KubeCon | CloudNativeCon
North America 2026

OpenFGA

Fine-Grained Authorization for Apps, APIs, and Agents

Will Vedder, OpenFGA maintainer

OpenFGA™

CLOUD NATIVE COMPUTING FOUNDATION

Poll 👋

Can `jon` deploy `payments-api`?



Can `jon`'s agent deploy the `payments-api` on his behalf?



Challenges answering those questions

1. Where are authz decisions made?
2. How granular are the permissions?
3. How are policies expressed?
4. How are AI agents authorized? 🤖

Challenge 1 - Where are authz decisions made?

```
func canDeploy(user User, svc Service) bool {
    if user.Role == "admin" { return true }
    if user.Team == "sre" { return true }
    if user.Team == svc.OwningTeam { return true }
    if contains(svc.TrustedUsers, user.ID) { return true }
    return false
}
```

Challenge 1 - Where are authz decisions made?

Decentralized ❌

- Logic drifts across services
- "Who can deploy this?" = hard to answer
- No audit trail
- Policy changes = coordinated deploys

Centralized ✅

- One source of truth
- *Decision* separate from *enforcement*
- Every decision is auditable
- Policy changes without redeploys

Challenge 2 - How granular are the permissions?

Coarse-grained 🪨	**Fine-grained** 🧂
"*can_deploy role can deploy all services*"	"*Jon can deploy this service*"
Per role	Per user, per resource
Static	Inheritance is expressible (teams, projects, services)

Challenge 3 - How are policies expressed?

RBAC, ABAC, ReBAC?

RBAC

Role-based access control

- What role you have => what permissions you have
- **Pros:** Simple, easy to audit
- **Cons:** Coarse-grained => role explosion

ABAC

Attribute-based access control

- Authz decision based on attributes of user, resource, and request
- **Pros:** Flexible, handles runtime context
- **Cons:** Reverse-lookup difficult, less auditable, logic scattered in code

ReBAC

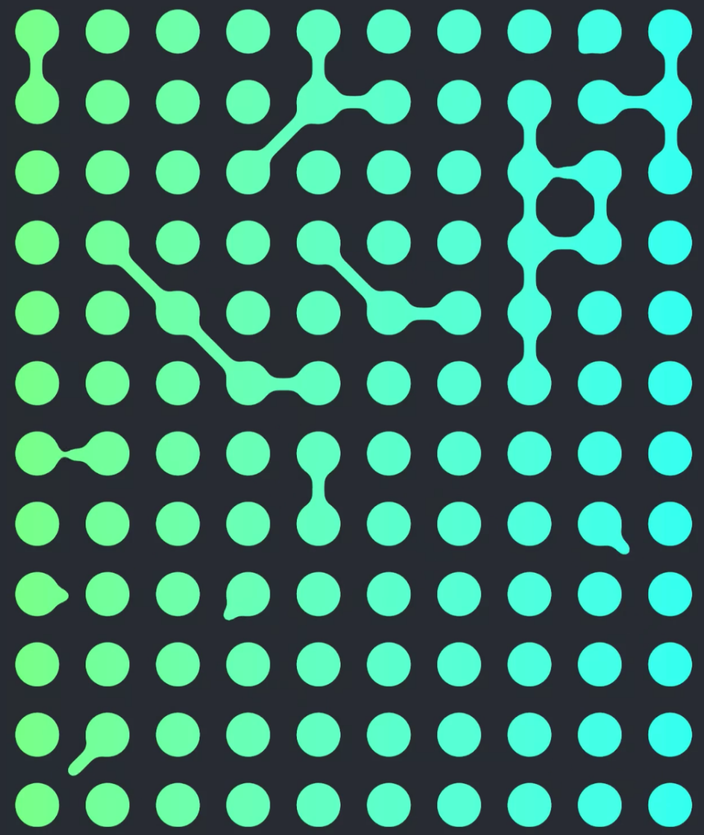
Relationship-based access control

- Access determined by relationships
- Jon is member of `sre` which can deploy `checkout` projects which include `payments-api` service
- **Pros:** Inheritance native, fine-grained, auditable, reverse lookups simple
- **Cons:** Requires writing/backfilling relationships; some complexity w/ graph

