Prescriptive guidance for AgentOps on AWS

Q3 2025

Author

Vikesh Pandey, Pr. GenAI Specialist Solutions Architect, AWS

Purpose

This paper provides an overview of AgentOps best practices on AWS. It is targeted to enterprise customers of AWS, including but not limited to those operating in regulated industries such as financial services, healthcare etc. This paper presents a holistic end-to-end perspective, from which customer may utilize a subset of these capabilities for their implementation.

Intended Audience

This document is targeted towards technical decision makers within the organization; personas including Head of Platform, Head of Engineering, Platform Lead, Principal Engineers, Solution Architects or similar.

Overview

AI Agents have captured the attention of large financial organizations, and many such organizations are experimenting with various agentic prototypes. An AI agent is a software program that can interact with its environment, collect data, and use the data to perform self-determined tasks to achieve a goal. Humans set these goals, and an AI agent independently chooses and performs the best action(s) to achieve those goals.

Many organizations have already tried various agentic frameworks like LangGraph, CrewAI, AutoGen, OpenAI Agents, Google ADK, Amazon Bedrock Agents, AWS Strands Agents and many more. They have built siloed agentic PoCs, some of which are in the process of going into production. As customers are putting AI agents into production, they are realizing that running agentic applications in production is more complex than building them due to many moving parts, and the lack of standardized operational practices for AI Agents. It is also creating duplication where teams are re-creating same/similar components over and again in each Line-of-Businesses (LoBs). Hence, there is a need to provide a centralized platform with foundational capabilities which can be used by different use-case teams to build and deploy agentic applications faster, while addressing cross-cutting concerns like governance, observability, auditability, cost tracking, scalability, resiliency, security and operational excellence. AgentOps refer to the practices required when building and running multiple AI agents in production, in a standardized and scalable way, while maintaining governance and controls. It takes inspiration from architecture patterns like Enterprise Service Bus and Integration Brokers.

This document presents two main actors in the AgentOps solution:

- **Platform team**: This team is responsible for providing all the common tools needed for building, running and maintaining agentic applications. The tooling can be centrally hosted by the platform team or provided as infrastructure blueprints to the teams building the agentic applications. It will also be responsible for defining processes and policies for using these blueprints.
  Based on the experience working with various enterprise customers, the platform team represents the platform team per LoB and not organization wide. We have enough customer evidence from the past to prove that org-wide central platforms do not scale well with increase in adoption and end up becoming a bottleneck. Though the IT team is generally central to the organization. This IT team is responsible for basic networking, infrastructure and security and Git setup but does not handle platforms for specific functions (Data, AI, analytics etc.)
  In some organizations, there are multiple platform teams available in LoBs which can share the responsibilities of the platform components and activities.

- **Use-case team**: This can be any team which is part of a Line-of-Business (LoB) working on solving one or more business use-cases using agentic capabilities.

AgentOps (conceptual view)

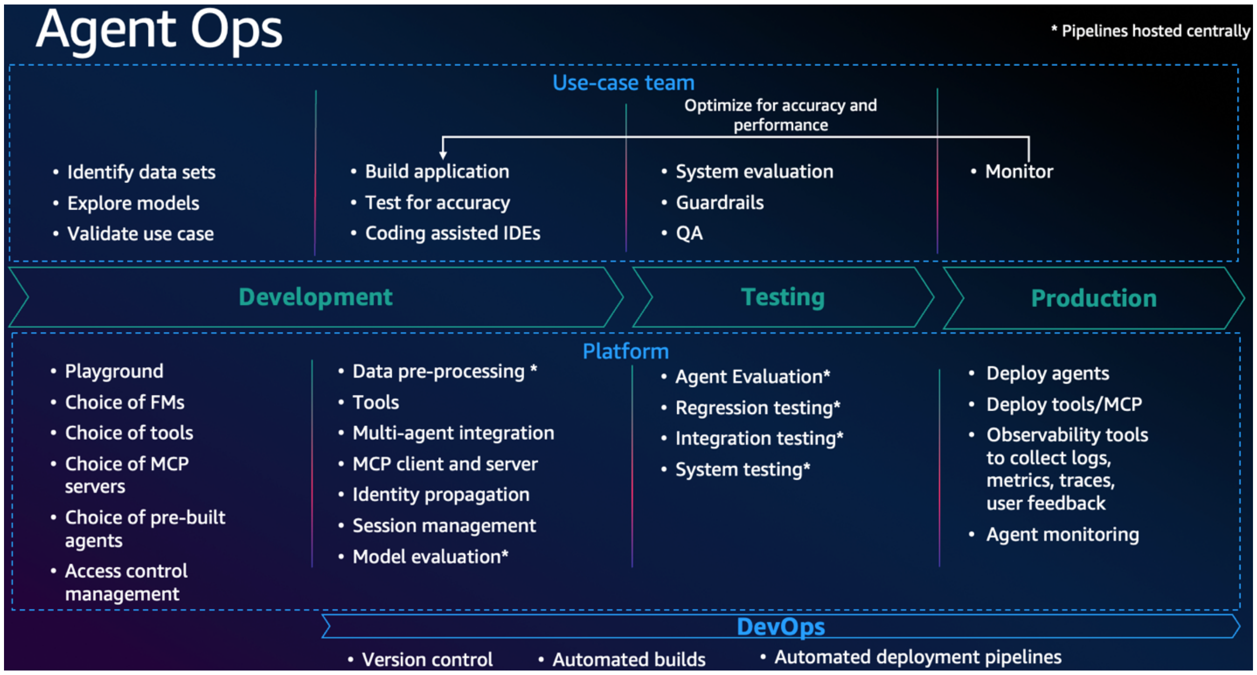

Figure 1: AgentOps (conceptual view)

