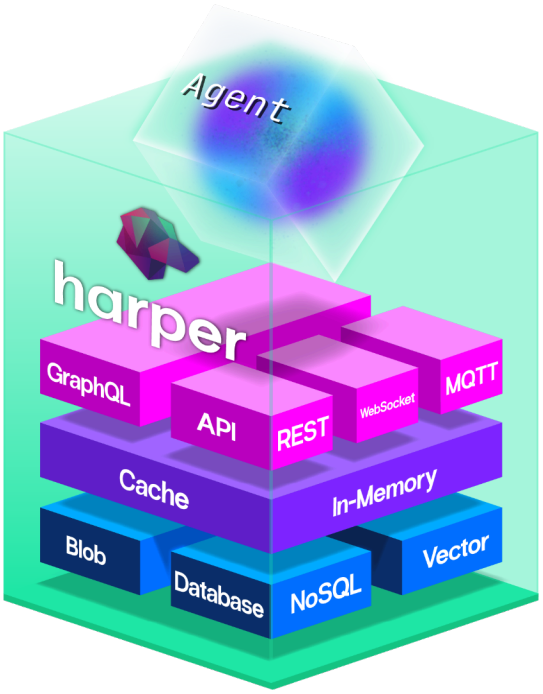
// A PRACTICAL GUIDE

Agents on Harper

Why the unified runtime fits agentic workloads, what outcomes it delivers, and how it compares to the alternatives.

Platform engineers · AI engineers · Architects · Technical buyers



INSIDE THIS GUIDE

23	5	10	9
PAGES	PARTS	OUTCOMES	WORKLOADS

// TABLE OF CONTENTS

What this guide covers

This guide walks through Harper as a runtime for production agentic workloads. It is organized around five questions a technical buyer or implementer typically asks when evaluating agent infrastructure.

> This document assumes a reader with technical context. It prioritizes specificity over polish and is designed to help real evaluation conversations rather than marketing-level pitches.

PART 01

What a unified runtime does for agents

Harper's architecture produces five distinct capabilities for agentic workloads. Each one replaces a service or stack layer that teams building on conventional infrastructure assemble from separate vendors. Understanding what these five are, and what each replaces, is the foundation for evaluating whether Harper fits a given workload.

The capabilities are intentionally orthogonal. Each one earns its place on its own terms. Their value compounds when a workload uses multiple of them, which is what makes the unified runtime uniquely suited for agentic workloads.

01

CAPABILITY

Data Pre-Positioning

// WHAT IT IS

Harper can be configured to proactively pull data from systems of record (Salesforce, Zendesk, ERPs, Confluence, document stores) and maintains local copies that agents read from directly. The movement can be event-driven on change, scheduled on a cadence, or write-through as the agent operates.

// WHAT IT REPLACES

On conventional stacks, teams stitch this together from several pieces: a webhook receiver, a job scheduler, a cache tier (Redis or similar), a vector database for semantic indexing, and custom ETL code that reconciles the whole arrangement. Each piece is a separate service with its own scaling model and failure mode.

// WHY IT MATTERS

Systems of record were built for transactional workloads, not for the read patterns agents create. Agents iterate, re-read, and retrieve constantly. Pointing an agent directly at Salesforce can result in hitting API rate limits at scale, slow response times, and an operational burden on systems the rest of the business depends on. Harper absorbs that read load locally, so the systems of record keep doing the job they were designed for.

02

CAPABILITY

Agent-native data systems

// WHAT IT IS

The data agents generate for their own use: working memory, conversation history, derived facts, embeddings of past interactions, durable task state, audit trails of decisions and actions. This data has no home in any system of record because it did not exist before the agent created it. Harper provides NoSQL storage, blob storage, vector indexing, and local embedding generation as native properties of the runtime.

