International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 7, Issue 2, 197-201, 2026
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V7I2P129

Original Article

Serverless Architecture Patterns for Enterprise AI Agents: ECS Fargate, OpenSearch k-NN, and DynamoDB for Knowledge-Grounded LLM Workflows

Raj Sunkara
Independent Researcher, USA.

Received On: 18/03/2026 ***Revised On: 19/04/2026*** ***Accepted On: 26/04/2026*** ***Published On: 04/05/2026***

Abstract: *Production deployment of enterprise AI agents that combine large language models with proprietary knowledge bases presents specific architectural challenges. State has to be managed across asynchronous workflows. Retrieval latency has to be kept low enough to support interactive use. The knowledge base has to stay fresh as the underlying source systems change. Costs have to be predictable. The agent has to be observable and recoverable when individual steps fail. This paper describes a next-generation serverless reference architecture for a domain-specific AI agent. The architecture is built on AWS ECS Fargate for compute, Amazon OpenSearch with k-Nearest Neighbor search using Hierarchical Navigable Small World indexing for vector search, Amazon Titan embeddings for semantic representation, Amazon DynamoDB for workflow and conversation state, and an automated ingestion pipeline that synchronizes issue trackers, architecture documentation, and source code repositories into the knowledge base. The paper describes the end-to-end data flow from ingestion through embedding, retrieval, language model invocation, and response persistence. It discusses the design choices around HNSW parameter tuning, embedding refresh strategy, Fargate cold-start mitigation, isolation between tenant workloads, and incremental knowledge base updates without full reindexing. The architecture supports a graphics engineering bug triage agent that builds on the Retrieval-Augmented Generation approach described in earlier work and extends it with the operational properties that production deployment at scale requires. The contribution is a deployment-tested blueprint for teams building RAG-based agents on AWS, with attention to operational concerns that frequently receive less coverage in prototype-focused literature.*

Keywords: *Serverless Architecture, AWS, ECS Fargate, OpenSearch, k-NN, HNSW, Vector Search, Amazon Titan Embeddings, DynamoDB, Retrieval-Augmented Generation, RAG, AI Agent, Knowledge Base, Ingestion Pipeline, AI Infrastructure.*

1. Introduction

Enterprise AI agents that combine a large language model with a proprietary knowledge base have moved from prototype to production over the past two years. The architectural shape of such an agent is by now well understood. An ingestion pipeline embeds the knowledge base. A retrieval layer looks up relevant content for each request. A language model is invoked with the retrieved content as grounding. The response is persisted along with the conversation state.

The shape is well understood, but the production-quality realization of that shape is where most of the engineering work lies. Each layer in the architecture has decisions to make. Ingestion has to be incremental, because re-embedding the whole knowledge base every time something changes is wasteful and slow. Retrieval has to be fast enough for interactive use, which constrains the vector index parameters. Language model invocation has to be reliable, which means handling rate limits and transient errors. State storage has to support the actual workflow the agent runs, which usually has more structure than a chat history. Costs have to be predictable, which favors serverless components whose pricing scales with usage rather than provisioned components whose pricing scales with capacity.

This paper describes a next-generation serverless reference architecture for a domain-specific AI agent. The architecture is the target architecture for a graphics engineering bug triage agent that previously ran on a different stack. The triage agent itself, including its weighted confidence scoring algorithm and its Flask dashboard, is described in earlier work. The focus of this paper is the infrastructure that supports the agent in production at scale, not the agent's domain logic. The rest of the paper is organized as follows. Section 2 describes the architecture at a high level. Section 3 covers the ingestion pipeline. Section 4 covers vector search with OpenSearch and HNSW. Section 5

covers compute on ECS Fargate. Section 6 covers state management with DynamoDB. Section 7 discusses operational concerns including isolation, cold-start mitigation, and incremental updates. Section 8 concludes.

2. Architecture Overview

The architecture has five primary components. The ingestion pipeline reads source systems and writes embeddings to the vector store. The vector store provides retrieval at request time. The compute layer runs the agent code. The state store holds workflow and conversation state. The language model is invoked through AWS Bedrock. A request to the agent flows through the compute layer, which retrieves relevant content from the vector store, invokes the language model with the retrieved content, persists the result to the state store, and returns the response to the caller.