From Figure 1, the agentic application lifecycle is presented as a linear software development lifecyle and is divided vertically in three phases, and it is also showing the responsibilities of the platform and use-case team as two horizontal swim lanes. Below is the definition for platform and use-case teams:

In each of the three phases, the use-case team and platform team need to collaboratively perform activities which help the application move forward to higher environments (i.e. test, production). The phases are described below:

- **Development:** In this phase, the use-case team is focused on picking the right set of tools, MCP servers, agentic framework, foundation model (FM), while using pipelines for data processing with their data and code, model evaluation and building the agentic application. Important to note that the platform team will provide a catalog and runtime for the tools but will not create all the tools. The use-case teams are expected to create tools and get them registered in the central platform catalog.
- **Testing:** In this phase, the agentic application is tested from various functional and non-functional aspects to ensure the desired outcome is achieved. It must be noted that development and testing phases are not linear and can also be circular where teams iterate between development and testing phases till functional requirements are met.
- **Production:** In this phase, the agentic application is deployed to a target environment with observability, logging and auditability tools enabled. The functional monitoring is also enabled to detect any degradation in the application's performance.

***NOTE**:* In later sections, this paper will go deeper on the activities performed in each environment. Apart from the conceptual understanding of AgentOps, it is important to understand how this would look like in customer environments, from an operational standpoint. Topics around security are explained in more details, in the later sections of the document.

AgentOps (operational view)

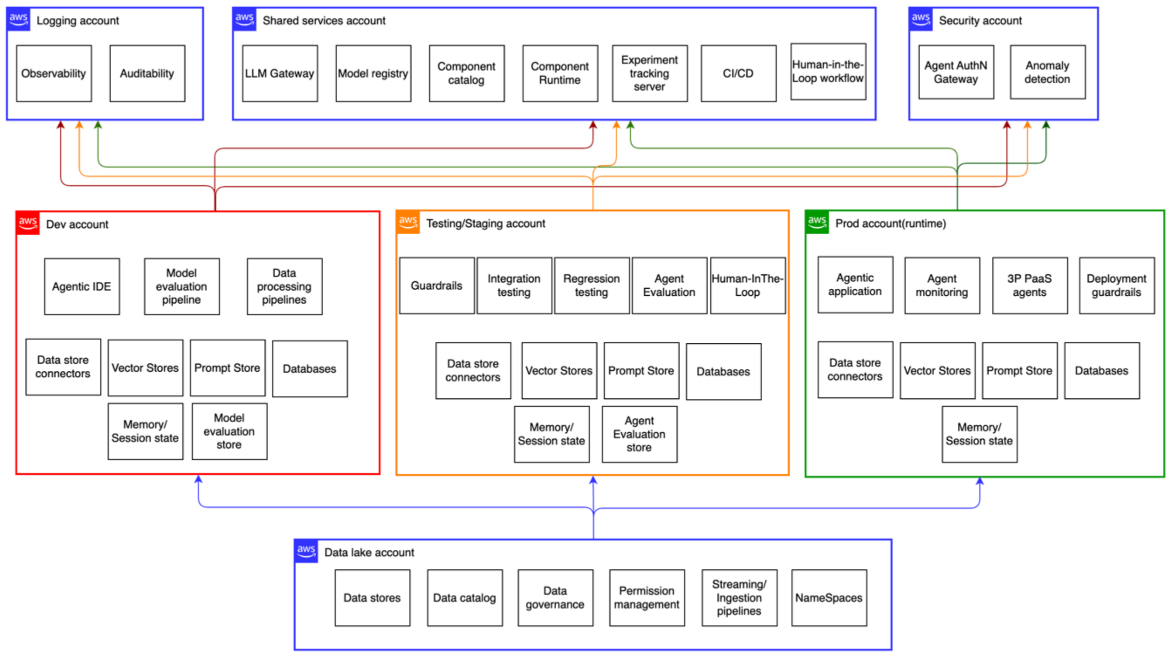

Figure 2: AgentOps (operational view)

As per figure 2, the AgentOps components have been grouped logically in an AWS account structure to provide inspiration to the reader to understand how the responsibilities are divided and grouped together. The architecture is divided in platform accounts and use-case accounts.

