ChatGPT

Graph-Based Cloud IAM in the GenAI Agentic World

Dinis Cruz and ChatGPT Deep Research

Introduction

The rapid rise of **generative AI agents** – autonomous software powered by large language models (LLMs) – is redefining how applications interact with cloud services. These AI agents can dynamically decide to invoke cloud APIs, often in unpredictable ways. This flexibility brings great power, but also new **security challenges**. Chief among these is ensuring that an AI agent only has the *minimum necessary privileges* for each action it takes – a long-standing principle known as **least privilege** [1] [2]. In practice, today's cloud Identity and Access Management (IAM) systems make this hard to achieve. Cloud applications frequently run with **over-provisioned permissions**, where a single identity (e.g. an AWS IAM role) has a broad set of permissions covering many possible actions the app *might* perform. As a result, at any given time an application or agent has far more privileges than it actually needs, opening the door to accidental or malicious misuse [2].

This white paper argues that the advent of GenAI agents makes it critical to adopt a new, **graph-based IAM and permission workflow** for cloud providers. By modeling cloud permissions, resources, and API calls as a **knowledge graph**, we can precisely determine the exact privileges required *per action* and issue ephemeral, context-specific credentials. This approach would act as a powerful **guardrail** for AI agents, containing their behavior within safe bounds even if they try to go "off-script." In the following sections, we examine the problems with current cloud IAM, explain why generative agents amplify those issues, and propose a graph-driven solution (drawing on semantic knowledge graph techniques from our prior work) to achieve fine-grained, just-in-time access control across cloud environments.

The Problem: Over-Provisioned Cloud Permissions

Modern cloud environments are extraordinarily complex – AWS, Azure, GCP and others expose thousands of services and API actions, each protected by granular IAM permissions. In theory, one can assign minimal rights for each task. In practice, however, organizations end up with **permission bloat**: *"Modern cloud estates hold thousands, often millions, of individual permissions, creating an ever-expanding attack surface"* [3]. Because it's difficult to know upfront which permissions a piece of code will need, developers and DevOps engineers often grant broad roles that encompass the union of all actions an application might perform in its lifetime. For example, a simple microservice might get an IAM role allowing full read/write access to an entire storage bucket or database, simply because it needs to read one object or write one record. Over time, such roles accumulate permissions "just in case," rather than strictly delineating what is needed. The result is **almost every request is over-privileged** – any given API call executes in a context that has more access than necessary for that call.

Over-provisioning is more than just a theoretical concern; it has real security impacts. Excessive permissions enlarge the possible **blast radius** if an account is compromised [4]. A breached service with broad IAM rights can be escalated to exploit resources that service never needed to touch. Indeed, *"over-privileged identities can inadvertently or maliciously expose sensitive data"* and enable attackers to move laterally through a cloud environment [2]. High-profile cloud breaches have repeatedly been traced to roles or keys with permissions beyond their strict requirements. Moreover, from a governance perspective, over-provisioned access makes it hard to evaluate risk – security teams lack visibility into which permissions are truly needed versus which are excess liabilities. Traditional auditing tools focus on static role definitions or on network connectivity, but **cloud security is deeply about relationships**: which identity can access which resource with what privilege [5]. Understanding those relationships requires analyzing not just isolated configs, but the connections between identities, policies, actions, and resources. In fact, cloud providers themselves acknowledge the importance of least privilege and recommend continuously trimming permissions. AWS, for example, urges customers to *"grant only the permissions required to perform a task"* and to refine broad policies over time [1]. They even provide IAM Access Analyzer to suggest narrower policies based on observing access patterns [6]. However, these tools are reactive – they rely on monitoring activity or developer trial-and-error. There is currently **no easy way to determine ahead of time exactly what permissions a given cloud API call will require**, especially when multiple services or resources are involved. This uncertainty leads to the "safe" route of overshooting on permissions to avoid runtime errors, at the cost of security. In summary, today's IAM paradigm often forces a coarse-grained, static approximation of access needs, and the gap between permissions granted and permissions actually needed remains dangerously wide.