// WHAT IT REPLACES

The standard stack uses a memory-layer service (Mem0, Zep, Letta), a vector database (Pinecone, Weaviate), an object store (S3), and often a separate database for structured state. Each is a separate vendor with its own API, credential, billing line, and failure mode.

// WHY IT MATTERS

Agent-native data is the fastest-growing part of any production agent workload. Every conversation produces memory. Every decision produces state. Every interaction produces embeddings. Operating this sprawl across multiple services becomes the dominant cost center as agent volume grows. Harper collapses that sprawl into the runtime itself.

03

CAPABILITY

Native communication protocols

// WHAT IT IS

REST, MQTT, WebSocket, and SSE are first-class interfaces in the Harper unified runtime, allowing agents to speak any of them without additional infrastructure. This supports three distinct use patterns: fetch-on-demand from external systems, agent-to-agent coordination across a runtime, and real-time communication with clients or devices.

// WHAT IT REPLACES

Conventional stacks use a separate product for each protocol: a hosted WebSocket service for streaming, an MQTT broker for event-driven and IoT workloads, API gateways for REST, and custom SSE implementations where needed. Each has its own scaling profile, authentication model, and state management story.

// WHY IT MATTERS

Real agent workloads are rarely just request-response. They include streaming interfaces to end users, event-driven triggers from backend systems, coordination between agents, and sometimes direct communication with edge devices or external services. Running each protocol as a separate piece of infrastructure is operationally expensive and creates state-sharing gaps between systems. Harper treats all four protocols as expressions of the same runtime, so the agent, its state, and its interfaces stay in one place.

04

CAPABILITY

Deterministic logic in-process

// WHAT IT IS

The code that surrounds the LLM. Intent classifiers, policy validators, business rules, retry logic, SLA guards, workflow routers, long-running processes, custom endpoints, response post-processors. Many production agents are an LLM loop plus a substantial amount of deterministic code that handles the work the LLM should not or cannot handle.

// WHAT IT REPLACES

In distributed stacks, this code runs in Lambda functions, backend services, or framework middleware. Each piece is deployed separately, scales separately, monitors separately, and connects to the agent through network calls.

// WHY IT MATTERS

This is the most under-told part of Harper's value. Every serious agent deployment eventually develops a body of deterministic code that exceeds the agent code itself in size and importance. When that code lives alongside the agent and its data in one runtime, the efficient agent patterns (plan-then-execute, deterministic dispatch between model calls, fast validation loops) become natural. When it lives in a distributed environment, the same patterns become integration projects that most teams never complete.

05

CAPABILITY

Full-stack hosting

// WHAT IT IS

Harper can host frontend assets alongside agent code and data. Web interfaces, embedded widgets, admin tools, and chat clients deploy to the same runtime as the agents they talk to.

// WHAT IT REPLACES

Vercel, Netlify, Cloudflare Pages, or self-managed static hosting, paired with a backend on another platform that talks to the agent on yet another.

// WHY IT MATTERS

For a specific class of workloads (internal tools, embedded agents, full-stack agent products, single-team deployments), collapsing the frontend into the runtime reduces operational complexity dramatically. One deploy instead of three. One monitoring story. One security boundary. This capability does not matter for every workload, but where it matters, it compounds with the other four in value added ways.

The architectural claim, stated honestly

Each of these five capabilities is available, in some form, on a conventional distributed stack. Teams can and do build equivalent setups using separate vendors for each piece. Harper is not the only way to do any of this.

What Harper does is put all five in one runtime. The capabilities stop being separate projects and become properties of the platform. The efficient patterns that require all five to work together become the path of least resistance rather than a target that sophisticated teams approach over quarters of investment.

That is the specific thing Harper does. Not better features than any individual vendor, but a different architectural choice about where the pieces live.

// THE THESIS

Harper makes the techniques that make agents good **cheaper, faster, and easier to operate**, because the pieces they depend on share a runtime.