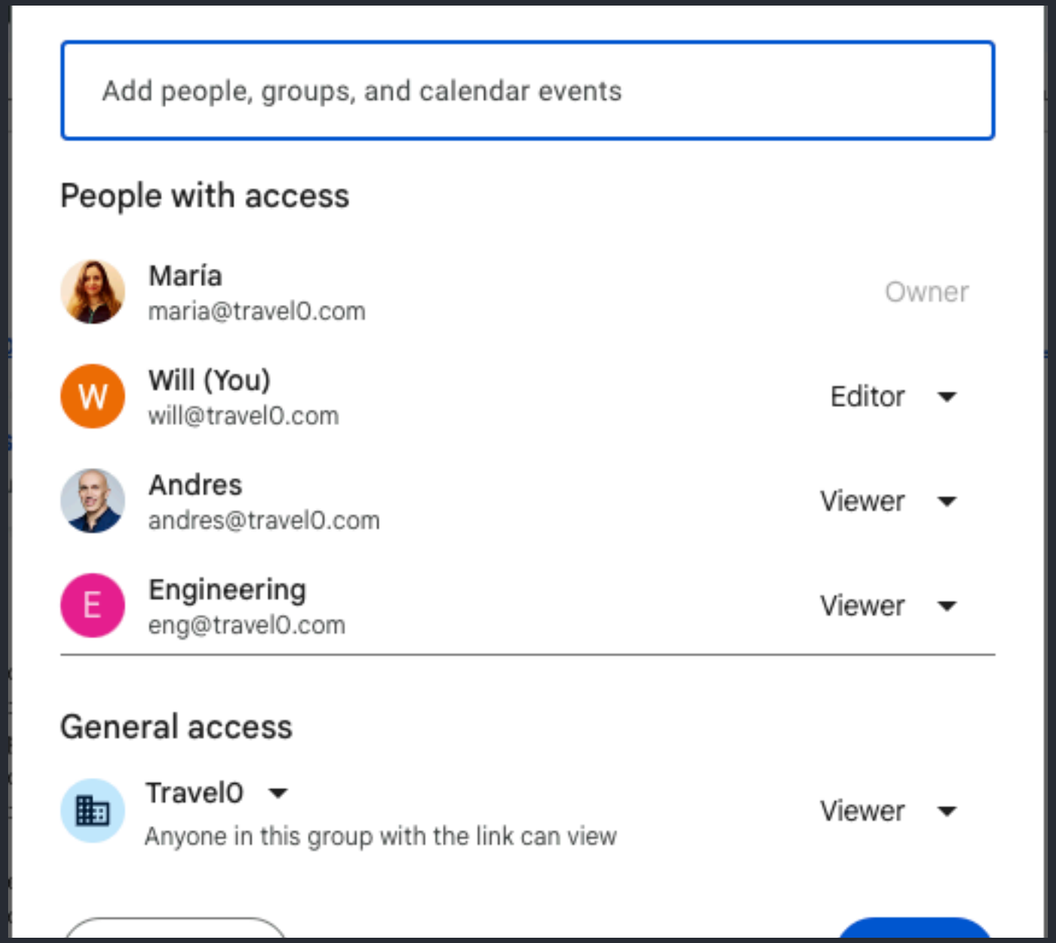
	RBAC	ABAC	ReBAC
Per-resource access	🟡	✅	✅
Inheritance	❌	🟡	✅
Reverse lookup (what can user access?)	✅	❌	✅
Runtime context	❌	✅	❌
Auditing	✅	❌	✅

What is OpenFGA?

- Authorization decision engine
- Google Zanzibar-inspired
- CNCF Incubating
- ReBAC*
- Centralized authz
- Open-source
- 6,100+ stars on GitHub



Fine-grained permissions



📈 OpenFGA Adoption

- Linux Foundation
- Agent Substrate*
- Grafana
- Supabase
- Canonical
- SigNoz
- Caipe IO
- Platform Mesh

openfga/community/ADOPTERS.md

🛠️ OpenFGA Primitives

- **Authorization Model**
  - *The* access control policy
  - Domain-specific language (DSL)
  - Highly expressive
  - Logical operators (and, or, but not)
- **Tuples**
  - The data that informs object-object relationships

^ Two primitives to introduce

Authorization Model

```
model
  schema 1.1

type user

type project
  relations
    define can_deploy: [user]
```

Authorization Model (cont.)

```
model
  schema 1.1

type user

type team
  relations
    define member: [user]

type project
  relations
    define can_deploy: [user, team#member]
```

Authorization Model (cont.)

```
model
  schema 1.1

type user

type team
  relations
    define member: [user]

type project
  relations
    define can_deploy: [user, team#member]

type service
  relations
    define parent: [project]
    define can_deploy: [user, team#member] or can_deploy from parent
```