Risks Amplified by Generative AI Agents

The advent of **agentic GenAI** – AI systems that can plan and execute sequences of actions – makes the least privilege problem not only pressing but **urgent**. In traditional software, the set of actions the code will perform is relatively fixed, defined by the developer. If an application has more privileges than it uses, there is at least a static control in the code that limits what gets executed. An AI agent, on the other hand, has a much more flexible decision space. It might receive an unexpected instruction (or even a malicious prompt injection) that causes it to attempt cloud operations outside the narrow scope its developers envisioned. If it has been given a broad role, nothing technically stops the agent from invoking any API allowed by that role. In essence, the **only thing preventing misuse was the code's intent** – and with AI, intent can be influenced or subverted in real-time.

Consider a GenAI agent designed to manage support tickets that's been granted general read/write access to a company's cloud storage for logging purposes. In normal operation, it would only append new log entries. But if the agent is prompted (by a user or by an attacker manipulating its input) to **search and delete files** in that storage, it may very well try to do so – not out of malice, but because the instruction seemed reasonable to it. If its IAM role included broad storage privileges (as is often the case), the cloud will dutifully execute those destructive calls. Under traditional safeguards, we might rely on application logic to prevent such a scenario, but with an AI agent the logic is not hard-coded; it's emergent from prompts and learned behavior. This fundamentally shifts the security model: we can no longer assume that an entity with credentials will only call the APIs it absolutely needs. We must instead **assume it might call anything it has permissions for**, and plan accordingly.

This new reality makes **strict privilege enforcement** at a granular level indispensable. Rather than trying to predict and constrain the AI's *intent* (an uphill battle as LLMs grow more complex), it is more reliable to constrain the **impact** of its actions via IAM. By drastically limiting what the agent's credentials allow at any given moment, we create a strong safety net. If the agent "goes rogue" or is tricked into a rogue action, the attempt will simply fail authorization, protecting the system. In effect, **cloud IAM becomes the last line of defense for AI behavior**. The principle of least privilege thus matters more than ever in the GenAI context – it turns a potentially free-roaming AI into a fenced-in one. Cloud providers have recognized related concerns; for instance, Microsoft's guidance on Azure OpenAI advises using isolated resource scopes for any AI that can execute actions. And security researchers note that misconfigured AI agents with cloud access could cause unintended data leaks or resource misuse if not properly sandboxed.

However, trying to manually craft ultra-fine roles for AI agents quickly runs into the complexity problem described earlier. We need a systematic way to calculate and enforce *just-enough* permissions dynamically, in sync with the agent's actions. In the next section, we outline such a solution: modeling the cloud's IAM universe as a **graph** that can be queried to get the precise permissions required for any given API call or sequence of calls. This approach will enable an AI orchestration system to automatically grant ephemeral credentials tailored to each step an agent takes, dramatically reducing the risk of an AI agent causing harm, whether by accident or by compromise.

A Graph-Based Approach to Cloud IAM

To overcome the challenges above, we propose treating cloud IAM as a **graph problem**. In essence, we build a **semantic knowledge graph** that maps out all relevant entities and relationships in the permission space: identities (users, roles, service accounts), services and APIs, specific cloud resources, and the permission primitives (actions) that link them. Graph theory is well-suited for capturing complex relationships, and has already proven valuable in cloud security contexts [5] [7]. Unlike a static list of policies, a graph can answer nuanced questions about connectivity and reachability in the permission landscape. For example, it can reveal that *Identity A can call API X which touches Resource Y*, or that *Role R has a path to access Data Z through a chain of permissions*. As one security thesis observed, knowledge graphs provide the flexibility and context needed to understand cloud interconnections, far beyond what simple checks or tree structures can do [7]. Our goal is to leverage this power to enable **per-call permission reasoning**.