PART 02

The outcomes, ranked

Capabilities matter only insofar as they produce outcomes buyers and builders care about. This section ranks the ten outcomes Harper's architecture produces, from most to least impactful across the typical buyer profile. Each outcome is framed for both the developer audience and the business audience.

01 Efficient agent patterns become the default

The techniques that make agents good in production (parallel tool calls, precise retrieval, durable state, plan-then-execute, deterministic dispatch between model calls) stop being heroic engineering efforts and become the path of least resistance.

> WHY THIS RANKS HERE
>
> Compounds across every other outcome. Cost, latency, reliability, and quality all trace back to whether teams can actually implement the efficient patterns.

• DEVELOPER FRAMING

The right way to build an agent is the easy way to build an agent.

• BUSINESS FRAMING

Your team ships production-quality agents without needing a specialized agent-infrastructure team.

02 Lower total cost of running agents at scale

Fewer vendor bills (no separate vector DB, memory service, cache tier, orchestration platform). Fewer tokens burned on wasteful context assembly. Lower infrastructure overhead from operating fewer systems.

> WHY THIS RANKS HERE
>
> The only outcome on this list that is provable with a spreadsheet. Three independent cost-reduction mechanisms make the argument resilient.

• DEVELOPER FRAMING

Fewer vendors to operate, fewer tokens wasted re-assembling context.

• BUSINESS FRAMING

Your agent cost per interaction drops as you scale instead of climbing with volume.

03 Faster time from prototype to production

The work of turning a working prototype into a production system collapses. Less stack to assemble, fewer integrations to harden, fewer services to monitor, fewer credentials to govern.

> WHY THIS RANKS HERE
>
> This is where most agent projects die today. Working prototypes that never ship because the production gap is too long.

DEVELOPER FRAMING

Your agent runs in production the same way it ran in your demo, just bigger.

BUSINESS FRAMING

Agents ship on weekly timelines instead of quarterly ones.

04 Lower latency and better user experience

Agents respond faster because retrieval is in-process, state is local, and orchestration does not traverse a distributed system. Multi-step tasks complete in seconds where they used to take tens of seconds.

> WHY THIS RANKS HERE
>
> Faster agents can afford to iterate more, which makes them smarter. Latency compounds with quality.

DEVELOPER FRAMING

Your agent loop runs at memory speed instead of network speed.

BUSINESS FRAMING

End users wait less, which means they use the agent more and trust it more.

05 Reduced operational surface for security and compliance

Fewer credentials for agents to manage, fewer vendor boundaries data crosses, coherent audit trails in one runtime, data minimization as the default rather than the exception.

> WHY THIS RANKS HERE
>
> For regulated industries (finance, healthcare, government), this moves up the ranking significantly and can become the primary reason to buy.

DEVELOPER FRAMING

Your agent authenticates to the runtime, not to six backing services.

BUSINESS FRAMING

The surface area your security team has to certify shrinks dramatically.

06 Architectural simplicity and operational sanity

Engineers (and AI coding agents) can reason about the whole agent system as one thing. Debugging, monitoring, and incident response happen in one place. The system is small enough to hold in your head.

> WHY THIS RANKS HERE
>
> A slow-burning outcome. Rarely the reason teams buy, often the reason they renew and expand.

DEVELOPER FRAMING

You can reason about your whole agent system at once because it is one system.

BUSINESS FRAMING

Your platform team operates fewer moving pieces, so small teams can run big workloads.

07 Higher quality agent outputs

Agents produce better answers because retrieval stays fresh, context is more precise, and efficient patterns are the default.

> WHY THIS RANKS HERE
>
> Harper does not make the model smarter. It makes the inputs the model receives better. Real benefit, modest framing.

DEVELOPER FRAMING

Your agents get the right context to the model instead of the available context.

BUSINESS FRAMING

Agent quality improves because the operational constraints that forced compromises are gone.

