BEDROCK PART  
  
Excellent. Let's finalize the blueprint. This is the detailed, strategic plan for leveraging AWS Bedrock for the "Hackathon Hunter" project, with a specific focus on intelligent context management to ensure our agent is both powerful and efficient.

This plan is built around the three distinct cognitive functions you outlined.

**The Overall Architecture: A Multi-Brain System**

Our agent will operate not as a single monolith but as a system with three specialized "brains," each powered by a specific Bedrock capability. This design is robust, scalable, and strategically impressive.

**Brain 1: The "Scout" Agent (Conversational & Discovery Core)**

This is the main, user-facing agent. Its primary responsibility is discovery, analysis, and understanding user intent. It houses our "wow" factor: autonomous tool generation.

**Bedrock Components Used:** Claude 3 Sonnet (for reasoning/code-generation), Amazon Titan Embeddings (for personalization).

**Detailed Workflow & Context Management Plan:**

1. **Grounding (Initial Query):** The agent's first step is always to query the **Bedrock Knowledge Base** for a trusted list of starting URLs.
   * **Context Strategy:** This is our first major win against context limits. Instead of hardcoding a long list of URLs into a prompt, we make a small, efficient API call to a managed RAG service.
2. **The "API Archaeologist" Phase (Discovering APIs):**
   * The agent's Python logic fetches the full HTML/JS source of a target URL.
   * **Context Strategy:** To avoid overwhelming the model, we perform a **pre-processing step in our Python code**. We'll strip out non-essential, high-token elements like inline CSS, SVG data, and long base64 strings. This creates a cleaner, more focused context.
   * The agent then makes a targeted boto3 call to **Claude 3 Sonnet** with this sanitized source code. The prompt is laser-focused on one task: finding an API endpoint and returning a machine-readable JSON object.
3. **The Strategic Decision Point:**
   * The agent's Python logic parses the response from Bedrock. Based on {"api\_found": true/false}, it decides which "persona" to adopt next.
4. **The "Code-Generating Engineer" Phase (Building the Tool):**
   * This is the second, most critical Bedrock call.
   * **Context Strategy:** This is where we are most intelligent.
     + **If an API was found (4A):** The context sent to Bedrock is **small**. It only includes the API details from the previous step and the prompt to write a requests function. We do *not* need to send the full HTML again.
     + **If no API was found (4B):** The context is the original HTML. To manage this, the agent's Python logic will perform a "summarization pass." It identifies the main <body> or <main> container of the document and sends only that relevant section to **Claude 3 Sonnet**, drastically reducing the token count while preserving the necessary structure for scraping.
   * The output from this step is a complete, executable Python function.
5. **Personalization (Storing Preferences):**
   * When a user states a preference, the agent sends that short text snippet to **Amazon Titan Embeddings**.
   * **Context Strategy:** This is inherently efficient. The embedding model is designed for short-to-medium length text. There are no context window concerns here.

**Why this design is strategic:** It mimics an expert developer's workflow. It tries the most efficient path first (API discovery), falls back to a more complex solution (scraping), and optimizes the context it uses at every single step. It's not just smart; it's efficient.

**Brain 2: The "Nudge" Agent (Scheduler & Reminder Core)**

This is a separate, lean, and reliable agent that runs on a schedule (e.g., via AWS Lambda + EventBridge). Its only job is to send proactive, valuable notifications.

**Bedrock Component Used:** Claude 3 Haiku.

**Detailed Workflow & Context Management Plan:**

1. **Deterministic Logic First:** The agent's Python code performs all the heavy lifting *before* calling Bedrock.
   * It queries the MongoDB to get the list of newly discovered hackathons.
   * It queries the Vector DB to get the user's specific interests.
   * It performs all the crucial logic: filtering hackathons based on interest, deduplicating against past notifications, and deciding whether to send a general update or a specific "interested list" reminder.
2. **The Bedrock Call (The Final Mile):**
   * **Context Strategy:** The agent only sends the **final, curated, and deduplicated list** of hackathons to Bedrock. The context is minimal and perfectly formed for the task. We are not asking the LLM to do any filtering or database logic.
   * The prompt to **Claude 3 Haiku** is simple: *"Craft a friendly and concise Telegram notification summarizing these new hackathons for a user."* We choose Haiku because it's the fastest and most cost-effective model, perfect for a high-frequency, low-complexity task like this.