In a graph-based IAM model, every cloud API call (down to the granularity of an SDK function or REST endpoint) is represented as a node connected to the specific **permissions** required to execute it. Each permission in turn links to the **resource type or object** it governs. Identities (like roles) connect to permissions they grant. The graph thus encodes the many-to-many relationships between actions and required privileges. Crucially, the graph can be made **context-aware** – for example, a node could represent "Write to S3 bucket *X*" which might require a different permission edge than "Write to bucket *Y*" if bucket policies differ. By querying this graph, one can derive the *minimal set of permissions* needed for a given operation. In other words, we take what is currently an undocumented or hard-coded mapping (the knowledge in a cloud provider's documentation of which API calls require which IAM actions) and make it an explicit, queryable part of the system.

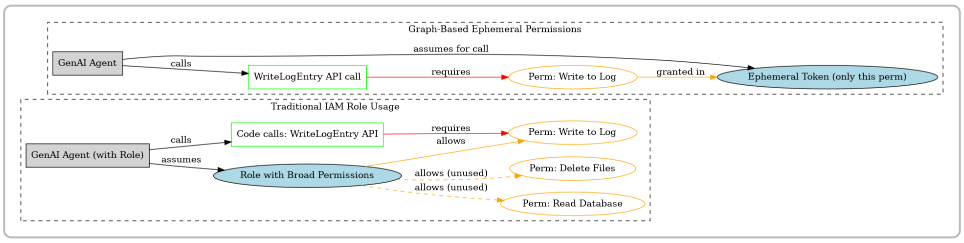

Figure: Traditional vs. graph-based permission usage. In the traditional model (bottom box), a GenAI agent assumes a single role that has the union of all permissions it might ever need, many of which (e.g. "Delete Files", "Read Database") are not required for the current action. This over-provisioning increases risk. In the graph-based approach (top box), the agent's requested API call (WriteLogEntry) is analyzed to determine exactly the privilege it requires ("Write to Log"). An ephemeral credential is then issued just *with that permission for the duration of the call, greatly limiting the agent's capabilities to the task at hand.*

Determining Required Privileges per Call

The heart of the graph approach is the ability to compute the **required privilege set** for any given request. Imagine an AI agent wants to perform an action, such as “write an entry to a log service.” Using a knowledge graph of the cloud, the system can trace from that high-level API call node to identify all IAM permissions that call demands. This might involve a single permission (e.g. `logs:WriteEntry`) on a specific resource (a log group), or it could be a combination of permissions if the operation is composite. With a well-constructed graph, this query becomes straightforward – essentially a graph traversal from the API node to connected permission nodes. The result is effectively an **authorization blueprint** for that action. This contrasts sharply with today’s trial-and-error approach where developers guess and attach policies until the call stops throwing “Access Denied.” Instead, the required permissions are known *deterministically* in advance. We shift from thinking in terms of “roles” (coarse bundles of permissions) to thinking in terms of **the intersection of privileges needed by this exact operation in this context** (much like the Java Security Manager used to enforce that every stack frame’s permissions intersect for an operation to proceed [8]).

One immediate benefit is **pre-execution validation**: given the graph, an agent or developer can ask *“If I attempt this call with Role R, do I have all the needed permissions?”* and the system can answer before anything runs. This would eliminate a huge class of runtime errors and insecure workarounds. No more deploying code and discovering via logs or CloudTrail that it needed an extra permission – the graph would reveal that upfront. It also means security teams could systematically check that roles are tight: if a role is supposed to only allow a certain function, the graph query for that function should exactly match the role’s permissions. Any extras in the role are, by definition, over-privilege that could be trimmed. This approach **flips the current paradigm**; instead of roles defining what calls are allowed, the *calls define what permissions are required*, and roles or tokens are derived from those requirements.