2.1. Why Serverless

The architecture is serverless in the sense that the compute, the vector store, and the state store are all managed services whose pricing scales with usage. The motivation is operational. Managed services remove the need for a separate team to operate the infrastructure. Usage-based pricing matches cost to value when the request volume is variable. Auto-scaling removes the need to provision for peak. The tradeoff against provisioned alternatives is that some performance characteristics, in particular cold start latency, depend on the managed service rather than on the agent team's choices.

2.2. Why AWS

The architecture is built on AWS because AWS Bedrock provides the language model and the embedding model the agent uses, because OpenSearch Serverless provides the vector store, and because the rest of the organization is already on AWS. The architecture would translate to other cloud providers with equivalent services, but the integration between Bedrock, OpenSearch, and the other AWS services is what makes the AWS realization straightforward.

3. Ingestion Pipeline

The ingestion pipeline brings content from the source systems into the vector store. The sources are an issue tracker, architecture documentation, and source code repositories. Each source has its own connector that pulls content from the source on a schedule.

3.1. Sources

The issue tracker source pulls resolved issues, including title, description, resolution text, and metadata. Each issue becomes a document in the vector store. The architecture documentation source pulls the relevant documents from the documentation system. Each document is chunked into sections that are individually retrievable. The source code source pulls the code from the relevant repositories. Code is chunked at function or module granularity and indexed both as text for keyword search and as embeddings for semantic search.

3.2. Incremental Updates

Full reindexing of the knowledge base is expensive. The pipeline performs incremental updates whenever possible. Each source connector tracks the last successful synchronization timestamp and pulls only content that has changed since that timestamp. Changed content is re-embedded and the new embeddings replace the old ones in the vector store. Deleted content is removed from the vector store. The incremental approach keeps the freshness lag short while keeping the cost of synchronization low.

3.3. Chunking Strategy

The chunking strategy matters because it determines what unit of content is retrieved at request time. The pipeline chunks content at semantically meaningful boundaries: issues at the issue level, architecture documents at the section level, code at the function level. Chunking at smaller granularity would increase the retrieval signal-to-noise ratio for narrow queries but would also fragment content that is more useful in larger pieces. The chunking decisions are made per source rather than globally.

3.4. Embeddings

Embeddings are produced using Amazon Titan embedding models on Bedrock. Each chunk is embedded once at ingestion time and the embedding is stored alongside the chunk in the vector store. The choice of embedding model is consequential because changing the model invalidates all existing embeddings and requires a full re-ingestion. The pipeline therefore treats the embedding model choice as a long-term commitment and includes a migration path for the rare cases where the model is replaced.

4. Vector Search with OpenSearch and HNSW

Retrieval at request time is performed by Amazon OpenSearch using k-Nearest Neighbor search with Hierarchical Navigable Small World indexing. HNSW is a graph-based approximate nearest neighbor algorithm that provides fast retrieval at the cost of small accuracy loss compared with exact search.

4.1. Why Approximate Nearest Neighbor

Exact nearest neighbor search over a large embedding collection is too slow for interactive use at scale. Approximate algorithms trade a small loss in retrieval quality for a large gain in speed. For the agent's use case, where the retrieved content is fed to a language model that further reasons over it, the small accuracy loss from approximation is acceptable because the language model is tolerant of retrieval that includes a few near-misses.

4.2. HNSW Parameter Tuning

HNSW has two principal parameters that the operator tunes: the maximum number of neighbors per node in the graph, and the size of the dynamic candidate list during search. Larger values for both improve recall at the cost of indexing time, memory, and search latency. The parameters were tuned by measuring recall against an exact nearest neighbor baseline on a representative query set and then choosing the smallest values that achieved acceptable recall.

Acceptable recall in this context was defined as the language model producing equivalent outputs when given the approximate retrieval results as when given the exact results, on the calibration query set.