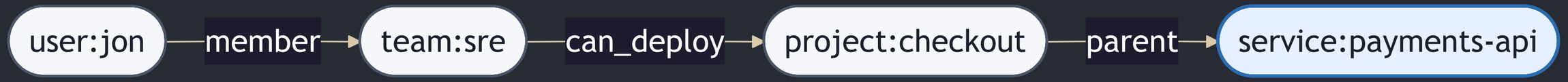
Tuples

- The data that inform object-object relationships
- Paired w/ model = inform authz decisions
- Written by application when events occur (share, invite, delete, add)
- Three parts
  - Object (ex: "project:checkout", "document:roadmap")
  - Relation
  - Subject (ex: "user:jon")
- Not code

Example tuples

user	relation	object
`user:jon`	`member`	`team:sre`
`team:sre#member`	`can_deploy`	`project:checkout`
`project:checkout`	`parent`	`service:payments-api`

Tuples as graph



Can Jon deploy `service:payments-api`?

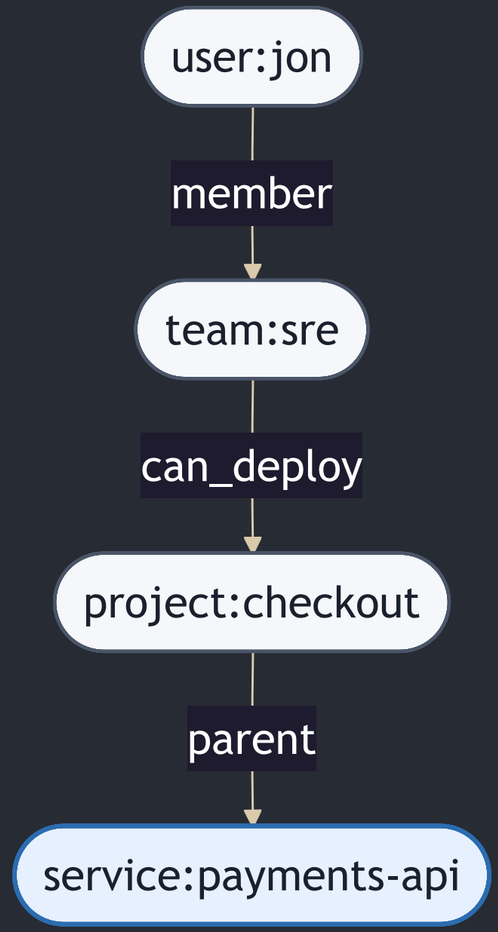
🔍 Query APIs

- **Check** - does *this user* have *this relation* on *this object*?
- **ListObjects** - which objects can *this user* access?
- **ListUsers** - which users can access *this object*?

Check API

```
POST /stores/{store_id}/check
{
  "user": "user:jon",
  "relation": "can_deploy",
  "object": "service:payments-api"
}

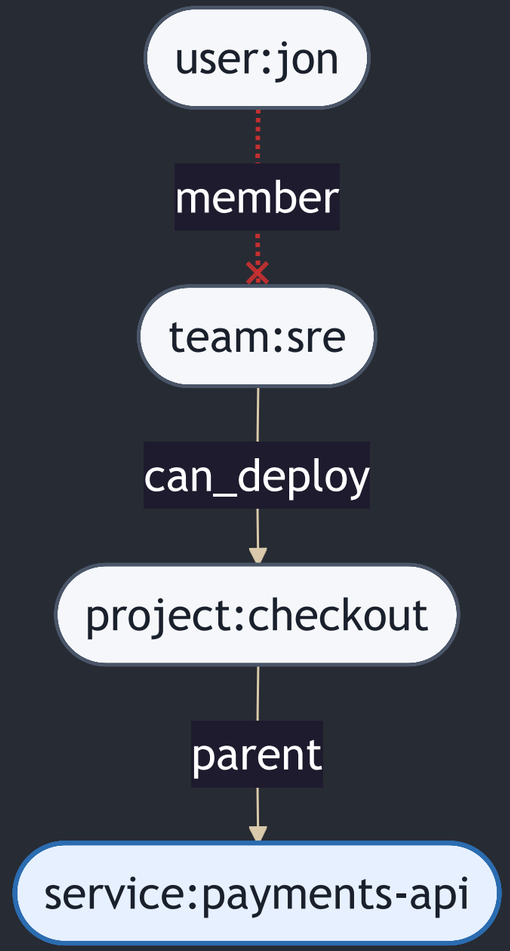
⇒ { "allowed": true }
```



❌ DELETE - user:jon member team:sre

```
POST /stores/{store_id}/check
{
  "user": "user:jon",
  "relation": "can_deploy",
  "object": "service:payments-api"
}
```

⇒ { "allowed": false }



ListObjects API (reverse lookup)

```
POST /stores/{store_id}/list-objects
{
  "relation": "can_deploy",
  "user": "user:jon",
  "type": "service"
}
⇒ [ "service:payments-api", "service:cart-api", "service:tax-proxy" ]
```

 Conditions

Sometimes authz decisions require **context** in addition to relationships.