08 Coherent observability and debuggability

Agent behavior, data access, model calls, and external integrations all happen in one runtime, which means one place to observe, log, trace, and debug.

> WHY THIS RANKS HERE
>
> Underappreciated at purchase time, heavily appreciated once running in production.

DEVELOPER FRAMING

Debugging an agent means reading one system's logs, not correlating six.

BUSINESS FRAMING

Your on-call engineer can diagnose agent issues without paging three teams.

09 Deployment and edge flexibility

Agents on Harper can run efficiently in decentralized configurations with Harper Fabric, and on edge devices or customer infrastructure with Harper-Pro. The same runtime works in all contexts.

> WHY THIS RANKS HERE
>
> Massively valuable for specific workloads (IoT, retail, manufacturing, regulated on-prem). Largely irrelevant for others. Position honestly.

• DEVELOPER FRAMING

Your agent runs anywhere the runtime runs, without rewriting.

• BUSINESS FRAMING

You choose where agents deploy based on your business needs, not your platform's limits.

10 Vendor simplification and procurement relief

One vendor to evaluate, contract with, negotiate with, and support instead of six or more separate agent-stack components.

> WHY THIS RANKS HERE
>
> Nobody buys Harper for this alone. Many buyers are relieved by it once they have.

• DEVELOPER FRAMING

One system to evaluate, integrate with, and monitor.

• BUSINESS FRAMING

Your procurement and vendor management overhead drops proportionally with your stack simplification.

What matters most by role

The ten outcomes distribute across two buyer profiles. Different audiences typically weight them differently.

// DEVELOPERS

Typically care most about

#1	Efficient patterns become default
#6	Architectural simplicity
#8	Coherent observability
#4	Lower latency

// BUSINESS LEADERS

Typically care most about

#2	Lower total cost at scale
#3	Faster time to production
#10	Vendor simplification
#5	Reduced security surface

// CROSS-CUTTING

Matter to both audiences

#4	Lower latency
#7	Higher quality outputs
#9	Deployment flexibility

PART 03

How capabilities map to workloads

Not every workload needs every capability equally. The matrix below shows how Harper's five key capabilities contribute to nine common agent workloads. Each cell rates the contribution as HIGH, MED, or LOW.

Use this matrix to identify which capabilities matter most for the agent you are building. A team building a support agent will find a very different story from one building edge IoT agents, even though both can run on Harper.

Workload	EXTERNAL DATA	AGENT-NATIVE DATA	NATIVE PROTOCOLS	DETERMINISTIC LOGIC	FULL-STACK HOSTING
Customer support at scale	HIGH	HIGH	HIGH	HIGH	MED
Internal knowledge agents	HIGH	HIGH	MED	MED	HIGH
Operational workflows (RevOps, FinOps, IT ops)	HIGH	MED	HIGH	HIGH	LOW
Data extraction and structuring	MED	HIGH	MED	HIGH	LOW
Edge and IoT agents	MED	MED	HIGH	HIGH	LOW
Real-time collaborative agents	MED	HIGH	HIGH	MED	HIGH
Sales prospecting and outreach	MED	MED	LOW	MED	LOW
Software engineering agents	LOW	MED	LOW	LOW	LOW
Research and synthesis agents	MED	HIGH	LOW	LOW	MED

HIGH Load-bearing MED Helpful LOW Marginal

Workload deep dives

The matrix gives the shape of the argument. The sections below give the substance. Each walks through how the capabilities produce outcomes for a specific workload, and names the reference architecture that typically results.

Customer support at enterprise scale

FLAGSHIP

Flagship workload for Harper. Every capability contributes meaningfully, and the combination produces something close to a category-defining architecture.

How it typically works on Harper. Customer records, tickets, and knowledge base articles flow into Harper via webhook sync from Salesforce, Zendesk, and Confluence. When a support request arrives via WebSocket from a chat widget, the agent assembles context from local reads in milliseconds. Deterministic code validates eligibility, checks policy, and routes to the right workflow. The LLM is called two or three times per conversation to reason and draft, not to orchestrate. Responses stream back via WebSocket. Resolutions write through to the ticketing system of record.