- **Platform team accounts**: These accounts are owned by platform team and responsible for providing components which are shared across use-case teams. The integration between components across platform team accounts, must be done by the platform team themselves. The accounts highlighted (in blue) are the production accounts for the platform. The team must have their own lifecycle of dev->staging->prod platform accounts to roll out the platform components to the use-cases team.
- **Use-case team accounts**: These accounts are owned by use-case teams containing applications and data specific to their LoB.

It is important to note that building such a platform would require significant amount of effort. We strongly recommend not to build the entire platform in the first iteration. Instead, the recommendation is to build it in an iterative fashion.
Before building such platform, the organization/LoB must identify few use-cases which would be running on this platform. The use-cases must be of medium-high complexity and the platform team must pick only one use-case to start with. Once the use-case has been selected, the platform team must work backwards from this use-case's requirements to build the platform. The platform team must only build the components needed for the use-case. Once the first use-case has been launched in production, the platform team must observe the scalability, resilience and functional maturity of the platform through diagnostic tools. Once the platform's stability has been proven on the first use-case, the next wave of use-case should be chosen to onboard on to the platform. Based on the experience of building such platforms with large enterprises, the first iteration of the platform may take 3-6 months and to attain full maturity with a handful of use-cases (~10), it may take between 12-18 months. These timelines are based on assumptions of all the business and IT processes aligned with the build of the platform along with resourcing in the place.

Platform team accounts

Let us discuss all the accounts mentioned as platform team accounts.

1/Shared services account: This account represents all the common technical tooling which is needed by all use-case teams. The platform team manages this account from an operational standpoint. This account also holds all the deployment topologies (namespace creation, code repositories etc.). Let us understand the components of this account:

- **FM Gateway**: Most enterprise organization today have some form of an FM gateway to allow access to various FMs through a standard API. The agentic applications from LoBs would be required to access FMs through this gateway. FM gateways must satisfy requirements of the enterprises around scaling, security, cost tracking, governance etc. Some of the concerns are covered below:

    - **Hosting**: FM gateways can be hosted either centrally by the platform team or individually for tenants, depending on the operating model and size and number of tenants. As a starting point, the recommendation is to host it centrally per LoB. Number of use-cases should not be used as a decisioning criteria to decide to decentralize the FM gateways. It should be driven from usage of FMs by use-cases. If the platform team observes that a single use-case is consuming high amount of gateway bandwidth, then it is advisable to have a dedicated FM gateway for such use-case by providing the use-case team, the blueprint as Infrastructure-as-a-Code (IaaC) and they can run their own FM gateways in their environments. In this case, the use-case team must own the responsibility of keeping the gateway up to date with latest gateway enhancements released from the platform team. For self-hosted models, the models should be deployed in the use-case environment.

    - **Scaling**: The gateway must be scalable. Using AWS Application Load Balancer (ALB) is the recommended way to setup FM gateways. In case of self-hosted models, those can be hosted on Amazon SageMaker AI Endpoints and exposed as an endpoint through the ALB. The gateway should be able to handle throttling fallbacks gracefully to lower consumption tiers of the models.

    - **Security**: The gateway must support authentication and authorization mechanism as prescribed by the organization. API key-based mechanism is a good place to start, with using OAuth2 and IAM together strengthening the security perimeter of the gateway. All the communication with the gateway must happen through private endpoints. Direct access to models shall be blocked by default in any environment.

    - **Cost tracking and governance**: FM gateways must provide cost tracking and attribution per tenant so they can charge back each use-case team appropriately. The gateway must implement rate limits, throttling and caching, intelligent routing, guardrails baseline capabilities as well.

  FM gateway tools like LiteLLM, Kong, Openrouter, Helicone provide lot of the above mentioned capabilities natively, and can be explored to implement the FM gateway. AWS also provides Guidance for Multi-Provider Generative AI Gateway on AWS as a reference implementation which platform team can refer to.

- **Model Catalog**: The onboarding and deboarding of models (serverless, self-hosted, fine-tuned models) should be tracked and stored through a model catalog. This model catalog will store metadata about the models, their versions, inference configurations, lifecycle details etc, hence providing a central catalog for all models used in the gateway and providing visibility to all the actors (platform, use-case) about the model's availability status in the gateway. This catalog can also contain details about which applications (RAG, agents) are using a particular FM. It contains details for all serverless, self-hosted, fine-tuned models in a single place. For implementing model catalog, MLFlow, SageMaker model registry or AWS DynamoDB can be used. The platform team must provide a permission model to allow use-cases to access the catalog.