- OpenFGA's ABAC feature
- CEL expressions
- Allows arbitrary arguments
- Ex:
  - Temporal access
  - Usage-based/feature-based policies (Entitlements)
  - IP allowlists

🚦 Modeling conditions

```
type project
  relations
    define can_deploy: [user with non_expired_grant]


condition non_expired_grant(
  current_time: timestamp,
  start: timestamp,
  duration: duration
) {
  current_time < start + duration
}
```

	RBAC	ABAC	ReBAC	**OpenFGA**
Per-resource access	🟡	✅	✅	✅
Inheritance	❌	🟡	✅	✅
Reverse lookup	✅	❌	✅	✅
Runtime context	❌	✅	❌	✅ (conditions)
Auditing	✅	❌	✅	✅



OpenFGA™

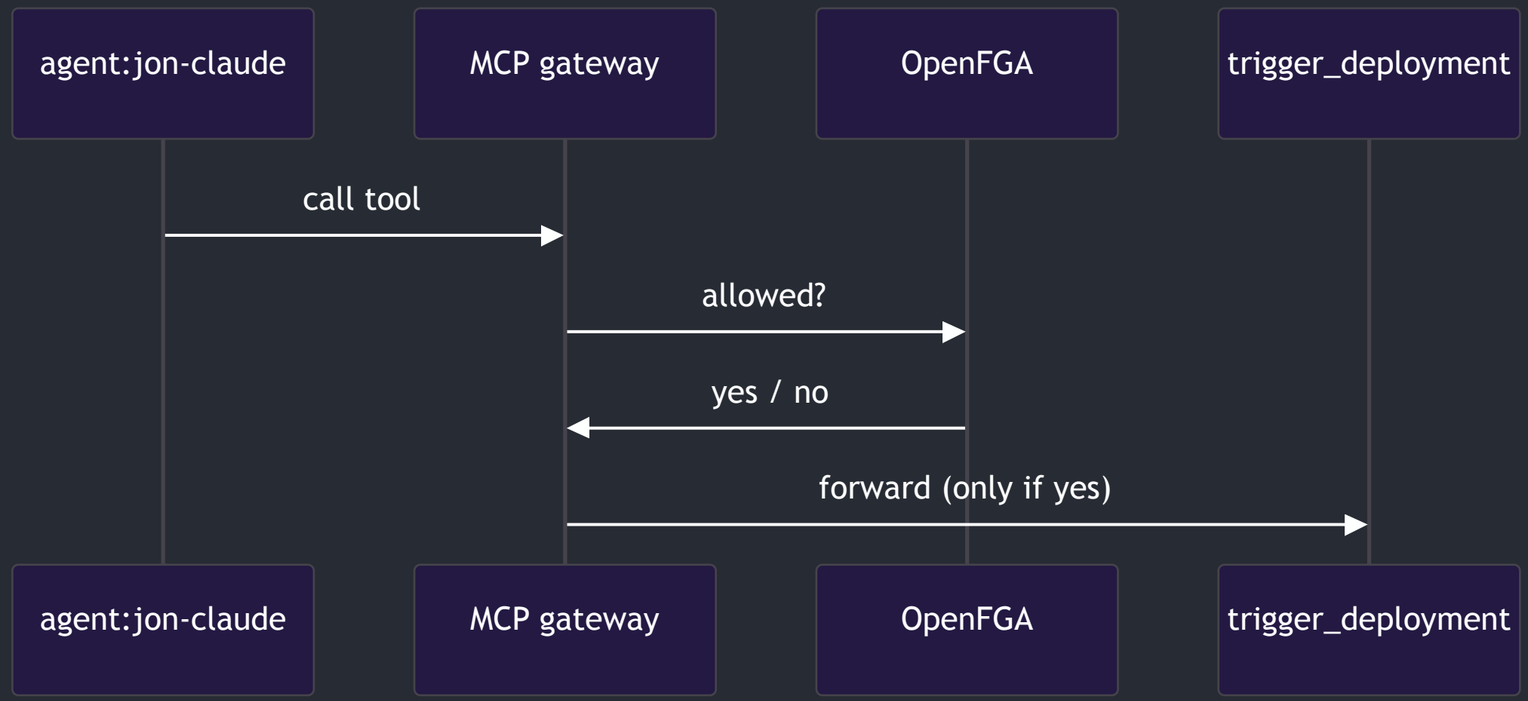
CLOUD NATIVE COMPUTING FOUNDATION

🤖 Challenge 4 - How are AI agents authorized?