Ephemeral, Just-in-Time Permissions

Knowing the minimal required permissions for an action enables a powerful enforcement mechanism: **ephemeral credentials minted per action**. In our proposed workflow, an AI agent would not hold a permanently overpowered role at all. Instead, when the agent decides to perform a cloud operation, it would request a temporary credential (token) scoped to exactly the permissions the graph deems necessary for that operation – nothing more. This is akin to **Just-in-Time (JIT) access**, a concept already gaining traction for human admins and automated tasks [9] [10] . Cloud providers have the technical primitives to support this (for example, AWS’s Security Token Service can issue time-limited credentials and even restrict them with session policies). However, today JIT is usually coarse (granting a role for a limited time). Here we are talking about JIT at the *millisecond scale* – a new credential for each API call or logical task the agent performs, then immediately revoking it. The agent might assume dozens of different micro-roles in a single session, one for each distinct API it touches. Each credential’s scope is so minimal that even if the agent tries something outside its current task, it will be blocked. Essentially, we achieve **runtime least privilege**, not just design-time least privilege.

The benefits of this approach for GenAI safety are immense. The AI’s freedom to harm is constrained by construction. If it was prompted to do something nefarious like “exfiltrate all user data,” it would first have to obtain a token for that action – which it wouldn’t, because such a request wouldn’t align with any permitted high-level operation in its allowed workflow. Even if the AI somehow tried to step outside, the lack of credentials stops it. As one cloud security expert noted, *“Permanent admin rights are gold for attackers. JIT access flips the model: privileges appear only when needed… and disappear automatically”* [10] . We apply this principle in extreme granularity.

Of course, performance and practicality need consideration – requesting a new credential from the cloud for every single call could add overhead. There are options to optimize, such as bundling closely related calls that always occur together into a single short-lived role. The graph could even facilitate such optimization by identifying common permission sets across sequences of calls. But even if there is some overhead, the security payoff may warrant it, especially for high-impact operations. The approach also encourages a **behavioral shift in application design**: services might be built to handle occasional "Access Denied – need new token" signals by pausing to obtain narrower credentials, rather than running with full privileges upfront. In effect, applications (including AI agents) become more conscious of **what authority they need at each moment** and explicitly request it, following the principle of **POLA** (Principle of Least Authority) at runtime. This was conceptually demonstrated decades ago in object-capability systems and the principle of least authority in software, but cloud platforms now give us the tools to implement it widely [4] .

Dynamic Adaptation and Reduced Blast Radius

Another advantage of graph-driven ephemeral permissions is the ability for software to **adapt to the permissions it has**. In current practice, if a cloud call fails due to missing permission, the application typically crashes or errors out; it's not common for the code to adjust functionality on the fly. But an AI agent or modern service, armed with knowledge of available permissions, could degrade or change its behavior instead of failing. For instance, if the agent knows it only has read access to a dataset and not write, it could avoid any actions that would require write, or queue them until permission is granted. This adaptability was difficult to achieve before, because applications had no easy way to introspect their precise authorization scope. With a permission graph and per-call tokens, however, the agent's context includes a clear picture of "what it can and cannot do right now." We can design agents to be **permission-aware**, checking the graph before attempting something and either requesting elevation (via a controlled workflow) or handling the lack of permission gracefully. This approach turns applications more resilient and **prevents exploitation** – an attacker who somehow influences the agent can't make it do disallowed things; the agent literally doesn't have the capability at that time. In cloud security terms, we are dramatically shrinking the **blast radius** of any single compromised component or misbehaving agent [4] . A vulnerability in one API call can only be abused within the very limited scope of that call's token. This containment aligns with zero-trust philosophies and would make cloud breaches significantly harder. Leaked credentials, for example, would be of little value if they expire after one use and only grant minimal rights.

Finally, by logging each ephemeral permission grant and use, we gain extremely granular audit trails. Instead of seeing that "Role X accessed 100 different things," we would see a graph of *intents*: e.g. "Agent attempted Action A -> was granted Token with Permission P -> accessed Resource R". This is much more informative for compliance and forensic analysis. It ties every cloud access to a justified action. It would even enable on-the-fly risk assessment: if an agent suddenly requests a permission that is out of pattern or higher sensitivity, the system could flag it or require extra approval (much like an MFA challenge for unusual user actions). In sum, a graph-based IAM coupled with ephemeral enforcement not only minimizes privileges but also opens the door to **intelligent, context-aware cloud security workflows**.