**Why this design is strategic:** It demonstrates architectural maturity. We are using a cheap, fast model for a routine task and reserving our powerful, expensive model for the complex reasoning in the Scout agent. We're using Python for what it's good at (data manipulation) and the LLM for what *it's* good at (natural language generation).

**Brain 3: The Knowledge Base (The Grounding Core)**

This is our agent's source of truth.

**Bedrock Component Used:** Amazon Bedrock Knowledge Bases.

**Detailed Workflow & Context Management Plan:**

* **Function:** This is a managed RAG service that we pre-load with a simple document listing trusted hackathon websites.
* **Context Strategy:** The Knowledge Base is the *solution* to a context problem. The entire process of document chunking, embedding, and retrieval is handled by the managed service. Our agent simply sends a short query string (e.g., "trusted hackathon sources") and gets a clean, relevant answer. We never have to manage the large document in our agent's direct context.

**Why this design is strategic:** It shows we know how to build a responsible and reliable agent. By grounding our agent's discovery process, we make its behavior more predictable and its results more valuable, which is a key concern for any real-world AI application.

AGENTS AND DATABASE

Of course. This is the critical phase where we translate our winning idea into a concrete, actionable plan. We'll design a robust, scalable, and modern architecture that will not only function flawlessly but will also deeply impress the hackathon judges.

Here is the detailed plan for the infrastructure, databases, and the implementation of Strands for the "Hackathon Hunter" project.

**Project Timeline Overview (Given today is Thursday, Oct 16th):**

* **Phase 1 (This Weekend):** Foundation - IAM & Database Setup.
* **Phase 2 (Early Next Week):** The "Scout" Agent - Core Logic & Tool Development.
* **Phase 3 (Mid-Late Next Week):** The "Nudge" Agent & User Interface.
* **Phase 4 (Final Weekend):** Integration, Testing, and Submission Prep.

**Phase 1: Foundation - IAM & The Dual-Database Memory**

**(Goal: By Sunday night, our entire data backend is provisioned, secured, and ready for data.)**

**1. Identity and Access Management (IAM) - The First Step**

Before writing a single line of code, we create a dedicated IAM Role. This demonstrates a security-first mindset.

* **Action:** Create a single IAM Role named HackathonAgentExecutionRole.
* **Permissions (Policies):** Attach policies granting this role necessary permissions for:
  + **Amazon Bedrock:** Full invocation access.
  + **Amazon DynamoDB:** Full read/write access to our specific tables.
  + **Amazon OpenSearch Serverless:** Full read/write access to our specific collection.
  + **AWS Lambda & ECS:** Permissions to write logs to CloudWatch.

**2. The Databases - Our Agent's Memory**

We will provision our two managed databases.

* **Database 1: Amazon OpenSearch Serverless (for Semantic Memory)**
  + **Action:** Provision a new **Serverless Collection**.
  + **Configuration:** Enable the **Vector Engine**.
  + **Index Name:** user\_preferences
  + **Schema (Fields within the index):**
    - user\_id (Text/Keyword)
    - preference\_text (Text)
    - preference\_vector (Vector, Dimension: **1536** for Amazon Titan Embeddings)
* **Database 2: Amazon DynamoDB (for Operational Memory)**
  + **Action:** Create three separate DynamoDB tables.
  + **Table 1: Hackathons**
    - **Partition Key:** hackathon\_id (String)
    - **Attributes:** source\_url, title, deadline, prize\_pool, raw\_data\_blob, discovered\_timestamp.
  + **Table 2: ScraperFunctions**
    - **Partition Key:** source\_url (String)
    - **Attributes:** scraper\_code (String - this will hold our generated Python code), function\_type ('api' or 'scraper'), last\_updated\_timestamp.
  + **Table 3: UserInterests**
    - **Partition Key:** user\_id (String)
    - **Sort Key:** hackathon\_id (String)
    - **Purpose:** A simple, highly efficient mapping table.

