacted directional runs that will later appear in the `MotionSummary`. The `contradictorySegments` field counts reversals as they are detected. The `lastPoint` field gives each new `mousemove` signal something to compare against, while `currentSegment` holds the directional run currently being accumulated. Together, these fields give the episode enough memory to compress thousands of `mousemove` events into a small number of behavioral segments.

The episode's `process()` method implements the segmentation logic. When a `mousemove` signal arrives, it calculates distance and direction from the previous point. We take jitter into account, so that small, inadvertent variations that do not carry behavioral meaning do not influence processing. Movements below a jitter threshold (4 pixels) are discarded as noise.

Direction is quantized into compass buckets rather than stored as exact angles. This provides behavioral stability: four movements at 43°, 41°, 44°, and 45° all map to `NE` and are recognized as a continuous segment. Exact angles would fragment this into four separate segments, obscuring the underlying pattern.

When a new movement's orientation matches the current segment, the segment is extended so that distance and duration accumulate, and the latest speed is recorded. When orientation changes, the current segment is finalized and pushed to the segments array. If the new direction contradicts the previous one (east then west, north then south), a reversal counter increments and a reversal primitive is added. These contradictions may indicate hesitation or uncertainty.

The acceleration classification compares the segment's final speed to its initial speed. If the final speed exceeds the initial by 15% or more, the segment is "accelerating." If it falls below 85%, it's "decelerating." Otherwise, it's "steady." This captures whether the user was building momentum, approaching a target, or maintaining constant speed.

When the episode ends (such as after 250ms of inactivity), `finalizePatterns()` analyzes the complete segment sequence. Any open segment is finalized first. Then pattern recognition runs: one or more contradictory segments produces a hesitation pattern. Multiple contradictions combined with several short segments (under 80 pixels and 250ms) produce a fidgeting pattern.

The following helper method shows how the episode converts contradictory and short movement segments into a single `fidgetScore`:

```
private calculateFidgetScore(): number {
  if (this.segments.length === 0) return 0;

  const shortSegments = this.segments.filter(
    (segment) => segment.distance <80&&segment.duration< 250
  ).length;

  return Math.min(
    1,
    (this.contradictorySegments + shortSegments) / (this.segments.length * 1.5)
  );
}
```

A high fidget score indicates many reversals and brief movements—the cursor wandering without

decisive action. A low score indicates smooth, directed motion. The score serves as a plausible conjecture. A high fidget score near a submit button followed by a form error might warrant interpretation as uncertainty. A low score terminating in a click might suggest confidence.

This exemplifies what a sensing layer should do. It doesn't preserve raw data or make definitive claims about user intent. It compresses high-frequency signals into a form where later phases can recognize meaningful patterns. The motion episode receives thousands of coordinate pairs but emits dozens of segments, reducing data volume by orders of magnitude while preserving behavioral nuance.

Reading Episodes

Reading episodes track how users consume page content through scroll behavior. Like motion episodes, they follow the same architectural pattern: accumulate signals, extract primitives, stabilize patterns, and infer candidate intent.

The act of page reading tends to produce bursts of scrolling followed by pauses. Those pauses matter because they often indicate attention. The reading episode does not try to understand content. It works entirely from interaction structure: distance, direction changes, and dwell intervals.

The episode summary is intentionally small. It keeps only the metrics needed to describe how the user moved through page content:

```
interface ReadingSummary {
    totalScrollDistance: number;
    reversals: number;
    dwellMoments: number;
}
```

These fields capture the total amount of scrolling, the number of direction changes, and the number of meaningful pauses. The summary does not attempt to identify what content the user read. It only records the structure of the reading behavior.

The next code block shows how those summary metrics are accumulated from raw scroll signals. The `process()` method converts scroll behavior into three classification primitives: `scan`, `reversal`, and `dwell`:

```
class ReadingEpisode extends Episode {
    private totalScrollDistance = 0;
    private reversals = 0;
    private dwellMoments = 0;
    private lastDirection: 'up' | 'down' | null = null;
    private lastScrollTimestamp: number | null = null;
    constructor(sessionId: string, pageViewId: string, sequenceNumber: number) {
        super('reading', sessionId, pageViewId, sequenceNumber);
    }
    process(signal: RawSignal): void {
        if (signal.type !== 'scroll')
            return;
        const delta = signal.scrollY - signal.previousScrollY;
        const direction: 'up' | 'down' = delta >= 0 ? 'down' : 'up';
        this.totalScrollDistance += Math.abs(delta);
        this.addPrimitive('scan');
        if (this.lastDirection && this.lastDirection !== direction) {
            this.reversals += 1; this.addPrimitive('reversal');
        }
```

```
if(this.lastScrollTimestamp&&signal.timestamp-this.lastScrollTimestamp> READING_DWELL_THRESHOLD_MS ) {
            this.dwellMoments += 1; this.addPrimitive('dwell');
        }
        this.lastDirection = direction;
        this.lastScrollTimestamp = signal.timestamp;
}
```

A scroll movement adds scan; a direction change adds reversal; and a long enough pause between scroll events adds dwell.