Implementation with Semantic Knowledge Graphs

How would we build such a system in practice? The foundation is a comprehensive **semantic knowledge graph** of the cloud provider's services, APIs, and IAM schema. Fortunately, much of this information is available – though not yet in graph form – via documentation and cloud metadata. For example, AWS publishes a list of all IAM actions for each service and what resources and conditions they support. We would ingest this data into the graph, creating nodes for each action (e.g.

`s3:PutObject`), linking them to resource types (S3 Bucket, Object) and to higher-level API groupings. On top of this, we add relationships that map API calls (as exposed in SDKs or REST endpoints) to the IAM actions they require. Cloud providers don't always spell this out explicitly, but it can be derived from docs and experimentation. In complex cases, an API call might require multiple IAM actions (for instance, an AWS *copy* object call might need both a Get on the source and a Put on the destination). All those would be captured as relationships in the graph. We also include the identities and roles defined in a given cloud environment and their attached policies as part of the graph data. This yields a living graph of *who can do what on which resource*. In effect, it's an **authorization knowledge graph**. Security companies are already moving in this direction to visualize cloud access; for example, Zscaler's cloud protection suite builds knowledge graphs to correlate assets, vulnerabilities, and permissions [5], and academic tools like HackerGraph have demonstrated merging cloud config data into graphs for analysis [11] [12]. Our approach is unique in that it focuses on *intent and required permissions*, not just on existing configurations.

To build and maintain such graphs at scale, we can use **semantic graph databases** and even employ LLMs to assist in data curation. In our own research, we developed an open-source engine called **MGraph-DB** – a memory-first graph database optimized for GenAI and semantic web use cases [13]. MGraph-DB allows easy manipulation of JSON data as graph nodes and edges, and can integrate with LLM-powered pipelines for knowledge extraction [14]. For example, one could feed in API documentation and use an LLM to extract a structured mapping of API -> required permissions, then store that in MGraph-DB. This process was used in the **MyFeeds.ai** project to build semantic knowledge graphs from unstructured inputs [15] [14]. The graph database's role is to provide a queryable, up-to-date model that can be consulted in real time by the agent orchestration logic. Because MGraph-DB is optimized for in-memory and serverless usage, it can be embedded alongside the agent, providing millisecond-speed graph queries for each agent action. The choice of graph database is flexible – one could also use Neo4j or AWS Neptune – but the performance and GenAI integration features of MGraph-DB are a strong match for this problem space (it was explicitly designed for knowledge graph construction and querying in GenAI applications [16]).

Beyond the graph storage, we need a **policy decision engine** that interfaces with the cloud to fetch and cache credentials. This could be implemented as a sidecar service or library. When an agent wants to perform an operation, it queries the graph for required permissions, then calls the cloud's API (such as AWS STS `AssumeRole` or GCP's short-lived token APIs) to get a credential limited to those permissions. Cloud providers might need to add new APIs to make this seamless – e.g., a single API call where you specify the desired permission set and get back a token if the base identity is allowed to wield those. In fact, cloud providers could internally **host the permission graph** and do this automatically: the request comes in and the system itself deduces and limits the scope. Until such native support exists, a middleware component can handle it. Notably, existing cloud IAM features like **session policies, conditions, and permission boundaries** can be leveraged to implement the runtime restriction. We can have a general "agent role" that is very constrained (perhaps it only has permission to assume more specific roles), and then use session policies to dial down to the needed action each time [9]. This approach ensures that even if the agent tries to bypass the intended workflow, it cannot escalate its privileges beyond what the graph permits.