**Phase 2: The "Scout" Agent - The Intelligent Discoverer**

**(Goal: By Wednesday, the Scout agent can analyze a new URL, generate a tool, and populate our databases.)**

**1. Infrastructure: AWS ECS with Fargate**

* **Action:**
  1. Write a Dockerfile for our main Python application (which will include Strands, Boto3, etc.).
  2. Define an **ECS Task Definition**. We'll specify the Docker image, assign our HackathonAgentExecutionRole, and allocate resources (e.g., 1 vCPU, 2GB RAM).
  3. Create an **ECS Service** running on the **Fargate** launch type. This will ensure our container is always running and managed by AWS.

**2. Strands SDK Implementation**

* **Core Logic:** In our main application file, we will initialize our Strands agent.
* **The Brain:** We will configure it to use **Claude 3 Sonnet** from Bedrock as its reasoning model.
* **The Tools:** We will build our core logic as Python functions and decorate them with the Strands @tool decorator.
  + discover\_extraction\_method(url): This is the master tool. It contains the logic for our "Archaeologist" workflow:
    1. Fetches HTML/JS from the URL.
    2. Sanitizes the code (removes CSS, etc.).
    3. Calls Bedrock to find a hidden API.
    4. Based on the result, it decides the function\_type ('api' or 'scraper').
    5. Returns a dictionary: {'type': function\_type, 'context': api\_details\_or\_html}.
  + generate\_python\_function(type, context):
    1. Receives the output from the discovery tool.
    2. Makes the second Bedrock call with the appropriate "Code-Generating Engineer" prompt.
    3. Returns the generated Python code as a string.
  + execute\_and\_store\_data(function\_code\_string, url):
    1. Safely executes the generated code.
    2. Takes the returned hackathon data and writes it to the Hackathons DynamoDB table.
    3. Saves the function\_code\_string to the ScraperFunctions DynamoDB table, linking it to the URL.
* **Orchestration:** The main Strands agent loop will be designed to chain these tools together in the logical order: discover -> generate -> execute\_and\_store.

**Phase 3: The "Nudge" Agent & User Interface**

**(Goal: By Friday, a user can talk to our agent via Telegram and receive scheduled updates.)**

**1. The "Nudge" Agent (A Second, Specialized Strands Agent)**

* **Infrastructure:** **AWS Lambda**.
  1. We will package this agent's code into a Lambda deployment package (a .zip file with its dependencies).
  2. Create a Lambda function, assign our HackathonAgentExecutionRole, and set a reasonable timeout (e.g., 60 seconds).
* **Trigger:** **Amazon EventBridge Scheduler**.
  1. We will create a schedule that triggers our Lambda function. For example, a cron expression for every Sunday at 8 PM (0 20 ? \* SUN \*).

**2. The User Interface: API Gateway + Telegram**

* **Action:** Create a **REST API in Amazon API Gateway**.
* **Integration:** The API Gateway endpoint (e.g., /chat) will be configured with a **Lambda Proxy Integration**. This means any request to the API will trigger a "handler" Lambda function.
* **Handler Lambda:** This function will receive the message from Telegram, process it, and then **asynchronously invoke our ECS Fargate task** for the "Scout" agent to perform the heavy lifting. It will then immediately return a message like "On it! I'll start looking for hackathons for you." This non-blocking architecture is crucial for a good user experience.
* **Telegram Bot:** We will register a bot with Telegram and configure its webhook to point to our new API Gateway URL.

This detailed plan provides a clear, step-by-step path to building a scalable, reliable, and incredibly impressive agentic AI system on AWS.

REMAINING PART

So yes, there are several crucial pieces remaining. Let's categorize them into three key areas: **The User Experience**, **The Presentation Package**, and **The "Production-Ready" Polish**.

**1. The User Experience (The "Front Door")**

This is about making our powerful agent accessible and delightful to use. Right now, a user interacts via a Telegram bot, but we need to define that interaction precisely.