4.3. Hybrid Search

OpenSearch supports keyword search as well as vector search. The agent uses hybrid search, combining keyword matches against the same indexed content with vector matches, and re-ranking the combined results. Hybrid search handles queries where the relevant content contains specific identifiers, function names, or error codes that vector search alone may not weight highly enough. The combination is more reliable than either approach alone.

5. Compute on ECS Fargate

The agent code runs on AWS ECS Fargate. Fargate is a serverless container compute service that runs containers without requiring the operator to manage the underlying virtual machines. The agent is packaged as a container image and Fargate provisions the runtime as needed.

5.1. Why Fargate

Fargate is the right fit for the agent's compute because the workload is request-driven and the cost of an always-on provisioned cluster would be high relative to the actual request volume. Fargate scales the number of running tasks with the request rate, and the operator pays for the CPU and memory consumed rather than for provisioned capacity.

5.2. Cold Start Mitigation

Serverless compute has a cold start cost. The first request after a period of inactivity pays the cost of provisioning a new container. For interactive use this cost can be visible. The agent mitigates cold start with two strategies. The first is keeping a small number of tasks always warm, so that the first request lands on a warm task. The second is keeping the container image small and the application start-up time short, so that the cost of a cold start is bounded. Both strategies are operational decisions that trade some always-on cost for predictable latency.

5.3. Concurrency

Each Fargate task can serve multiple concurrent requests. The concurrency level is chosen based on the memory footprint of the agent and the I/O profile of its requests. The agent is I/O-bound during retrieval and language model invocation, which favors higher concurrency. The actual choice is set so that a single task can absorb modest concurrency without thrashing, and additional tasks are provisioned by the auto-scaler as the rate increases.

6. State Management with DynamoDB

State that the agent needs to persist is stored in Amazon DynamoDB. The state includes workflow state for in-flight triage operations, conversation state for multi-turn interactions, and the audit log of all triage outputs.

6.1. Workflow State

A triage operation has a small number of steps, each of which produces intermediate state. Storing this state in DynamoDB allows the operation to be resumed if the Fargate task processing it fails between steps. The workflow record carries the current step, the inputs to that step, and the outputs of all previous steps. Recovery is a matter of reading the workflow record and resuming at the recorded step.

6.2. Conversation State

Conversation state stores the back-and-forth history of multi-turn interactions with the agent. The state is keyed on the conversation identifier and contains the messages exchanged so far. The conversation state is consulted at the start of each request to provide the language model with the relevant prior context.

6.3. Audit Log

Every triage the agent performs is logged to DynamoDB. The audit log is the source of truth for the dashboard analytics described in companion work on the agent's domain logic, and it is the basis for the calibration of the confidence scoring algorithm over time. The log is structured so that filters by component, time range, confidence band, and outcome category are efficient.

7. Operational Concerns

7.1. Isolation between Tenant Workloads

The agent is designed to support multiple tenant workloads, where a tenant in this context is a team or sub-team whose knowledge base and workflows are logically separated from those of other tenants. Isolation is implemented at the DynamoDB level by partitioning state on tenant identifier, at the OpenSearch level by using separate indices per tenant where required, and at the Fargate level by ensuring that no in-memory state carries across requests from different tenants. Cross-tenant retrieval is explicitly prevented in the retrieval layer.

7.2. Incremental Knowledge Base Updates

Incremental updates to the knowledge base are the default. The ingestion pipeline described in Section 3 brings new and changed content into the vector store without full re-indexing. This is what keeps the operational cost of knowledge base maintenance bounded. Full re-indexing remains an option for the rare cases where the embedding model is changed, but it is not part of the normal operational cadence.

7.3. Observability

The agent emits structured logs for every retrieval, every language model invocation, and every workflow state transition. The logs are the input to the dashboard described in companion work, and they are also the input to the operations team's investigation of any failures. Observability is treated as a first-class concern in the architecture rather than as an afterthought.

7.4. Cost Predictability

Costs are predictable because every component of the architecture has usage-based pricing. The dominant cost components are the language model invocations on Bedrock, the embedding generation during ingestion, the Fargate compute time, and the OpenSearch vector store. Each of these scales with usage rather than with provisioned capacity. The cost model lets the agent's operator predict the marginal cost of additional request volume.