A key part of implementation is also **multi-cloud abstraction**. Each provider has its own IAM syntax and quirks (AWS IAM vs Azure Role-Based Access Control vs GCP IAM). We can make the knowledge graph **cloud-agnostic** by introducing a conceptual layer – for instance, abstract nodes for common intents like "read object" that link to provider-specific actions like `s3:GetObject` or `azure:StorageBlob Read`. This abstraction layer could ease portability and provide a consistent framework to reasoning about permissions across clouds. It also addresses the "security config as cloud lock-in" issue: currently, migrating from one cloud to another is hard partly because you have to redo all

IAM policies. A graph model could facilitate translation of intent. Indeed, one can envision an **open standard** for describing cloud permissions and APIs in graph form, which cloud providers could adopt to make security more transparent. While providers might have been reluctant to standardize IAM (since differences create stickiness), the growing pressure for secure AI deployment might encourage more collaboration. The first cloud to implement a robust graph-based permission system could gain a reputation as the safest place to run AI agents – a competitive advantage in the market. As one industry roadmap notes, reducing permission sprawl and implementing just-in-time, least-privilege across AWS/Azure/GCP is now seen as a business win, not just a security checkbox [3] [10].

Case Study and Feasibility

To illustrate how this works, let's walk through a simple scenario on AWS. Suppose our GenAI agent needs to perform two main tasks: (1) read a customer record from DynamoDB, and (2) write an entry to an S3 log bucket. Traditionally, we might attach a role to the agent (or the Lambda it runs on) that has `dynamodb:GetItem` on the table and `s3:PutObject` on the bucket. Often, though, we'd grant broader access like all GetItem on any table or access to all S3 buckets in a project, for convenience. In our graph-based model, when the agent is about to read from DynamoDB, it queries the graph and finds that it needs the action `dynamodb:GetItem` on that specific table. It then obtains a token from AWS with just that permission (and perhaps a very short TTL, say 1 minute). The DynamoDB read executes under that token. If the agent (or an attacker hijacking it) tried at this moment to do anything else – say delete an item or access a different table or call S3 – it would fail because the token doesn't allow it. Next, when the agent proceeds to the logging step, it releases or lets the first token expire, and requests a new token for `s3:PutObject` on the log bucket. That operation then executes with the second token. In CloudTrail (AWS's audit log), one would see the agent assuming these scoped credentials and using them; from an audit standpoint, it's clear that at time T1 the agent only had DB-read rights, and at time T2 only log-write rights. If an error occurs due to missing permission (which should not happen if our graph is accurate), it indicates our graph might be incomplete, and we can update it – over time the knowledge graph becomes more and more precise by incorporating any "Access Denied" learnings. In fact, the system could automatically feed such events back to improve the model. This synergy between **runtime observation and graph knowledge** can create a self-improving cycle. Our previous research on **self-improving knowledge graphs with GenAI** suggests that an AI can help reconcile differences between expected and actual permission requirements, continuously fine-tuning the graph's accuracy [15] [17].

Performance-wise, the assumption of roles introduces some latency, but AWS STS calls are on the order of a few hundred milliseconds at most. If an agent makes thousands of calls per second, a per-call STS might not be feasible, but as mentioned we can batch certain operations or use one token for a tight loop of identical operations. In many workflows, the overhead is acceptable given the security gain (especially since generative agents often operate at human-like speeds, not raw machine speeds, when doing higher-level tasks). Moreover, cloud providers could optimize the path for issuing these ephemeral credentials if it became a common pattern – possibly via in-process tokens or by allowing a parent role to spawn child permissions internally without a full network call. These are implementation details that can be refined with provider support.

Conclusion

As generative AI agents become first-class cloud citizens, we must reimagine identity and access management with much finer brushstrokes. The traditional approach of assigning a static role with broad permissions to an application is increasingly untenable in a world where AI-driven actions are unpredictable. We have presented a vision for a **graph-based IAM and permission workflow** that dynamically computes and grants the least privilege needed *in the moment*, acting as an effective guardrail for AI behavior. By leveraging semantic knowledge graphs, this model provides clarity and context that has long been missing in cloud security – it makes relationships and requirements explicit, allowing both humans and machines to reason about access control logically [7]. Implementing this will require effort from cloud providers and the community: building the graphs, developing supporting tooling, and possibly extending IAM APIs. Yet, the components are already coming together. Companies are using knowledge graphs to map cloud risks [5], and JIT permission frameworks are emerging for critical workloads [9]. Our own work with MGraph-DB and MyFeeds.ai demonstrates that highly performant, GenAI-integrated graph databases are available to underpin these solutions [16] [14].