- **Component Catalog:** This catalog contains information about various components used in agentic applications. Here the word component represents: a tool, an MCP (Model Context Protocol) server or an AI

agent. Every use-case team must have access to this catalog where they can browse and search for preferred component of choice. Here the components can be either PaaS (Platform-as-a-Service) or SaaS (Software-as-a-Service). The catalog must provide appropriate filters so use-case teams can find the right tools/agents effectively. The catalog must contain information around the intended usage of the tool/agent, their limitations, consumption choices (SaaS, PaaS), endpoint details (local or remote), lifecycle details (including End-Of-Life), version management, owner information, active usage etc. This catalog should also act as a lineage tracker for these components. An agentic application has many moving parts like agentic framework, FM, prompts, dependencies, tools, MCP servers, collaborator agents, vector stores and many more. All this information acn to be tracked in this catalog while registering an agentic application. The platform team must provide a specification format, following their organization best practices, and any component added to the catalog must provide the required information. Each version of the component must be immutable in nature and follow Semantic versioning 2.0.0 strategy. Each component must follow the Single Responsibility Principle (SRP) with Domain-Driven-Design (DDD) inspired philosophy. For an agentic application to connect to tools, the recommendation is to use MCP. Whereas for an agent to connect to other agent, the recommendation is to use Google's Agent to Agent (A2A) protocol as its an open source agent communication standard accepted widely today.
The catalog works on a collaborator operating model where anyone within the organization can write a component and contribute to the catalog. There could also be cases where a third-party GenAI component needs to be used by a use-case team. In such case, the component can be added to the catalog by the use-case team but the ownership of keeping it up to date lies with platform team. There could also be cases where the platform team provides some common components, which can be used by any use-case team.
AWS DynamoDB or similar No-SQL database can be used to build such a catalog. AWS also provides AI Agent Marketplace where a use-case team can browse different agents and tools available.

***NOTE:** Some enterprises are trying to use MCP as an integration protocol for both agent to tool and agent to agent collaboration. This approach has few limitations like MCP being stateless while A2A being stateful. When two heterogeneous agents collaborate with each other, they need to transfer state from one to another and maintain that consistently across agents. Examples include task delegation, handoffs and coordination. In such scenarios, using MCP would not be optimal hence A2A is recommended for agent-to-agent communication.*

- **Component runtime**: Once a use-case team choose a component, they must decide on how to consume it. Based on the component, the consumption model may vary:

  - **MCP**: As a default, all MCP servers must be deployed centrally by the platform team. Note that the MCP server lives in the platform, but the actual tool/resource exposed by MCP might be present locally or remotely. The platform team must provide a way to securely propagate the trusted identity from the agentic application to the MCP server and to the API/resource.
    In a different scenario, the use-case team might want to deploy the MCP server in their own environment. In such case, the MCP server owner should provide the IaaC blueprint to run it anywhere. In cases, where the server owner does not provide this detail, the use-case team must build IaaC to deploy the server in their own environment and submit a PR to the server owner for merging the IaaC instructions and code to the component catalog.
  - **Tool**: The recommended approach to consume any tool should be via MCP, but there could be cases like latency sensitive and high throughput use-case, edge deployments, bandwidth constrained environments, air-gapped environments where direct access to the tool is necessary to meet the KPIs. For such cases, the use-case teams consume the tool directly in their environments. The tools owner may choose to provide PaaS and SaaS options and depending on the choice. The mechanism for connecting to the tool depends on the tool's specification.
  - **Agent**: Agents can be deployed as PaaS or SaaS. For SaaS agents, the use-case teams must be able to subscribe for the agent via the component catalog itself. In case of PaaS deployment, the recommended approach is to deploy it in the use-case team environment. If an agent owner provides PaaS, then they must provide the deployment instructions and blueprints. Irrespective of the agent type, the platform team should not be responsible for providing runtime for agents. Though it must be noted that in case of

local deployment, they are responsible for pulling any upstream changes which might be pushed by original agent creator.

For authentication and authorization between agents and tools, OAuth 2.0 or similar mechanisms can be used. For local deployment, the recommendation is to use services like AWS Fargate for ECS/EKS or AWS Lambda. The recommendation is to use a central AuthN gateway deployed in the security account. This gateway is managed by the platform team. Amazon Bedrock AgentCore Gateway* can be used for this purpose.

For setting up the runtime, Amazon Bedrock AgentCore Runtime* can be used which provides a serverless scalable runtime for components. Alternatively, Amazon Elastic Container Service (ECS), Amazon Elastic Kubernetes Service (EKS) or AWS Lambda can also be leveraged. Irrespective of the option chosen, each component must be able to scale independently of each other.

- **Experiment tracking server**: This component provides a central experiment tracking server where all the experiments for all the use-case teams within LoB are stored and tracked. While working on agentic applications, there are many tasks like model evaluations, prompt engineering, agent evaluations, testing the application with different versions of tools/agents and MCP servers which are iterative in nature. A central experiment tracking server can serve as a central logging server where all experiments can be viewed and compared. Every use-case team will have their own namespace/workspace created on the server and their access must only be restricted to only that workspace. The platform team is responsible for maintain the experiment server.
- **CI/CD**: Once developed, every agentic application should be packaged through continuous integration pipelines. Docker can be used to package the application. Any Continuous Integration/Continuous Deployment (CI/CD) tool of the organization's choice can be used here. The pipeline must provide steps to support different types of testing disciplines like unit testing, integration testing, system testing, security and vulnerability, stress testing, agent evaluation etc. it should provide a runtime for running any pipeline where the use-case team has authority to customize the steps of the pipeline. It is possible that the agentic application is acting as backend for a standard user interface led application (Eg: Chatbot). In such case, appropriate mechanisms need to be in place to build the software application separately, and the application invokes an already built and deployed agent (much like how software applications invoke traditional machine learning model endpoints). Amazon ECR can be used a docker registry where the deployable artifacts get registered.

