Here are a couple of excellent alternatives in PostgreSQL that achieve similar "enum-like" behavior, each with its own trade-offs:

1. Using `TEXT` with a `CHECK` Constraint

This is often the preferred method for many developers due to its flexibility. You define the column as a `TEXT` (or `VARCHAR`) type, and then add a `CHECK` constraint to restrict the values it can hold.

How to implement:

```
ALTER TABLE your_table_name
ADD COLUMN tipo_usuario TEXT NOT NULL;

ALTER TABLE your_table_name
ADD CONSTRAINT chk_tipo_usuario
CHECK (tipo_usuario IN ('Coordenador', 'Control Materiais', 'Administrador'));
```

Explanation:

- `tipo_usuario TEXT NOT NULL`: Defines the column as a text type, ensuring it cannot be null.
- `CHECK (tipo_usuario IN (...))`: This is the crucial part. It enforces that any value inserted into `tipo_usuario` must be one of the specified strings. If you try to insert anything else, PostgreSQL will throw an error.

Advantages:

- **Flexibility:** Easily add or remove values by simply altering the `CHECK` constraint. This is usually a much lighter operation than altering a custom `ENUM` type, which can sometimes require rewriting the table.
- **Standard Data Type:** Works with standard `TEXT` operators and functions.
- **Self-documenting:** The allowed values are directly visible in the table's definition, making it easy to understand.
- **No custom types:** You don't need to define a separate `TYPE` object, simplifying your schema.

Disadvantages:

- **No type safety outside the column:** While the column enforces the values, if you pass these values around in your application code, they are just strings. With a custom `ENUM` type, the database's type system provides a stronger guarantee.
- **Slightly less space efficient:** Stores the full text string for each row, whereas `ENUM` types internally store a smaller representation (like an integer OID). However, for typical "enum" scenarios, this difference is often negligible.
- **No inherent ordering:** Unlike a native `ENUM` (which maintains the order in which values were declared), a `CHECK` constraint on `TEXT` doesn't

inherently define an order. If you need a specific order for sorting, you'd need to handle that in your queries or application logic.

2. Using a Lookup Table with a `FOREIGN KEY` Constraint

This is the most normalized and robust approach, especially if your "enum" values might grow, change frequently, or have associated metadata.

How to implement:

a. Create the lookup table:

```
CREATE TABLE tipo_usuario_lookup (
    id SERIAL PRIMARY KEY, -- Or use a TEXT primary key if you prefer the text itself as the
    nome_tipo VARCHAR(50) UNIQUE NOT NULL
);

INSERT INTO tipo_usuario_lookup (nome_tipo) VALUES
('Coordenador'),
('Control Materiais'),
('Administrador');
```

b. Add the column to your main table with a foreign key:

```
ALTER TABLE your_table_name
ADD COLUMN tipo_usuario_id INTEGER NOT NULL; -- Or TEXT if you used text as PK in lookup

ALTER TABLE your_table_name
ADD CONSTRAINT fk_tipo_usuario
FOREIGN KEY (tipo_usuario_id) REFERENCES tipo_usuario_lookup (id);
```

Explanation:

- `tipo_usuario_lookup`: A separate table that holds all the valid "Tipo Usuario" values.
- `id SERIAL PRIMARY KEY`: A unique identifier for each type.
- `nome_tipo VARCHAR(50) UNIQUE NOT NULL`: Stores the actual name of the user type and ensures it's unique.
- `tipo_usuario_id INTEGER NOT NULL`: The column in your main table that will store the `id` from the `tipo_usuario_lookup` table.
- `FOREIGN KEY (...) REFERENCES ...`: This constraint ensures that only valid `id` values (from the `tipo_usuario_lookup` table) can be inserted into `tipo_usuario_id`.

Advantages:

- **Maximum Flexibility:** Adding or removing types is a simple `INSERT` or `DELETE` statement on the `tipo_usuario_lookup` table. No schema changes are required for new values.

- **Normalization:** Follows database normalization principles, avoiding data redundancy.
- **Associated Metadata:** You can easily add more columns to `tipo_usuario_lookup` (e.g., `description`, `permissions_level`, `is_active`) to store additional information about each user type.
- **Clear Relationships:** The foreign key explicitly shows the relationship between your main table and the valid user types.

Disadvantages:

- **Requires a Join:** When you want to retrieve the actual `nome_tipo` (e.g., 'Coordenador'), you'll need to `JOIN` with the `tipo_usuario_lookup` table. This adds a slight overhead to queries.
- **More Complex Setup:** Requires creating an additional table and managing foreign key relationships.
- **Less "Direct":** You're storing an ID, not the human-readable string, in your main table.

Which Option to Choose?

- **For simple, static lists (rarely change, no associated data):** PostgreSQL's native `ENUM` (your initial thought) is perfectly fine and provides good type safety.
- **For lists that might change, but don't need associated data, and you prefer direct string storage:** The `TEXT` column with a `CHECK` constraint is a strong contender due to its flexibility. This is a very common and pragmatic approach.
- **For lists that are likely to grow, change frequently, or need associated attributes:** The **lookup table with a `FOREIGN KEY`** is the most robust and scalable solution.

Given your concern about PostgreSQL not having a *native* `ENUM` in the way some other databases do (implying you want more flexibility than the built-in `CREATE TYPE AS ENUM` provides), I would lean towards either the **`TEXT` with `CHECK` constraint** or the **Lookup Table with `FOREIGN KEY`**. The `TEXT` with `CHECK` is simpler to implement for this specific case, while the lookup table is more robust for future expansion.