// OUTCOMES THAT LAND

- #4 Lower latency, retrieval and state are local
- #2 Lower total cost, token waste collapses
- #1 Efficient patterns by default
- #3 Faster time to production

Internal knowledge agents

STRONG FIT

The full-stack hosting capability is a surprise differentiator for this workload. Teams building "chat with our docs" tools benefit from the whole system (frontend, agent, embeddings, source content) collapsing into one deploy.

How it typically works on Harper. Documents from internal sources (Notion, Confluence, Google Drive, Slack archives) sync into Harper with embeddings generated locally. The frontend chat UI is hosted on Harper, served from the same runtime that answers queries. Query routing and access control run as deterministic endpoints. The agent assembles context from local vectors, answers the user, and logs the interaction for continuous improvement.

// OUTCOMES THAT LAND

- #6 One platform runs the whole product
- #3 One deploy replaces three
- #4 Vector search is in-process
- #7 Embeddings stay fresh on changes

Operational workflows (RevOps, FinOps, IT ops)

STRONG FIT

Deterministic logic does the heavy lifting in these workloads. Most of the work is rule-based with the LLM handling anomaly interpretation or natural-language routing. MQTT is frequently load-bearing for event-driven ops.

How it typically works on Harper. Records and events stream in from CRM, billing, and monitoring systems via webhooks and MQTT. Deterministic endpoints handle routine classification, routing, and validation. The LLM is invoked for anomaly interpretation, natural-language summaries, or cases that fall outside the rules. Actions write back to systems of record via REST. Agent coordination happens over MQTT topics inside the runtime.

// OUTCOMES THAT LAND

- #2 Dramatic, token usage minimized
- #5 Ops workflows touch sensitive systems
- #8 Debugging spans the workflow
- #9 On-prem when required

Data extraction and structuring at scale

STRONG FIT

Streaming ingestion workloads light up Harper's architecture unusually well. Document intelligence pipelines that would be overnight batch jobs on conventional stacks become real-time streams.

How it typically works on Harper. Source documents arrive via MQTT or webhook as they are produced. The agent extracts structured fields, generates embeddings, and checks for semantic duplicates, all in-process. Deterministic validators enforce schema and flag anomalies for review. Results write to downstream systems or are exposed via REST for consumption.

// OUTCOMES THAT LAND

- #2 Local embedding avoids per-token fees
- #7 Dedup and re-embed stay continuous
- #3 Stream-native pipelines ship faster
- #4 Real-time feed for downstream

Edge and IoT agents

UNIQUELY STRONG

This workload is uniquely strong on Harper because the combination of native MQTT plus edge-capable deployment via Fabric is rare in the market. Few competitors can credibly serve this workload without substantial custom work.

How it typically works on Harper. The Harper runtime deploys to edge nodes (retail stores, manufacturing sites, vehicles, building systems). Devices communicate via MQTT directly into the runtime. Deterministic control logic handles thresholds, rules, and immediate responses. The LLM is invoked for anomaly interpretation, natural-language reports, or remote diagnostics. Local state persists during connectivity interruptions and syncs to central systems when connectivity resumes.

// OUTCOMES THAT LAND

- #9 Deployment flexibility, decisive
- #4 Cloud round-trips infeasible
- #1 Edge budgets are tight
- #5 Regulated edge environments

Real-time collaborative agents

STRONG FIT

An emerging workload class where multiple humans and agents operate on shared state with live updates. Harper is architecturally well-positioned for this category even though the market is still forming.

How it typically works on Harper. Shared state lives in the runtime. Connected clients receive live updates via WebSocket. Agents participate in the same state as humans, reading and writing alongside them. Deterministic code enforces consistency and access control. The LLM is called for reasoning about edits, suggesting completions, or summarizing activity.