The episode accumulates these primitives throughout the scroll sequence. When the episode closes, `finalizePatterns()` incorporates the collected primitives in higher-level behavioral patterns:

```
protected finalizePatterns(): void {
    if (this.reversals >= 1) {
        this.addPattern('read-reread');
    }
    if (this.dwellMoments >=1&&this.reversals>= 1) {
        this.addPattern('hesitation');
    }
}
```

A single `reversal` suggests the user scrolled back to review content they already passed. This corresponds to the `read-reread` pattern. When `reversals` combine with `dwell` moments, it suggests the user paused, reconsidered, and returned to previous content. This corresponds to the `hesitation` pattern.

Only after pattern finalization does `inferIntent()` propose what the user might have been attempting:

```
inferIntent(): { type: CandidateIntent; confidence: number } |
undefined {
    if (this.reversals >=2&&this.dwellMoments>= 1) {
        return { type: 'inspect' as const, confidence: 0.75 };
    }
    return undefined;
}
```

Multiple reversals with pauses suggest careful examination rather than casual browsing. The episode proposes inspect as the candidate intent, with moderate confidence. If the pattern doesn't meet the threshold, no intent is inferred because the sensing layer only makes claims it can support.

The reading episode therefore follows the same discipline as the motion episode: accumulate signals, stabilize them into patterns, summarize the episode, and only then produce a tentative inference.

Navigation Episodes

Navigation episodes capture how users move through the application's route structure. While reading episodes track movement within the viewport, navigation episodes track movement between pages. A single route transition tells us very little. But route sequences can reveal intent. Consider these two examples, a single transition:

```
/dashboard → /projects
```

Followed by a route transition:

```
/dashboard → /projects → /dashboard → /projects → /settings
```

The single transition might mean almost anything, whereas the sequence reveals oscillation: the user visited projects, returned to the dashboard, revisited projects, then moved to settings. This back-and-forth pattern suggests the user is searching for something or comparing options.

The navigation episode summary records the route path and the simplest signals needed to reason about that path:

```
interface NavigationSummary {
    routeSequence: string[];
    reversals: number;
    returnedToOrigin: boolean;
}
```

The `routeSequence` field preserves the ordered path. reversals count how often the user returns to a previously visited route. `returnedToOrigin` records whether the user eventually came back to the starting route, which can be useful when detecting backtracking or failed exploration.

The next code block shows how the episode builds that summary during accumulation:

```
class NavigationEpisode extends Episode {
    private routes: string[] = [];
    private reversals = 0;
    constructor( sessionId: string, pageViewId: string, sequenceNumber: number, initialRoute: string, ) {
        super('navigation', sessionId, pageViewId, sequenceNumber);
        this.routes.push(initialRoute);
    }
    process(signal: RawSignal): void {
        if (signal.type === 'route') {
            const seen = this.routes.includes(signal.route);
            this.routes.push(signal.route);
            if (seen) {
                this.reversals += 1;
                this.addPrimitive('reversal');
            }
    }
    if (signal.type === 'scroll') { this.addPrimitive('scan'); }
    if (signal.type === 'click' && signal.role === 'link') {
        this.addPrimitive('commit');
    }
}
```

The `process()` method monitors route changes, scroll events, and link clicks. Route revisits produce `reversal`; scroll activity produces `scan`; and link clicks produce `commit`.

Each route change extends the sequence. The episode doesn't evaluate meaning during accumulation, but simply records the path and notes when `reversals` occur.

When the episode terminates, `finalizePatterns()` examines the complete route sequence and derives higher-level patterns:

```
protected finalizePatterns(): void {
    if (this.reversals >= 2) {
        this.addPattern('backtracking');
    }
    if (new Set(this.routes).size >= 4) {
```

```
        this.addPattern('exploration');
    }
}
```

Multiple `reversals` indicate `backtracking`, where the user repeatedly returned to previously visited pages. Visiting many distinct routes indicates `exploration`, where the user moved through a wide area of the application. These patterns are mutually compatible; a user can both backtrack and explore in the same episode.