- **Human-in-the-loop (HITL) workflow**: Some of the agentic applications require HITL review at different stages of execution. In such cases, the platform must offer a HITL workflow. HITL workflow orchestration on AWS can be triggered serverless by various teams where each workflow can be run individually in use-case environments. HITL should not be hosted centrally as every use-case will have their sensitive business data processed in HITL workflows and all domain SMEs would not have access to a common AWS account. The platform team must provide blueprints (IaaC) on how to setup HITL workflows. AWS provides HITL workflows as part of Amazon Bedrock and Amazon SageMaker which can be triggered serverless by the use-case teams. This github repository presents samples for how the workflow user interfaces can be built. In some cases, the organization might deploy an HITL tool of their choice. In such case, the platform team carries ownership of managing the tool.

Note that this is a representational view of the responsibilities of the platform and an actual platform account may have additional components depending on the complexity of the applications and tooling in the organization. The rule of thumb is to put any technically shared component in this account.

2/Logging account: This account is responsible for capturing all logging, observability, auditability and tracing for all the agentic applications. Let us understand its components:

- **Observability**: All agentic application must log every action and state change. Every execution must have a trace id, and the entire execution chain across agents (in a multi-agent scenario) and tools/MCP server must be traceable with the same trace id. It must log the reasoning, thinking and action steps of the agent for each execution, traceable through the same trace id. Apart from application-level logging, it must all also log technical/infrastructure related metrics like invocation latency of the FM, overall latency of the agentic application, token consumption per execution, throughput etc. All this log should be funneled into a central observability account accessible in read-only mode by the use-case teams with elevated privileges assigned to platform team. Different agentic frameworks provide varying level of logging in various formats; hence we propose to follow Semantic conventions for generative AI systems from OpenTelemetry which is an open-source universal format to collect logs and traces from generative AI applications. AWS provides native support for Agent observability through Amazon Bedrock AgentCore Observability* which supports Open telemetry with integrations available with Amazon CloudWatch as well.
- **Auditability**: Every interaction of the end user with the application must be auditable. Auditability is different from logging and observability as we focus on the network calls/event which have been executed by a user. This is to track which user did what actions with what kind of payload. Again, this information needs to be funneled into the central logging account. The use-case teams should have read-only access to the audit logs and platform team must have elevated privileges. AWS CloudTrail provides inherent auditability capabilities.

Note that this is a representational view of the responsibilities of the logging account and an actual logging account may have some more components depending on the complexity of the logging requirements in the organization. The rule of thumb is to put any logging and observability component in this account. Let us discuss the security account responsibilities now.

3/Security account: This account represents all shared security components by all the use-case teams. Note that, we are proposing functional/application security components and not infrastructural security components as those might sit in a landing zone setup. Here the focus is on providing common security components so use-case teams can have secure communication between FM, tools, MCP servers and agents in a secure way. Let us understand the components of security account:

- **Agent AuthN Gateway**: All the authentication and authorization for components in an agentic application must use this central Agent gateway. The authN server would be needed when an agentic application authenticates itself with an MCP server/tool or another agent. Note, we are not recommending all IAM constructs to be created in this account; we are emphasizing application authentication and authorization mechanisms. This gateway can be run on Amazon Bedrock AgentCore Identity* in a serverless fashion. For hosted deployments, a bespoke implementation of this can be hosted on ECS/EKS. The server might need a persistent store metadata. For such purpose, DynamoDB can be leveraged.

- **Anomaly detection**: It is recommended to use anomaly detection and detective controls on the environment which can identify and usual activity and raise alarms. Amazon GuardDuty can be used to detect anomalies in the APIs, AWS services calls and anomalies in traffic pattern through Amazon VPC logs.

4/Data lake account: This account represents a centralized (per LoB) data lake account which holds all the data for the LoB. It should be managed by the platform team but depending on the operating model, it might not be the same platform team which is managing all other platform components. Enterprise organizations tend to have a data and AI platform teams as separated functions often. The recommendation is to keep it separate to improve agility of the AI platform. It covers the following components:
- **Data stores**: This represents any internal proprietary data sources of the organization. It can also represent a mirror to information repositories like Sharepoint, Salesforce etc.
- **Data catalog**: This component provides metadata for the data accessible through the data lake account. It forms a key component for seamless data access with in the AgentOps platform.
- **Data governance**: This component represents the governance related requirements like data validation, lineage tracking, bias detection and mitigation strategies, data masking and redaction implementations etc.

- **Permission management**: This component represents the implementation of the permission model in place to allow access to the data from the data lake.
- **Streaming/Ingestion pipelines**: This component represents the pipelines implemented for ingesting the data into the data lake. It might talk to data catalog and governance components to execute end to end.
- **Namespaces**: This component enables the multi-tenancy of the platform by providing each team access to the shared data without impacting other tenant's usage patterns.