// OUTCOMES THAT LAND

#6 Shared-state painful on distributed stacks

#4 Live updates need in-process coordination

#3 Real-time products ship faster

Sales prospecting and outreach

MODERATE

Moderate fit. Harper works for this workload but does not produce dramatic architectural advantages. The value is largely in third-party enrichment APIs and model reasoning, neither of which Harper changes.

How it typically works on Harper. Account data syncs in from CRM. The agent calls third-party enrichment APIs on demand to fetch data that is not worth storing locally. Deterministic scoring and filtering narrow the list. The LLM drafts personalization. Outputs write back to the CRM for the human seller to review.

// OUTCOMES THAT LAND

#2 Lower total cost at high volume

#6 Tired of managing the stack

Position honestly. Harper is fine here. It is not unusually strong.

Research and synthesis agents

MODERATE

Moderate fit. Harper helps with retrieval quality and operational simplicity, though the ceiling on agent quality is set by the model's reasoning ability, which no runtime changes.

How it typically works on Harper. Source documents sync in with embeddings. Deep research queries trigger multi-step agent loops that retrieve, synthesize, and cite. Frontend hosting matters if the product is end-user-facing. Otherwise the value is retrieval freshness and operational coherence.

// OUTCOMES THAT LAND

#7 Retrieval stays fresh

#6 Small research-product teams

Software engineering agents

WEAK FIT

Weak fit. This is a product category more than a runtime category, and it is already well-served by purpose-built tools like Claude Code, Cursor, and Devin.

Why Harper is not the answer here. Most of the value in coding agents lives in the IDE integration, the model quality, and the tool-use loop against a local codebase. None of these are problems a unified runtime solves. Teams building coding agents are better served by dedicated products or by frameworks tightly coupled to developer tooling. If you are building a coding agent, Harper is probably not the right place to run it.

// VERDICT

Use a purpose-built coding-agent product or a framework coupled to developer tooling. Harper has no architectural advantage in this category.

// WORTH NOTING

Building software with agents is a different story.

Harper is not the place to run a coding-agent product, but it is unusually well-suited as the target coding agents build with. Conventional stacks force agents to reason across a database, cache, API layer, messaging system, auth, and deploy pipeline as separate services with separate credentials. More surface, more turns, more drift.

Harper collapses that surface. Schema, API, and business logic are declarative files in the repo. The agent works in code, not across services. Fewer turns to production, fewer architectural errors, no credentials in the loop, and what runs locally is what ships.

PART 04

How this compares to LangChain & the distributed stack

Most teams evaluating Harper are also considering, or already using, LangChain and LangGraph. This section is the honest comparison. It starts with what LangChain did well, names where the library-plus-vendors pattern breaks down, and describes how Harper composes with or replaces pieces of the typical LangChain stack.

What LangChain did, and why it matters

LangChain did the hardest work in the ecosystem. They abstracted the messy parts of working with LLMs (prompts, tool calls, chains, model switching, retrievers) into patterns developers could grab and use. Most of what developers know about how to build an agent, they learned from LangChain's tutorials, examples, and documentation. That contribution lowered the barrier to experimenting with agents from months of plumbing to a weekend.

LangChain helped define how developers build agents. Harper addresses a different layer: the runtime environment agents need as they move into production.

If your team is on LangChain today, that is not a mistake. It is how most teams should start. LangChain taught the industry how to think about agents, and that education is worth something.

// THE HANDOFF

LangChain helped define how developers build agents. **Harper addresses a different layer: the runtime environment agents need as they move into production.**

Where the library-plus-vendors pattern breaks down

LangChain is a library. It runs inside a Python or TypeScript application. That application needs a place to run, a database, a vector store, a memory service, an observability platform, possibly a model router, and integrations to every external system the agent touches. LangChain provides elegant abstractions for stringing these pieces together. The pieces themselves are someone else's product.