7.5. Failure Modes and Recovery

The architecture is designed to recover from failures in individual components without losing user-visible work. A Fargate task failure mid-workflow is recovered by reading the workflow state from DynamoDB and resuming on a different task. A transient OpenSearch unavailability is handled by retrying retrieval. A Bedrock throttling event is handled by backing off and retrying. The user-visible effect of these recoveries is increased latency rather than failed requests.

7.6. Knowledge Base Versioning

The knowledge base evolves over time as new content is ingested and old content is updated or removed. The architecture stores a version stamp on each ingested chunk so that the agent's outputs can be traced back to the specific version of the knowledge base that produced them. This is what allows the operations team to investigate cases in which the agent's output disagrees with the eventual ground truth, by reproducing the retrieval against the version of the knowledge base that was in effect at the time of the original request. Without this versioning, the investigation would be blind to changes that happened between the original request and the investigation.

7.7. Security and Access Control

The architecture treats access control as a per-component concern enforced at the API surface of each managed service. OpenSearch indices are protected by IAM policies that limit which Fargate tasks can read and write them. DynamoDB tables are similarly protected. Bedrock invocations are gated by IAM roles assigned to the Fargate task. The agent's own application code does not reimplement these controls; it relies on the underlying service to enforce them. This avoids a class of bugs in which the application's access control diverges from the actual permissions granted by the platform.

7.8. Migration from the Previous Stack

The architecture described here is the target architecture for the bug triage agent. The migration from the previous stack is staged so that production traffic continues to be served while individual components are cut over. Embeddings are regenerated under the new architecture before the retrieval layer is switched over to use them. Workflow state is read from both stores during the cutover window so that in-flight workflows complete on whichever store they started on. The migration plan trades some duplication of cost during the cutover window for the absence of user-visible disruption, which is the right tradeoff for a workload that the team depends on day to day.

8. Conclusion

Serverless architecture is a good fit for enterprise AI agents that combine a large language model with a proprietary knowledge base. The combination of ECS Fargate, OpenSearch with HNSW vector search, Amazon Titan embeddings, DynamoDB for state, and Bedrock for language model and embedding generation provides a deployment-tested blueprint that handles the production concerns that prototype-focused literature often leaves implicit. The ingestion pipeline keeps the knowledge base fresh through incremental updates. The HNSW parameters are tuned for the right balance of speed and recall. Cold start latency is bounded by warm-pool and small-image strategies. State is managed in DynamoDB so that workflows are recoverable. Isolation between tenant workloads is explicit at each layer. Costs are predictable because every component is usage-priced. Teams building similar agents on AWS can use this architecture as a starting point and adjust it to the specifics of their own knowledge base, request volume, and tenancy model.

Acknowledgments

This work is the target architecture for a graphics engineering bug triage agent. The author thanks the engineers who contributed to the original agent and to the migration planning, and the platform teams that provide the AWS environment in which the architecture runs.

References

1. Amazon Web Services. Amazon ECS and AWS Fargate documentation.
2. Amazon Web Services. Amazon OpenSearch Service documentation, including k-NN search.
3. Amazon Web Services. Amazon Bedrock documentation, including foundation model integration and Knowledge Bases.
4. Amazon Web Services. Amazon Titan embeddings documentation.
5. Amazon Web Services. Amazon DynamoDB documentation.
6. Malkov, Y. A. and Yashunin, D. A. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2018.
7. Lewis, P. et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS 2020.
8. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need. NeurIPS, 2017.
9. Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. NAACL-HLT, 2019.
10. Brown, T. B. et al. Language Models are Few-Shot Learners. NeurIPS, 2020.
11. Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W. Dense Passage

Retrieval for Open-Domain Question Answering. EMNLP, 2020.
12. Johnson, J., Douze, M., and Jegou, H. Billion-Scale Similarity Search with GPUs. IEEE Transactions on Big Data, 2021.
13. Reimers, N. and Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. EMNLP-IJCNLP, 2019.
14. Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., and Wu, C. Serverless Computing: One Step Forward, Two Steps Back. CIDR, 2019.
15. Jonas, E. et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv:1902.03383, 2019.