Note that this is a representational view of the responsibilities of the security account and an actual security account may have some more components depending on the complexity of the logging requirements in the organization. The rule of thumb is to put any security related component in this account. Let us discuss the use-case teams' responsibilities now.

Use-case team accounts

These accounts represent the environments owned by the use-case teams. The use-case teams continue to play an active role in AgentOps platform as they rely on the platform team for infrastructure blueprints but are responsible for keeping the environments compliant to the platform team's specification. Use-case teams have access to development, staging and production accounts.

2/Dev account: This is environment where the application development and prototyping happen. Platform team must provide this environment with all the configurations around networking, security and integrations with tools like git, CI/CD, Amazon EventBridge, Credential management, Identity based access controls etc. This account consists of the following components:

- **IDE**: Developers require an IDE to build the application. The recommendation is to provide Agentic IDEs with coding agents integrated so development tasks can be accelerated. Amazon Kiro is an agentic IDE with built in integration with latest Claude models and Amazon Q which can be used for this purpose.

- **Model evaluation pipelines**: The developers must have access to multiple FMs through the gateway so they can experiment and evaluate different models for the use-case. For evaluating the models, they need a model evaluation pipeline where they can configure the metrics for evaluation, based on the success KPIs of the use-case. The IaaC for the pipeline must be provided by the platform team but it will be run in the use-case team accounts. Bedrock Evaluations can be used for this purpose.

- **Data processing pipelines**: The use-case team might also need to run data ingestion and processing for populating the vector store and any other type of data needed for the agentic application. These pipelines should run in Dev account. The IaaC for these pipelines is provided by the platform team but the use-case team has the permissions to configure and customize it to a certain degree. The degree of customization would vary based on the operating model of the organization.

- **Data store connectors**: This component represent connectors to unstructured data stores like Sharepoint, Confluence etc. These connectors can be implemented serverless through Amazon Bedrock Knowledge base, but the use-case team can choose any connector of their choice as well. This connector should connect to the Dev endpoint of the proprietary data source.

- **Vector stores**: This component represents the vectorized store of the unstructured data for any information retrieval tasks need in the agentic applications. Use-case team can either create vector stores through Amazon Bedrock knowledge Base or can create any vector store of their choice. The Platform team provides IaaC to create these vector stores, but use-case teams must own the deployment and maintenance of the vector stores. This is primarily done to keep the data within the team, as it is hard to move data across teams in large enterprises. Vector stores must have access to production like data so the grounded-ness and relevance of answers can be verified correctly. Appropriate masking and redaction techniques must be followed through

data processing pipelines before the data is ingested in the vector stores. Following is a reference to setup the vector store:
  - Create the vector store following the blueprint and guidelines from the platform team.
  - Register it in the GenAI component catalog
  - Write an MCP server for it and run it locally for testing. Once tested, use the CI/CD pipeline provided for running the MCP server through the platform account.
  - Access the vector store through MCP.

- **Prompt store**: Prompt store provides a central repository of all the system prompts and prompt templates used in a use-case. As the model landscape is changing quickly, use-case teams may need to move from one model provider to another, which will require changes in the prompt. Similarly, upgrading the version of a model from the same provider may also require changing prompts. A use-case team might work on multiple use-cases and all the prompts from all the use-cases need to be stored, versioned and tracked at a central place. Prompt store can serve all these requirements. Amazon Bedrock provides its own prompt store, but the platform team may provide blueprints (Iaac) for any other prompt store of choice. Non-relational databases from AWS like DynamoDB can also be used as a prompt store.

- **Database**: Agentic application might also require access to business data through relational and non-relational databases. Access to these databases should be via MCP to keep the connection interfaces uniform throughout the application. In some cases where the use-case teams have direct access to the database, it may be easier and faster for them to connect to the database directly and skip the MCP step. In such instances, the use-case team must get exception approval from platform team. For setting up databases on AWS, there are plenty of relational, non-relational, graph and time-series options available which can be used. Note that the data available in this environment might be a copy of masked production data as production data is often not accessible directly in non-prod accounts.

- **Memory**: Agentic applications need short term and long-term memory to function effectively. Hence, the recommendation is to have a memory store hosted in the use-case account. The platform team can provide blueprints (IaaC) to setup the memory store, but the ownership of the memory store lies with the use-case team. The memory store can also be referred to, as the session/state store. Amazon Bedrock AgentCore Memory* can be used for this purpose. Apart from it, any serverless database or cache store on AWS like Amazon Elasticache can be used as a memory store. Teams can also use mem0 by deploying it on ECS/EKS or be used as SaaS offering. In case of multi-agent systems, this memory component will hold the shared memory across all agents working together.

- **Model evaluation store**: For evaluating the models, the developers need to build model evaluation datasets. The datasets should be specific to the use-case and the evaluation metrics. The dataset must be a diverse representation of positive, negative and edge scenarios. The platform team provides guidance and IaaC on how to setup the model evaluation store where the dataset along with its results can be stored. The dataset can be stored on Amazon S3 and referred to in the model evaluation store as well. All such evaluations must be tracked via central experiment tracker provided by platform team. The use-case team uses this experiment server to compare the metrics of different evaluation jobs.