The payoff is substantial. Cloud platforms that enable graph-based IAM could offer strong guarantees that running an AI agent on their infrastructure is safer – the agent can effectively be **sandboxed by privilege**. This assurance will be crucial for enterprises embracing AI automation: they can allow AI to interface with sensitive systems without handing it the "keys to the kingdom" all at once. Instead, the keys are given one-at-a-time, just as needed, and taken away immediately. Beyond AI, all cloud software stands to benefit from such granular security. Development teams would gain the ability to debug and reason about permissions easily (no more guesswork and lengthy IAM docs trial). Security teams would be able to visualize exactly who can do what in their cloud at a deep level and set **guardrails** that catch abuses instantly [18]. Compliance frameworks would get much more concrete evidence of least-privilege enforcement rather than broad role audits.

In conclusion, adopting graph-based IAM is a timely innovation at the intersection of cloud computing and artificial intelligence. It moves us closer to the ideal of *continuous least privilege* – an environment where every action is performed with exactly the rights it requires and no more. This principle has been known for decades, but technologies like semantic graphs and AI-driven automation are finally making it attainable at scale. By investing in this approach now, cloud providers and security architects can stay ahead of the curve, ensuring that the exciting capabilities of GenAI are delivered **safely and securely**. The era of AI agents calls for a new paradigm in cloud permissions, and graph-based workflows provide the map to get us there. As we often say, *"everything is really just graphs and maps"* [15] – and securing the future of cloud AI is no exception.

Sources: The ideas and research presented in this paper are based on Dinis Cruz's published work on semantic knowledge graphs and open-source projects (such as MGraph-DB [13]) as well as industry best practices and studies on cloud IAM and security [1] [2] [7]. Key references include cloud provider documentation on IAM best practices [1], analyses by cloud security experts highlighting the need for least privilege and just-in-time access [4] [9], and recent innovations in using knowledge graphs for cloud security posture management [5] [3]. These sources reinforce the critical need for a more graph-centric and dynamic approach to permissions in the era of AI-driven automation.

---

[1] [6] Security best practices in IAM - AWS Identity and Access Management
https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html

[2] [4] [9] The Principle of Least Authority: Your Cloud's First Line of Defense | by cyber_pix | Mar, 2025 | Medium
https://medium.com/@use.abhiram/the-principle-of-least-authority-your-clouds-first-line-of-defense-51174e762de2

[3] [10] [18] Enforcing Least Privilege in Multi-Cloud: Manager's Six-step Roadmap
https://www.unosecur.com/blog/a-managers-six-step-roadmap-for-secure-access-across-cloud-environments

[5] Applying Knowledge Graphs to Public Cloud Security | Zscaler
https://www.zscaler.com/blogs/product-insights/applying-knowledge-graphs-public-cloud-security

[7] [11] [12] HackerGraph: Creating a knowledge graph for security assessment of AWS systems
http://kth.diva-portal.org/smash/get/diva2:1844008/FULLTEXT01.pdf

[8] Running with Scissors Secure Coding in C and C++
https://insights.sei.cmu.edu/documents/3820/2013_017_101_293815.pdf

[13] [16] mgraph-db · PyPI
https://pypi.org/project/mgraph-db/

[14] [15] [17] Building Semantic Knowledge Graphs with LLMs: Inside MyFeeds.ai's Multi-Phase Architecture
https://www.linkedin.com/pulse/building-semantic-knowledge-graphs-llms-inside-myfeedsais-dinis-cruz-jub5e