The `inferIntent()` method then interprets these patterns as candidate intents:

```
inferIntent(): {
    type: CandidateIntent; confidence: number
} | undefined {
    const uniqueRoutes = new Set(this.routes).size;
    if (this.reversals >=2&&uniqueRoutes<= 3) {
        return {
            type: 'locate' as const, confidence: 0.75
        };
    }
    if (this.reversals >=1&&uniqueRoutes>= 4) {
        return { type: 'compare' as const, confidence: 0.7 };
    }
    if (uniqueRoutes >= 5 && this.reversals === 0) {
        return { type: 'inspect' as const, confidence: 0.65 };
    }
    return undefined;
}
```

This example uses simple heuristics so the sensing model remains easy to follow. In a production system, these thresholds should become configurable adn eventually adaptive, using feedback from the interpretation and learning phases.

Form Episodes

Form episodes capture attempts to complete structured input tasks. Forms are one of the richest places for behavioral signals because they can expose user difficulty very clearly. The sensing layer does not need to know what a form field means semantically to detect that a user is encountering trouble. Focus patterns, partial entries, validation errors, and repeated attempts reveal difficulty through behavior alone.

The form episode records only the behavioral facts needed to describe the user's attempt to complete the form:

```
interface FormSummary {
    formId: string;
   totalInputs: number;
    invalidInputs: number;
    fieldsTouched: string[];
}
```

The `formId` identifies which form the episode describes. `totalInputs` records how much input activity actually occurred, while `invalidInputs` records how often that activity failed validation. `fieldsTouched` preserves the set of fields the user interacted with, without storing the actual values they entered.

The next code block shows how those summary fields are accumulated and how form-related

signals become primitives. The `process()` method handles three kinds of evidence: focus, input, and submit clicks. Focus adds `dwell`, because the user has paused on a field. Partial input adds `retry`, because the user is still working through the value. Filled input or a submit click adds `commit`, because the user has taken an action that moves the form toward completion.

```
class FormEpisode extends Episode<FormSummary> {
    private totalInputs = 0;
    private invalidInputs = 0;
    private touched = new Set();
    constructor( sessionId: string, pageViewId: string, sequenceNumber: number, private formId: string, ) {
        super('form', sessionId, pageViewId, sequenceNumber);
    }
    process(signal: RawSignal): void {
        if (signal.type === 'focus' && signal.fieldId) {
            this.touched.add(signal.fieldId);
            this.addPrimitive('dwell');
        }
        if (signal.type === 'input') {
            this.totalInputs += 1;
            if (signal.fieldId)
                this.touched.add(signal.fieldId);

            if (signal.invalid) {
                this.invalidInputs += 1;
            }

            if (signal.valueState === 'partial') {
                this.addPrimitive('retry');
            }

            if (signal.valueState === 'filled') {
                this.addPrimitive('commit');
            }
        }
        if (signal.type === 'click' && signal.elementType === 'submit') {
            this.addPrimitive('commit');
        }
    }
```

Notice what the episode captures and what it discards. Actual field values are never recorded, only their high-level state: `empty`, `partial`, or `filled`, plus whether validation failed. This preserves behavioral meaning while respecting privacy. The sensing layer doesn't need to know that the user typed `"john@examplecom"` to detect that an email field received an invalid value.

The result is a compact record of form behavior. We know that input happened, which fields were touched, and whether validation failed, but we do not preserve the sensitive contents of the form itself.

Once the episode closes, the form episode turns accumulated validation trouble into a pattern. In this simplified version, any invalid input is enough to mark the episode with "field-struggle".

```
protected finalizePatterns(): void {
      if (this.invalidInputs >= 1) {
          this.addPattern('field-struggle');
      }
}
```

The pattern deliberately stays behavioral. It doesn't claim to know why the user struggled, which

validation rule failed, or what they intended to type. It only records that the form episode contained enough failed input activity to count as field-level struggle.

After patterns have been finalized, `inferIntent()` can propose a candidate intent. Here, repeated invalid input suggests `recover`, while ordinary input activity without significant errors suggests `commit`.

```
inferIntent(): {
    type: CandidateIntent; confidence: number
} | undefined {
    if (this.invalidInputs >= 2) {
        return { type: 'recover' as const, confidence: 0.7 };
    }
    if (this.totalInputs > 0) {
        return { type: 'commit' as const, confidence: 0.6 };
    }
    return undefined;
}
```

Multiple invalid inputs suggest `recover`, where the user may be trying to correct errors. Input activity without significant errors suggests `commit`, where the user appears to be making ordinary progress toward form completion. The lower confidence on `commit` reflects that successful input is the default path through a form, while `recover` is a more distinctive pattern.

The form episode shows the full sensing-layer discipline in miniature. It captures raw signals, compresses them into primitives, stabilizes those primitives into patterns, and proposes candidate intent only when the evidence supports it. A later interpretation layer may combine this form episode with motion, navigation, or reading episodes to decide whether help is warranted. But at the sensing state, the layer's responsibility ends with producing an accurate, compact behavioral record.