When it comes to choosing the agentic framework, the platform team can provide recommendations on which framework to choose, but this choice should be given to the use-case. Once the agentic code is built, it should be pushed to a Git repository which triggers the CI process which tests and packages the code and make deployable artifact for it. The artifact must get registered in the GenAI component catalog. For storing the actual binary of the artifact, any existing artifact repository (Nexus, Docker or similar) can be used. AWS provides tools to setup Amazon Elastic Container Registry (ECR) and fully managed Nexus repository through AWS marketplace.

Based on the use-case team's needs and operating model of the organization, it may have more components.

3/Staging/Test environment: Once the agentic application has been built and packaged, the test environment is where the use-case team would perform all kinds of tests. All the testing should be performed through automated tools with minimal human oversight needed. Once all the tests have passed, the application's production version is registered in the GenAI component catalog and ready to be deployed to production. Some of the common components of staging environment are:

- **Guardrails:** This component represents the guardrails setup for the use-case team. The team might need to run multiple cycles of guardrail configurations to find the right one for them. They can use the central experiment tracking server to log their guardrail configurations along with the outcomes. To test the guardrails, they must prepare a guardrail evaluation dataset. It is like model evaluation dataset but more tailored on security rather than functional tests. Platform team should provide CI for testing guardrails through a central pipeline.

- **Integration testing:** These tests are run to test the integration of the agentic application with the tools, MCP server and any other APIs. The focus is on testing if the integration between the components works well or not.

- **Regression testing:** These tests ensure that a newer version of the application does not introduce any unintended bugs in the existing functionality. This would mostly be like a System test.

- **Agent Evaluation:** One of the most critical components of Staging environment is Agent evaluation. The use-case team must prepare an end-to-end evaluation dataset and test the entire agentic application. Like System test, the goal should be to validate the input and output boundaries of the application. It contains sample datasets which can be used to test the accuracy of the entire agentic application. This must be owned by the use-case teams and platform team and provide guidance through IaaC on how to set up agent evaluation store. The results of the agent evaluation will also be stored in this store for review and tracking purposes. Agent evaluation pipelines should generally be run in the testing/staging account. AWS provides an open-source toolkit or LLM-as-a-Judge can be used to do this and the platform team may choose a different framework of choice for this. Note that platform team will only provide the blueprint on how to run it, it is the use-case team's responsibility to run and store the results in the evaluation store.

- **Human-In-The-Loop (HITL):** For use-cases involving HITL, HITL workflows can also be triggered from the test environment. Agent evaluation metrics must be reviewed by humans to validate the accuracy of the agent through triggering HITL workflows.

Rest of components like Data store connectors, Vector store, Prompt store, Databases and memory/session store have already been covered in the Dev environment section, hence will not be repeated here. Note that, these components will have a dedicated server (Paas) or endpoint (SaaS) for staging environment only.

4/Production environment: This is where the agentic application gets deployed for production usage. The use-case must own this environment, and the platform must provide IaaC for setting up the environment and running the application. Some of the components of the production environment are shown below:

- **Agentic application(runtime)**: This is the runtime component where the agentic application runs in production. The platform team provides the IaaC blueprints for running the application. Note that, the actual user application may be running elsewhere. For example: the chat bot need not be deployed in the same environment and may be owned by a different team. The team owning the chatbot would invoke the agent from this prod account. In case of multi-agent systems, the entire application must be registered with a single version in the Agent registry, which must have reference to other collaborator agent versions, which in turn refer to MCP servers, tools versions along with their Git tags. For deploying the application. For deployment, Amazon Bedrock AgentCore Runtime* can be used a serverless environment. Other options include ECS/EKS/Lambda.

- **Agent monitoring:** This component is responsible for monitoring any drift in the performance of the agentic application. The platform team must provide the blueprint and IaaC for it. Agent monitoring logs should go to the logging account where the use-case team has read-only access. Use-case teams must also be able to setup and change agent monitoring rules independently and platform team must provide connectors to pass on those signals back to the data lake account for preparing a newer version of the agent. The monitoring must be running in a separate runtime (container) so that it does not impact the bandwidth of the application. The monitoring behavior can be batch analysis, analyzing the inputs/outputs at every hour or higher. In case the use-case demands a higher frequency; the behavior can be configured near real-time.
- **3P SaaS agents:** The agentic application might need to talk to any third-party (3P) agentic application. The 3P application can be a SaaS or PaaS application. In case of PaaS, it must be deployed in the same production environment as the agentic application. In case of SaaS, it would depend on if the 3P application provides an MCP or A2A interface to connect. If neither are provided, the use-case team would have built an MCP server for it during the development phase and can use a production endpoint for the same.