- New type of principal
- Acting on behalf of users
- Call tools via MCP
- Agents are sneaky, unpredictable

MCP gateways

Natural agent authorization enforcement point, evaluate access before agent invokes tool.



Modeling agents

```
model
  schema 1.1

type user

type team
  relations
    define member: [user]

type project
  relations
    define can_deploy: [user, team#member]

type service
  relations
    define parent: [project]
    define can_deploy: [user, team#member] or can_deploy from parent

type agent

type tool
  relations
    define can_call: [agent]
```



OpenFGA™

CLOUD NATIVE COMPUTING FOUNDATION

AI Agents - highly dynamic

- Not enough to know if an agent can call a tool
- Heavily dependent on **request's runtime context**
  - Permissions user granted during session
  - Rate-limits
  - Time of day
  - Arbitrary arguments (e.g. `environment: prod`)

Dynamic situation example

Jon's agent should only have permission to deploy non-prod environments

🤔 Why not regular conditions?

- Agent rules can vary per task, per session
- Ex:
  - Jon's agent - non-prod only
  - Maria's agent - `payments-api` only, for 1 hour
  - Andres' agent - max 3 deploys
- New kind of rule = new condition in model
  - Requires coordination, review => slow
  - Users delegating to agents can't wait

```
condition non_prod_only(environment: string) {
  environment ≠ "prod"
}

condition single_service_ttl( … ) { … }

condition max_deploys( … ) { … }
```



Dynamic Conditions

- New OpenFGA feature - built for AI agent authz
- Like conditions, but no model change per rule
- **CEL expressions** attached to tuples, not the model
- Expressions evaluated at query-time

```
define can_call: [agent, agent with $expression]
```

```
user: agent:jon-claude
relation: can_call
object: tool:trigger_deployment
condition:
  name: "$expression"
  context:
    expression: "environment ≠ 'prod'"
    parameters:
      environment: string
```

```
user: agent:jon-claude
relation: can_call
object: tool:trigger_deployment
condition:
  name: "$expression"
  context:
    expression: "environment ≠ 'prod'"
    parameters:
      environment: string
```

```
POST /stores/{store_id}/check
{
  "user": "agent:jon-claude",
  "relation": "can_call",
  "object": "tool:trigger_deployment",
  "context": {
    "environment": "staging"
  }
}
→ { "allowed": true }
```

```
POST /stores/{store_id}/check
{
  "user": "agent:jon-claude",
  "relation": "can_call",
  "object": "tool:trigger_deployment",
  "context": {
    "environment": "prod"
  }
}
→ { "allowed": false }
```



OpenFGA™

CLOUD NATIVE COMPUTING FOUNDATION

⚠️ Hand-waviness

Just because Jon's agent can deploy doesn't mean that Jon has permissions to deploy too.

Either need to:

- Two checks
  - Jon's agent `can_call` tool *and*
  - Jon's `can_deploy` access
- Express directly in authorization model

```
model
  schema 1.1

type user

type agent
  relations
    define owner: [user]

type team
  relations
    define member: [user, agent]

type project
  relations
    define can_deploy: [user, team#member]

type service
  relations
    define parent: [project]
    define can_deploy: [user, team#member] or can_deploy from parent

type tool
  relations
    define service: [service]
    define can_call: [agent, agent with $expression]
    define can_deploy: can_call and can_deploy from service
```

🤝 Conclusion - OpenFGA

1. **Centralized** decision engine 🌐
2. Fine-grained 🧂
3. Best of ReBAC, ABAC, RBAC 💪
4. Agents as first-class principals w/ dynamic conditions 🤖

Open-source - CNCF Incubating - SDKs, CLIs, Helm charts - Widely adopted - ... much more

📚 Continued exploration

- Playground
- Docs
- Sample implementations (incl. MCP Gateway examples)
- Dynamic Conditions Announcements:
- Community

Questions?



OpenFGA™



CLOUD NATIVE
COMPUTING FOUNDATION