Across these four episode types, the important point is not the exact heuristic chosen in each example. The important point is the shape of the sensing architecture: raw signals are compressed into primitives, primitives are stabilized into patterns, and patterns support candidate intent. Motion episodes compact cursor movement, reading episodes compact scroll behavior, navigation episodes compact route movement, and form episodes compact input attempts. Each one preserves behavioral meaning without preserving unnecessary raw detail, producing a compact record that later phases can interpret in richer context.

Summary

In this chapter, you learned how to build the sensing layer of an ambient agent. This layer transforms raw browser events into structured behavioral evidence that later phases can interpret. You saw how low-level events such as mouse movement, scrolling, route changes, focus, input, and clicks can be compressed into classification primitives such as `scan`, `dwell`, `reversal`, `retry`, `commit`, and `approach`. You then saw how these primitives combine into higher-level patterns like hesitation, field-struggle, backtracking, and read-reread, and how those patterns can support tentative candidate intents such as `locate`, `compare`, `recover`, `inspect`, or `commit`.

This information is important because ambient agents cannot offer proactive assistance unless they first have a disciplined way to observe behavior. Raw event streams are too noisy, too large, and too low-level to support reliable interpretation. The episode architecture gives the agent a

compact behavioral record that preserves meaning while discarding unnecessary detail. Through motion, reading, navigation, and form episodes, you saw how the sensing layer can remain lightweight on the client while producing structured evidence for downstream reasoning.

In the next chapter, you'll move into the interpret phase. There, you'll learn how to analyze sequences of behavioral episodes together, combine them with application UI context, and infer what the user may be trying to accomplish. This will allow the ambient agent to move from structured observation toward contextual understanding, which is the foundation for deciding whether, when, and how to intervene.

Building Ambient AI Agents

1. Welcome to Packt Early Access
    1. Building Ambient AI Agents, First Edition: Create autonomous always-on product experiences that move beyond chat
2. Chapter 1: Beyond Chat: The Emergence of Ambient Agents
    1. Traditional AI Agents: Lacking Initiative
        1. Failure 1: Cognitive Load
        2. Failure 2: Context Reset
        3. Failure 3: Missed Timing
        4. Failure 4: Unknown Unknowns
        5. Failure 5: Interaction Burden
        6. Why These Failure Modes Are Now Solvable
        7. The Architecture Will Outlast the Model
    2. Ambient Agents: Observation, Memory, Restraint
        1. Case Example: Deploy-Time Code Review
        2. 1.Observation
            1. Three Principles of Observation
            2. What the Agent Can Monitor
            3. An Observation Example
            4. Observation and Memory
        3. 2.Memory
        4. 3.Restraint
        5. The Dance of Three: Observation, Memory, Restraint
    3. Is Your Application a Good Candidate for an Ambient Agent?
        1. Do you need to understand the full spectrum of user behavior?
        2. Do you need this understanding in real time?
        3. Do you need to take action on what you learn?
    4. Summary
3. Chapter 2: Core Technologies: Building Blocks for Ambient Agents
    1. The Ambient Agent Five-Phase Loop
        1. Sense
        2. Interpret
        3. Decide
        4. Act

5. Learn
6. The Loop

2. Neo4j: Memory and Application Context
    1. The Application Context Graph
    2. Heuristic Configuration
    3. Outcome Storage
3. LangGraph: The Decision Spine
    1. Why a Graph?
    2. State as the Agent's Memory
    3. The Graph Topology
4. CopilotKit and AG-UI: The Human-Agent Collaboration Layer
    1. The AG-UI Protocol
    2. Frontend Tools with useCopilotAction
    3. The Provider Pattern
5. Deepgram: Voice as a Sensing Channel
    1. Challenge of voice processing
    2. The Backend Proxy Pattern
6. ElevenLabs: Voice as an Act Channel
    1. The Streaming Proxy and MediaSource Playback
7. Voice Orchestration: Turn-Taking and Interruption
    1. Voice Activity Detection
    2. The Turn-Taking State Machine
8. What the Technology Stack Gives You
    1. Stateful session memory across episodes
    2. Frontend tool execution from backend logic
    3. Sub-second voice I/O
    4. Learnable application context
    5. Backend-neutral architecture
9. Summary

4. Chapter 3: Observation: Sensing User Behavior
    1. Understanding Our Sensing System
    2. Implementing the Sensing Model
    3. Raw Signals
    4. Classification Primitives
    5. Patterns
    6. Candidate Intent

7. Interaction Episodes
    1. The Base Episode Class
        1. Accumulation
        2. Analysis
        3. Inference
        4. Serialization
        5. Finalization
    2. Episode Types
        1. Motion Episodes
        2. Reading Episodes
        3. Navigation Episodes
        4. Form Episodes
8. Summary