- **Deployment guardrails:** Deployment guardrails like A/B testing, Blue-Green and Canary deployments must be supported through production deployment pipeline so the use-case team can easily migrate to newer versions of the application and can handle fallbacks gracefully. The application guardrail configuration would be pointing to a prod version of the guardrail, and it must be recreated in the prod account through automated process (no human intervention). Any upgrades to the agentic application would go through the traditional dev->test->prod lifecycle using all the components described earlier in the document.

Rest of components like Data store connectors, Vector store, Prompt store, Databases and memory/session store have already been covered in the Dev environment section, hence will not be repeated here. Note that, these components will have a dedicated server (PaaS) or endpoint (SaaS) for production environment only.

Note that these environments are a representational view of the responsibilities of the use-case team and a use-case team in a real environment may have more responsibilities across these environments. The readers are advised to take this as general guidance and adapt it based on the needs of the use-case team.

General considerations

- **Cost**: Every application should use the cost tracking tools provided by the platform team. Each use-case team resources must be tagged with AWS tags, and each use-case team must be able to filter their costs through these tags via AWS cost explorer. To track costs across multiple AWS accounts, platform team must use AWS Organizations with Consolidated Billing, which aggregates costs from all member (use-case) accounts into a single bill. Platform team can then use AWS Cost Explorer in the management account to view costs for all accounts, grouped by linked account. Note that FM gateway costs would only be tracked here for Bedrock FMs through tags. For other FMs, the platform team need to provide a way to propagate the third party LLM costs through AWS Cost Allocation Tags feature. Each use-case team owner can be provided read-only access to this cost explorer so they can view the billings for their account. There is also a possibility of the creating dashboards per use-case team and provide use-case teams access only to their cost dashboard.

- **Governance**: All the components specified in the shared services, logging and security account contribute towards providing governance around the AgentOps platform. Additionally, components like Agent Evaluation, Guardrails, Agent Monitoring also add to compliance requirements around governance.

Topics not covered in this document

- **ModelOps/FMOps/LLMOps**: We have excluded the process of fine tuning the models in this document. But there can be cases where a use-case team finetunes a model, later to be used in an agentic application. For such scenario, we need to have the traditional Foundation Model Operationalization (FMOps) lifecycle implemented. The fine-tuned models must register in the model registry.
- **Detailed deployment topology**: Details around how to design workspaces, repository names, tooling and MCP registry conventions are not covered in this document. Each organization has their own set of standards around how to build this topology; hence the platform and use-case teams together can design the topology which works for them.
- **Infrastructure security:** Infrastructure security has not been included as we expect the central IT and infosec team will provide that for all software applications in the organization.
- **Comparison of various agentic frameworks:** There are many popular agentic frameworks out there which can be used to build agentic application. This document avoids providing any opinions on transient comparisons. As all agentic frameworks are moving at a rapid pace and the comparison will get outdated quickly as newer features get launched across these frameworks. For that reason, the focus of the document is on the things which will not change, irrespective of the agentic framework choice. AgentOps, as a practice, must be decoupled from the framework chosen for building the agent.

Common anti Patterns

1. **Platform team acting as data producer for use-cases:** The platform team managing use-case specific data will add additional responsibility on the platform team who already has a lot of their side. It will be very hard to manage, govern and scale use-case specific data in the platform. Hence, the data remains in the tenant environment.
2. **Re-inventing the wheel in FM Gateways:** The FM gateway should be as thin as possible in terms of custom logic. Managing multiple FMs which have their own authentication standards, context length, features (deep thinking etc.) and throttling, is a lot of work already. The platform team should minimize bespoke logics as much as possible and use tools which provide features like rate limiting, throttling, path-based routing etc. out of the box. Also, if the organization does not need an FM gateway, they need not have one. Accessing an LLM directly with the right governance configuration may work fine for small and lean teams.
3. **AgentOps platform is replicated across multiple cloud providers**: Replicating a GenAI platform across providers is very hard to manage and attempts to provide parity across all providers will be difficult due to feature variability. Hence, the platform should exist on one provider only. There may be reasons to replicate on multiple providers because of data residency concerns and, If this is the case, the platform team must evaluate if it's more expensive moving data or building and maintaining redundant platforms.
4. **Trying to define one common guardrails for all use-case teams:** No two tenants will have the same requirements when it comes to guardrails. The platform teams should avoid creating common guardrails. Tenants needs to have access to an API to create guardrails for their use-case. The platform team should enforce no FM calls are allowed without supplying guardrails config. If the platform team wants to set up some baseline common guardrails, they can do so by attaching a system prompt to every invocation to the FM gateway.
5. **Trying to define one agentic framework for all use-case teams:** As emphasized earlier, there is no single best agentic framework which will work well for all teams and all use-cases. With constructs like MCP, A2A, Docker, HTTPs – the agentic implementation is getting abstracted. Hence, the platform team must not enforce a single framework for everyone. The platform team can still provide guidance in terms of pros and cons, but they must keep it up to date as these frameworks get updated frequently (sometimes on a weekly basis).

**Capability in preview, not recommended for production use. For platforms which cannot wait for it to become Generally available, alternatives shall be considered.*