This architecture works well for first agents. It works well for prototypes. It scales adequately for teams running one or two agents at moderate volume. It strains at three specific points:

STRAIN POINT **A**

At agent-fleet scale

Running 20 agents across 6 vendors produces 120 operational relationships. Credentials, monitoring, billing, incident response, vendor updates, and compliance reviews all multiply. The overhead becomes the dominant cost center.

STRAIN POINT **B**

At efficient-pattern scale

Parallel tool calls, plan-then-execute, and deterministic dispatch between model calls all require low-latency access to data, vectors, and state. Achieving that on a distributed stack requires heroic engineering. Most teams settle for naive patterns and absorb the cost.

STRAIN POINT **C**

At enterprise-trust scale

Each vendor in the stack is a separate subprocessor, a separate audit trail, a separate breach-notification chain, and a separate security review. Regulated industries hit this wall quickly.

These are not LangChain's faults. They are consequences of the architectural assumption that agents are applications running on top of a distributed infrastructure, rather than workloads running inside a purpose-built runtime.

How Harper relates to an existing LangChain stack

Harper is not a LangChain replacement in the sense that a team throws out their LangChain code on day one. Most production migrations happen incrementally. Harper typically enters a team's stack in one of three ways:

PATTERN
01

Harper as the runtime, LangChain as the orchestration library

LangChain has a first-party TypeScript implementation (LangChain.js) that runs natively on Node-based runtimes like Harper. Teams can keep LangChain's abstractions for prompt composition, tool definitions, chains, and model calls, while Harper provides the runtime those components execute in: the data, the vectors, the memory, the endpoints, the protocols. The team keeps what they know and swaps what was hurting them.

Most common entry point for teams with LangChain investment.

PATTERN
02

Harper for a new agent, LangChain for legacy agents

A team with existing LangChain agents builds their next agent on Harper directly. Both run in parallel. Over time, as the LangChain agents hit operational walls, they migrate to Harper one at a time.

Common for teams adding agents to an existing product rather than rebuilding.

PATTERN
03

Full Harper build

New agent projects starting from scratch often skip LangChain entirely and build on Harper directly. The LangChain abstractions are valuable for learning how agents work; once a team understands that, Harper-native development is frequently simpler because the runtime already solves the problems LangChain's abstractions are designed to hide.

Common for greenfield platform-team builds and opinionated teams.

How Harper compares to the other alternatives

Beyond LangChain, teams evaluate several other categories of tools. A quick honest take on each.

LangGraph and stateful workflow engines

LangGraph brings legitimate engineering discipline to agent orchestration. Nodes, edges, checkpointing, and state machines solve real problems that naive agent loops have. For teams committed to a library-based architecture, it is the right answer.

The honest comparison is that LangGraph and Harper solve the same problem in incompatible ways. LangGraph owns agent state through its graph abstraction and pluggable checkpointers, which live inside your application process. Harper owns agent state as a property of the runtime itself, through in-process data, memory, and durable endpoints. These are two different architectural bets on where workflow state belongs.

Running LangGraph on Harper may make sense for teams that want to preserve an existing graph-based orchestration model. However, teams get the most architectural benefit from Harper when state, data, logic, and endpoints are treated as runtime-native rather than layered through an external workflow abstraction.

For teams picking between the two, the question is less about features and more about the scaling profile. LangGraph scales well when the workflow engine is the hard part and the surrounding data is straightforward. Harper scales well when the surrounding data, state, and integrations are the hard part and the workflow logic is tractable. Most production agents are the second shape, not the first.

Memory-layer services `Mem0 · Zep · Letta`

These services solve the real problem that agents need durable structured memory. They do it by being a separate service the agent calls out to, which produces the sidecar pattern and all its costs. Harper's architecture makes durable memory a property of the runtime rather than a service. Teams already invested in a memory layer can keep using it with Harper, but most discover they no longer need it once they have the runtime.