* **Onboarding & Guidance:** What happens when a user opens the bot for the first time? We need a clear welcome message.
  + **Action Plan:** Design a /start command that introduces the agent, explains what it can do (discover, track, and notify), and provides example commands like /find hackathons on devpost or /set\_preferences.
* **Interactive Feedback & State Management:** The agent's "thinking" process (analyzing a new site, generating code) can take time. A user should never be left with a silent, unresponsive bot.
  + **Action Plan:** Implement "thinking" messages. When the agent starts a complex task, it should immediately reply:
    - *"Okay, I'm analyzing newsite.com for the first time. This might take a moment while I build a custom tool..."*
    - *"Tool created! Now fetching the latest hackathons for you..."* This makes the agent feel alive and manages user expectations.
* **Rich Results Formatting:** Simply dumping a list of hackathon titles is not enough.
  + **Action Plan:** Use Telegram's formatting capabilities. Present results in a clean, readable format using Markdown. More importantly, use **inline buttons** for each hackathon: [More Details], [Track This Hackathon]. This makes the experience interactive and allows users to easily add hackathons to their UserInterests database.

**2. The Presentation Package (The "Sales Pitch")**

This is arguably the most important part of any hackathon. A brilliant project with a poor presentation will lose to a good project with a great presentation.

* **The Video Demo (The 3-Minute Story):** This is your most critical submission asset.
  + **Action Plan:** Create a compelling 3-minute video. The script should be:
    1. **The Problem (15s):** "Finding the right hackathon is a messy, manual process."
    2. **The Solution (30s):** Introduce the "Hackathon Hunter" agent. Show the clean Telegram interface.
    3. **The "WOW" Moment (90s):** This is the core of the video. Show the agent receiving a URL for a site it has **never seen before**. Show the agent's feedback message ("*I'm analyzing this new site...*"). Then, show the final, correctly scraped results. This demonstrates the autonomous tool generation.
    4. **The Impact & Vision (45s):** Briefly show the proactive "Nudge" agent's notifications and talk about the future roadmap.
* **The README.md File (The Blueprint):** Your project's repository is your written pitch.
  + **Action Plan:** Create a detailed README.md. It must include:
    1. A clear project summary.
    2. The same problem/solution narrative as the video.
    3. A high-level **Architecture Diagram**. This is non-negotiable and shows the judges you had a clear plan.
    4. A "How it Works" section explaining the **agentic workflow** (API discovery, code generation, multi-agent system).
    5. A "Tech Stack" section explicitly listing every AWS service used and *why* it was chosen (e.g., "We chose ECS Fargate for our Scout agent to handle long-running analysis tasks...").
* **Code Quality & Comments:** The judges *will* look at your code.
  + **Action Plan:** Ensure the code is clean, well-organized, and commented. Specifically, comment the prompts you are using for Bedrock. This gives the judges direct insight into the "mind" of your AI.

**3. The "Production-Ready" Polish (The Differentiators)**

These are the details that show you're thinking beyond the demo and building a real, robust application.

* **Observability & Logging:** What happens when something goes wrong?
  + **Action Plan:** Implement structured logging using **Amazon CloudWatch**. Every major decision the agent makes should be logged. For example:
    - INFO: Found API endpoint for devpost.com. Generating API function.
    - WARN: No API found for newsite.com. Escalating to scraper generation.
    - ERROR: Generated code failed to execute.
* **Security & AI Safety:** How do we make sure our agent is responsible?
  + **Action Plan:** Implement **Guardrails for Amazon Bedrock**. We can configure a simple Guardrail to prevent the agent from scraping malicious URLs or handling inappropriate user input. Mentioning this demonstrates a commitment to building safe and responsible AI.
* **Cost Awareness:**
  + **Action Plan:** Add a "Cost & Scalability" section to your README. Briefly explain how the serverless architecture (Lambda, Fargate, DynamoDB, OpenSearch Serverless) is inherently cost-effective and can scale from one user to millions with no infrastructure changes. This shows business acumen.