Vector-database vendors `Pinecone · Weaviate · Qdrant · Elastic`

Best-in-class vector databases do vector search extremely well. Harper's vector capabilities are competitive for agent workloads, not necessarily for every vector workload. The distinction is that vectors in Harper live next to the source data, the agent code, and the memory, which makes them cheaper to operate and easier to keep fresh. Dedicated vector databases remain the right choice for workloads that are purely about vector search at massive scale.

Vertical-vendor agent frameworks Elastic AI · Snowflake agents · MongoDB agents

Each of these extends a data platform with agent capabilities scoped to data in that platform. They are good for single-domain retrieval-flavored agents. They struggle when agents need to cross data domains or write to multiple systems of record, because their architecture assumes their platform is the center of gravity. Harper's position is neutral runtime: agents coordinate across whatever data they need, regardless of where it originates.

Hosted agent platforms OpenAI Assistants API · AWS Bedrock Agents · Google Vertex

Hosted platforms optimize for rapid adoption and tight integration with their own ecosystems. They work well for teams building agents as features of existing applications in those clouds. They are weaker when agent workloads need portability, when they need to integrate with data outside the hosting cloud, or when they need deterministic custom logic that the platform does not support. Harper is the architectural alternative to these: a runtime you control rather than a service you call.

PART 05

When Harper is the right choice

Harper is not the right runtime for every agent or every team. Being clear about fit helps both sides. Teams that are not yet ready for Harper should not be forced into it; teams that are ready should not be talked out of it.

✓ Strong fit

Harper is the right choice when most of the following are true:

- → **Production volume is real,** or will be within 6–12 months. Operational benefits compound with scale.
- → **The workload is data-heavy or retrieval-heavy.** Agent quality depends on getting the right context quickly.
- → **The workload involves more than one protocol.** Real-time clients, event-driven triggers, or devices alongside REST.
- → **There is meaningful deterministic code** around the LLM. Policy validation, routing, rules, or long-running workflows.
- → **Operational constraints matter.** Compliance, audit, data residency, credential governance, or on-prem.
- → **The team has felt the pain** of operating a distributed agent stack. Teams with three or more agents in production see it acutely.

✗ Not yet a fit

Harper is not the right choice when any of the following are true:

- × **The goal is a weekend prototype** or proof of concept. Whatever has the most tutorials is the right tool.
- × **A single simple workflow with no real scale.** The simplification does not yet outweigh the cost of a new runtime.
- × **The use case is purely model-bound.** Coding agents, research agents, creative writing where value is in model reasoning.
- × **The team is fully committed to a specific cloud's** hosted agent platform and does not value portability.

// DECISION FRAMEWORK

A practical way to decide

Three questions that usually clarify the fit quickly.

// ASK THESE THREE QUESTIONS

If two or more are yes, Harper is worth the conversation.

1. Will this agent be in production serving real users within a year?
2. Does its quality depend on fast access to your data and your state?
3. Would your team benefit from long-term operational simplicity?

3 yes	2 yes	0–1 yes
Worth serious evaluation.	Worth a conversation.	Conventional stack for now.

// CLOSING NOTE

The arguments in this guide are architectural, not marketing.

Harper does not make models smarter, does not eliminate prompt engineering, and does not replace systems of record. What Harper does is collapse the distributed stack that agents traditionally run on into one runtime with five first-class capabilities ideal for agentic workloads, and that collapse produces real, measurable outcomes for workloads that fit the runtime's shape.

For teams evaluating where to run their next production agent, the question is not whether Harper is better than LangChain, or better than Pinecone, or better than a hosted platform. The question is whether the unified-runtime bet is the right architectural bet for the workload being built. This guide is designed to make that question answerable.

// THE BET, IN ONE SENTENCE

On Harper, the right way to build an agent is **the easy way to build an agent.**